Università degli studi di Roma Tre
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea in Matematica

Tesi di Laurea in Matematica di

Patrizia Tropea

Immersione di grafi planari su griglie

Relatore
Prof. Marco Liverani

Il Candidato Il Relatore

Anno Accademico 2000/2001
Luglio 2001

Classificazione AMS: 68C05, 68C25, 68E10
Parole chiave: grafi, grafi planari, ottimizzazione, circuiti VLSI

Indice

Introduzione

In questa tesi considereremo il problema di ottimizzazione relativo alla costruzione di un disegno, composto da segmenti rettilinei verticali ed orizzontali su una griglia, di un grafo planare G con n vertici. Un grafo, come vedremo nelle pagine seguenti, è planare se è possibile *immergerlo* nel piano, senza creare intersezioni tra gli spigoli. Spesso nella pratica è necessario che l'immersione non avvenga in un piano generico, dove sono ammissibili anche archi "curvi": è richiesto che il disegno del grafo avvenga all'interno di una griglia rappresentando gli spigoli mediante una linea spezzata che consista in una sequenza alternata di segmenti verticali ed orizzontali. Questo vincolo è molto forte, visto che, ad esempio, non è possibile disegnare su una griglia bidimensionale grafi che abbiano vertici di grado maggiore di 4.

In particolare verrà considerato il problema caratterizzato da un ulteriore vincolo sul numero di segmenti ortogonali fra loro mediante i quali è possibile rappresentare ogni singolo spigolo del grafo: il numero di pieghe (ovvero curve di 90^0, nel seguito chiamate *bend*) non potrà essere superiore a tre o a due.

Analizzeremo in questo contesto due algoritmi: il primo che verifica la planarità di un grafo; il secondo che costruisce una immersione su griglie bidimensionali di un grafo planare con al più due o tre pieghe degli spigoli. Tutto questo trova naturalmente diverse applicazioni nella pratica ed in particolare nella progettazione di circuiti elettronici ad altissima integrazione (VLSI – *Very Large Scale Integration*). Nella progettazione di questo genere di circuiti

è cruciale poter trovare una rappresentazione planare del grafo che rappresenti il circuito, limitando al massimo il numero di pieghe dei singoli segmenti del circuito stesso. Un altro problema applicativo legato alla progettazione di circuiti elettronici è costituito dal fatto che non tutti i circuiti (cioè non tutti i grafi che li rappresentano) sono *planari*, ovvero non tutti i grafi, pur essendo planari, possono essere rappresentati su una griglia bidimensionale (pensiamo ad esempio a grafi con vertici di grado maggiore di 4). Esistono quindi grafi (circuiti elettronici) che non possono essere rappresentati su un piano a meno di intersecare fra loro due o più spigoli. Nella realizzazione di un circuito elettronico i segmenti di circuito che connettono fra loro le componenti non possono intersecarsi. È necessario allora disporre il circuito su più strati sovrapposti: su ogni strato (*layer*) sarà rappresentata una porzione planare del circuito (grafo) e saranno aggiunti dei connettori (spigoli del grafo) che consentiranno il passaggio del segnale elettrico tra uno strato e l'altro del circuito. In questo caso il problema di ottimizzazione richiede di minimizzare il numero di strati su cui vengono disposte le porzioni planari di circuito. Nell'ultima parte della tesi vengono quindi affrontati in termini piuttosto generali alcuni metodi standard per la soluzione di tale problema.

La tesi si articola su tre capitoli; in appendice sono presentati alcuni esempi ed i programmi in linguaggio C che codificano gli algoritmi analizzati. Nel primo capitolo, dopo aver definito una notazione ed una terminologia uniforme, forniremo le definizioni e le proprietà fondamentali, riutilizzate in seguito. Quindi descriveremo un algoritmo per la verifica della planarità di un grafo, proposto da Hopcroft e Tarjan [11]. Questo algoritmo utilizza l'approccio della visita in profondità per stabilire l'ordine con cui dovranno essere eseguite le operazioni sui vertici e sugli spigoli del grafo e per aumentare l'efficienza complessiva dell'algoritmo stesso. Ha una complessità computazionale pari a $O(|V|)$, ed anche la quantità di memoria utilizzata è lineare nel numero dei vertici del grafo G.

Nel secondo capitolo descriveremo un algoritmo che, dato un grafo planare G con $|V| = n$ vertici, costruisca una immersione per G sulla griglia bidimensionale tale che ogni arco abbia al più due piegature (eccetto l'ottaedro per il quale sono necessarie tre piegature) [20]. Il numero totale di piegature è al più $2n+4$ se il grafo è biconnesso e al più $(\frac{7}{3})n$ nel caso generale. L'area occupata dal disegno è $(n+1)^2$ nel caso peggiore.

Nel terzo capitolo, infine, prenderemo in esame alcuni metodi per la rappresentazione di grafi su griglie tridimensionali, disponendo porzioni di grafo su *layer* sovrapposti, cercando di minimizzare il numero dei *layer*.

Capitolo 1

Grafi planari e test di planarità

In questo capitolo presenteremo un algoritmo efficiente per determinare se un grafo G può essere immerso nel piano senza che si verifichino intersezioni tra gli archi. Nel paragrafo 1.1 introdurremo alcuni concetti di base relativi alla teoria dei grafi, necessari per comprendere l'algoritmo di planarità. Nel paragrafo 1.2 riporteremo la definizione di planarità ed alcuni esempi di grafo planare e non planare, presentando inoltre la formula di Eulero e il Teorema di Kuratowski. Nel paragrafo 1.3 illustreremo l'algoritmo di planarità: innanzitutto spiegando l'idea che c'è dietro, poi sviluppandone i dettagli ed infine riportando l'intero algoritmo.

1.1 Preliminari

Un *grafo orientato* G è una coppia (V, E), dove V è un insieme finito ed E una relazione binaria su V. L'insieme V è chiamato *l'insieme dei vertici*, l'insieme E è chiamato *l'insieme degli archi* di G.

In un grafo *non orientato* $G = (V, E)$ l'insieme degli archi E è costituito da coppie non ordinate di vertici piuttosto che da coppie ordinate: (u, v) e (v, u)

vengono considerati lo stesso arco. Se (u, v) è un arco di un grafo orientato, si dice che (u, v) *esce dal* vertice u ed *entra nel* vertice v. Il *grado* di un vertice in un grafo orientato è il numero di archi entranti più il numero di archi uscenti.

Si dice che un grafo $G' = (V', E')$ è un *sottografo* di $G = (V, E)$ se $V' \subseteq V$ ed $E' \subseteq E$. Dato un grafo orientato $G = (V, E)$ la *versione non orientata* di G è il grafo non orientato $G' = (V', E')$, dove $(u, v) \in E'$ se e solo se $u \neq v$ e $(u, v) \in E$.

Si dice che un grafo $G' = (V', E')$ è una *suddivisione* di $G = (V, E)$ se G' è ottenuto da G sostituendo gli archi con cammini che hanno al più i vertici finali in comune.

Figura 1.1: Una suddivisione di (a) K_5 e (b) di $K_{3,3}$.

Un *cammino* da un vertice u a u' è una sequenza di vertici $< v_0, v_1, v_2, ..., v_k >$ tale che $u = v_0$, $u' = v_k$ e $(v_{i-1}, v_i) \in E$ per $i = 1, 2, ..., k$. Un cammino p è *semplice* se tutti i vertici sono distinti. Se esiste un cammino da u a v si scrive $p : u \rightsquigarrow v$. In un grafo orientato un cammino $< v_0, v_1, v_2, ..., v_k >$ forma un *ciclo* se $v_0 = v_k$ e il cammino contiene almeno un arco. Il ciclo è *semplice* se $v_1, v_2, ..., v_k$ sono distinti.

Un grafo non orientato è *connesso* se ogni coppia di vertici è collegata da un cammino. Le *componenti connesse* di un grafo sono le classi di equivalenza dei vertici sotto la relazione "raggiungibile da". Se G contiene tre vertici

distinti x, v, w tali che w è raggiungibile da v, ma ogni cammino $p : v \rightsquigarrow w$ contiene x, allora x è un *punto di articolazione* di G. Se G è connesso e non contiene punti di articolazione allora G è *biconnesso*. Una *componente biconnessa* di G è un insieme massimale di archi tale che due archi qualunque dell'insieme giacciono su di uno stesso ciclo semplice. Un grafo non orientato è *connesso* se è costituito da un'unica componente connessa, cioè se ogni vertice è raggiungibile da ogni altro vertice.

Un grafo orientato è *fortemente connesso* se ogni coppia di vertici è collegata da un cammino. Le componenti fortemente connesse di un grafo sono le classi di equivalenza dei vertici sotto la relazione "essere mutuamente raggiungibili". Un grafo orientato è *fortemente connesso* se è composto da un'unica componente fortemente connessa.

Un *albero* orientato radicato T è un grafo orientato in cui uno dei vertici r chiamato *radice* si distingue dagli altri in modo tale che ogni altro vertice in T è raggiungibile da r, nessun arco entra in r ed un solo arco entra in ogni altro vertice di T. La relazione "(u, v) è un arco in T" viene indicata con $u \rightarrow v$, la relazione "esiste un cammino da u a v in T" è indicata con $u \rightsquigarrow v$.

Se $u \rightarrow v$ è un arco di T allora u è il *padre* di v e v è il *figlio* di u. Se esiste un cammino da u a v allora diremo che u è un *antenato* di v e v è un *discendente* di u. Se $u \rightsquigarrow v$ e $u \neq v$, u è un *antenato proprio* di v e v è un *discendente proprio* di v.

Se T_1 è un albero ed è un sottografo di un albero T_2, allora T_1 è un *sottoalbero* di T_2. Se T è un albero che è un sottografo di un grafo orientato G e T contiene tutti i vertici di G, allora T è un *albero ricoprente* di G.

1.2 Planarità

Un grafo G è *planare* se e solo se esiste un'applicazione che manda i vertici e gli archi di G nel piano in modo tale che:

1. ogni vertice viene mandato in un punto distinto;
2. ogni arco (v, w) viene mandato in una curva semplice che ha come estremi v e w;
3. l'immagine di archi diversi ha intersezione vuota eccetto che per i vertici in comune.

Figura 1.2: (a) Un grafo G; (b) una immersione planare di G.

Consideriamo l'esempio in figura 1.2(a) e chiediamoci se è planare, ovvero ci chiediamo se possiamo disegnare il grafo nel piano collocando in modo opportuno i vertici e gli archi senza che ci siano intersezioni di archi eccetto che per i vertici in comune. Il disegno in figura 1.2(a) ha due intersezioni che possono essere eliminate collocando il vertice v_6 all'esterno del poligono $v_1v_2v_3v_4$ come mostrato in figura 1.2 (b).

Figura 1.3: (a) I grafi di Kuratowski: K_5 ;(b) $K_{3,3}$.

Non tutti i grafi sono planari. Ad esempio consideriamo il grafo K_5, il grafo completo con cinque vertici, (figura 1.3(a)) e chiediamoci se è planare. Supponiamo che lo sia; allora possiamo assumere senza perdita di generalità che $v_1v_2v_3v_4v_5$ possa essere disegnato nel piano come un pentagono regolare. L'arco (v_1, v_3) può essere disegnato all'interno del pentagono; allora entrambi gli archi (v_2, v_5) e (v_2, v_4) devono essere disegnati all'esterno e di conseguenza l'arco (v_3, v_5) deve essere disegnato all'interno. Allora ci sarà una intersezione quando l'arco (v_1, v_4) verrà disegnato all'interno oppure all'esterno (figura 1.4 (a)). Quindi possiamo concludere che K_5 non è planare. Un altro esempio di grafo non planare è il grafo completo bipartito $K_{3,3}$ (figura 1.3 (b)). Anche in questo caso supponiamo di disegnare l'arco (u_1, v_2) all'interno dell'esagono $u_1v_1u_2v_2u_3v_3$ e quindi l'arco (v_1, u_3) all'esterno. Allora (u_2, v_3) non può essere disegnato senza produrre una intersezione (figura 1.4 (b)). Dunque anche $K_{3,3}$ non è planare.

1.2.1 Formula di Eulero

Sia $G = (V, E)$ un grafo planare connesso con $|V| > 2$. Una *faccia* di G è una parte massimale del piano tale che, per ogni coppia di punti x e y al suo interno, esiste una linea continua da x a y che non ha punti in comune con

Figura 1.4: (a) Immersione parziale di K_5 e (b) di $K_{3,3}$.

la rappresentazione planare di G. Una delle facce di G ha area infinita, ed è chiamata la *faccia esterna* del grafo.

Teorema 1.1. *Sia G un grafo planare connesso e siano n, m, f rispettivamente il numero di vertici, archi e facce di G. Allora si ha*

$$n - m + f = 2$$

(formula di Eulero).

Dim. Dimostriamo la formula di Eulero per induzione su m. Il risultato è ovvio per $m = 0$ e $m = 1$. Supponiamo che la formula sia vera per tutti i grafi planari connessi che hanno meno di m archi, dove $m \geq 2$ e supponiamo che G abbia m archi. Consideriamo prima il caso in cui G è un albero. Allora G ha un vertice v di grado 1. Il grafo planare connesso $G - v$ ha $n-1$ vertici, $m-1$ archi ed f facce, cosí, per ipotesi induttiva, si ha $(n-1) - (m-1) + f = 2$ quindi $n - m + f = 2$. Consideriamo ora il caso in cui G non è un albero.

Allora esiste un arco e di G che appartiene ad un ciclo. In questo caso il grafo planare connesso $G - e$ ha n vertici, $m - 1$ archi e $f - 1$ facce; allora la formula di Eulero deriva dall'ipotesi induttiva. □

Un grafo *planare massimale* è un grafo a cui non può essere aggiunto nessun arco senza perdere la planarità. Per questo in ogni immersione di un grafo planare massimale G con $n \geq 3$ si ha che ogni faccia di G è un triangolo. Sebbene un grafo generico possa avere fino a $\frac{n(n-1)}{2}$ archi questo non è vero per un grafo planare.

Corollario 1.1. *Sia G un grafo planare con $n \geq 3$ vertici e m archi allora si ha $m \leq 3n - 6$.*

Dim. Possiamo assumere senza perdita di generalità che G sia un grafo planare massimale; altrimenti possiamo aggiungere nuovi archi senza aumentare il numero dei vertici in modo tale che il grafo risultante sia planare massimale. Consideriamo una immersione planare di G. Ogni faccia è un triangolo e ogni arco appartiene esattamente a due facce. Allora si ha che $3f = 2m$. Applicando la formula di Eulero si ha che $m = 3n - 6$. □

1.2.2 Teorema di Kuratowski

Uno dei teoremi più interessanti della teoria dei grafi è quello di Kuratowski. Dal momento che sia K_5 che $K_{3,3}$ sono non planari ogni grafo planare non può contenere una suddivisione di K_5 o $K_{3,3}$; sorprendentemente anche il viceversa è vero.

Teorema 1.2. *Un grafo G è planare se e solo se non contiene una suddivisione di K_5 o $K_{3,3}$.*

La dimostrazione di questo Teorema non viene riportata perché richiede la

dimostrazione di Lemmi di carattere generale che esulano dalla verifica della planarità. Si veda [5].

1.3 Test di planarità

In questa sezione viene presentato un algoritmo che determina se un grafo G può essere immerso nel piano senza che si verifichino intersezioni tra gli archi eccetto che per i vertici in comune. La prima caratterizzazione dei grafi planari è stata data dal Teorema di Kuratowski ma, sebbene questa condizione sia molto elegante, non è utile per verificare la planarità di un grafo perché l'algoritmo che verifica se K_5 o $K_{3,3}$ sono una suddivisione di un grafo G può raggiungere una complessità almeno di $O(|V|^6)$ per alcuni grafi orientati. L'approccio migliore per la risoluzione di questo problema sembra quello di tentare di costruire una immersione planare di un grafo dato. Se questa rappresentazione può essere completata allora il grafo è planare altrimenti non lo è. Abbiamo bisogno di un Lemma sui grafi planari.

Figura 1.5: Illustriamo il Lemma 1.3. Il cammino p_2 è all'interno della curva semplice chiusa formata da p_1 e p_3.

Lemma 1.3. *Sia G un grafo planare immerso nel piano. Per semplicità identifichiamo ogni arco di G con la sua immersione. Siano p_1, p_2, p_3 tre cammini che vanno da x a y tali che a due a due abbiano solo x e y come vertici in*

comune. Siano $(x, v_1), (x, v_2), (x, v_3)$ *i primi archi di* p_1, p_2, p_3 *rispettivamente e siano* $(w_1, y), (w_2, y), (w_3, y)$ *gli ultimi archi di* p_1, p_2, p_3*. Se l'ordine in cui gli archi vengono disegnati attorno ad* x *nel piano è* $(x, v_1), (x, v_2), (x, v_3)$ *allora l'ordine in cui gli archi vengono disegnati attorno ad* y *sarà* (w_1, y)*,* (w_2, y)*,* (w_3, y)*.*

La dimostrazione di questo Lemma che omettiamo per brevità è riportata in [10, 28]. Questo Lemma è un corollario del Teorema della Curva di Jordan che riportiamo di seguito:

Teorema 1.4. *Se* C *è l'immagine di una curva continua, semplice e chiusa in* R^2 *allora il complementare di* C *in* R^2 *ha esattamente due componenti connesse, una (l'interno) limitata e l'altra (l'esterno) illimitata, tali che* C *è la frontiera di entrambe.*

La dimostrazione di questo Teorema che omettiamo per brevità è riportata in [10, 28].

Per il corollario 1.1 dato un grafo planare connesso G con $|V| \geq 3$ si ha che $|E| \leq 3|V| - 6$. Per questo motivo è possibile verificare la planarità di un grafo con un algoritmo che ha una complessità lineare nel numero dei vertici. Un modo per rappresentare un grafo in un computer è quello di usare una *matrice di adiacenza* M, ossia una matrice quadrata di ordine $n = |V|$ in cui $m_{i,j} = 1$ se (i, j) è un arco e $m_{i,j} = 0$ altrimenti. Lo spazio (quantità di memoria) necessario per memorizzare una matrice di adiacenza è $O(|V|^2)$; si può dimostrare rigorosamente che il problema della verifica della planarità di un grafo richiede il controllo di ogni elemento della matrice e che questa operazione ha una complessità proporzionale almeno a $|V|^2$. Per questo motivo utilizzeremo una differente rappresentazione del grafo G utilizzando delle *liste di adiacenza*: per ogni vertice v di G la lista di adiacenza $A(v)$ conterrà i vertici w adiacenti a v.

Spesso gli algoritmi che operano sui grafi richiedono di esplorare un grafo in modo sistematico. Anche nel nostro caso, come vedremo in seguito, è necessario compiere una visita del grafo. Per farlo risulta conveniente sfruttare l'algoritmo di *visita in profondità*: gli archi vengono esplorati a partire dall'ultimo vertice scoperto v che abbia ancora degli archi inesplorati uscenti da esso. Quando tutti gli archi di v sono stati esplorati, la visita torna indietro per esplorare gli archi uscenti dal vertice da cui v era stato scoperto. Questo processo continua finché non vengono raggiunti tutti i vertici che sono raggiungibili dal vertice sorgente s. Se rimane qualche vertice non visitato, allora uno di essi viene selezionato come nuovo vertice sorgente e la visita viene ripetuta a partire da tale vertice: l'intero processo viene ripetuto finché non vengono raggiunti tutti i vertici.

Se G è un grafo non orientato, una visita in profondità impone una orientazione su ogni arco di G data dalla direzione in cui l'arco è stato percorso durante la visita. Per questo motivo la visita in profondità trasforma un grafo non orientato G in un grafo orientato G'.

La visita inoltre divide gli archi in due classi: un insieme di *archi dell'albero* che definiscono l'*albero ricoprente* di G' ed un insieme di *archi di ritorno* (v, w) tali che esiste un cammino p da w a v in T. Un *arco di ritorno* (v, w) è indicato con $v^- \rightarrow w$. Un grafo orientato G', partizionato in questo modo, è chiamato *palm tree*. La scelta dell'algoritmo di visita in profondità per l'esplorazione del grafo è importante perché la struttura dei cammini in un *palm tree* prodotto da questo tipo di visita è molto semplice.

Per implementare la prima visita in profondità di un grafo non orientato connesso usiamo la semplice procedura ricorsiva descritta in precedenza. Questa procedura scorre le liste di adiacenza dei vertici del grafo e l'ordine in cui i vertici vengono visitati dipende dall'ordine in cui sono accodati gli archi nelle liste di adiacenza; i vertici vengono numerati da 1 a n, dove n è il numero

di vertici di G, nell'ordine in cui essi sono stati visitati e gli archi vengono classificati in *archi di ritorno* e *archi dell'albero*.

```
DFS(G = (V, E))
procedura per la visita in profondità di in grafo G rappresentato attraverso liste di
adiacenza A(v); m è l'ultimo numero assegnato ad un vertice
1   for i := 1 to n
2      NUMBER(i):= 0
3   end for
4   m := 0
5   la visita inizia dal vertice 1
6   VISITA (1, 0, m)
```

```
VISITA(v, u, m)
il vertice u è il padre di v nell'albero ricoprente che stiamo costruendo
1   m := m + 1
2   NUMBER(v):= m
3   for each w ∈ A(v)
4      if NUMBER(w)= 0
5         w è un nuovo vertice
6         (v, w) è un arco dell'albero
7         VISITA (w, v, m)
8      else if NUMBER(w)<NUMBER(v) and w ≠ u
9         (v, w) è un arco di ritorno
10     end if
11  end for
```

Lemma 1.5. *La procedura* DFS *esegue correttamente la visita in profondità di un grafo non orientato e ha una complessità di* $O(|V| + |E|)$ *dove* $|V|$ *è il numero di vertici ed* $|E|$ *il numero di archi di* G*. I vertici vengono numerati in modo tale che se* (v, w) *è un arco dell'albero allora* NUMBER(v) <NUMBER(w) *e se* (v, w) *è un arco di ritorno allora* NUMBER(w) <NUMBER(v).

La dimostrazione di questo Lemma che omettiamo per brevità è riportata in [24].

Figura 1.6: (a) Un grafo G; (b) un *palm tree* generato da G. Gli archi tratteggiati sono archi di ritorno, gli altri sono archi dell'albero. I vertici sono numerati secondo l'ordine in cui sono stati visitati nella procedura ORDINA.

1.3.1 Algoritmo di planarità.

In questo paragrafo parleremo dell'idea che è alla base dell'algoritmo di planarità; successivamente ne discuteremo i dettagli.

Il primo passo di questo algoritmo sarà quello di contare il numero di archi in un grafo e se questo numero è maggiore di $3|V| - 6$ concluderemo che il grafo è non planare per il Corollario 1.1. Successivamente divideremo il grafo in componenti biconnesse: osserviamo infatti che un grafo è planare se e solo se tutte le sue componenti biconnesse sono planari. A questo punto verificheremo la planarità di ogni componente. Per farlo applicheremo l'algoritmo di visita in profondità, trasformando il grafo in un *palm tree* P e numerando i

suoi vertici. Quindi, utilizzando una procedura che chiameremo PATHFINDER, l'algoritmo troverà un ciclo nel grafo e lo eliminerà, producendo in questo modo un insieme di componenti disconnesse. Successivamente si verificherà la planarità di ogni componente più il ciclo originale(applicando la procedura ricorsivamente) e si determinerà in che modo immergere queste componenti al fine di costruire una rappresentazione planare dell'intero grafo.

Esaminiamo separatamente il processo di ricerca un ciclo nel grafo e quello di verifica della planarità. Ogni chiamata ricorsiva dell'algoritmo richiede che nella componente venga trovato un ciclo da verificare per la planarità. Questo ciclo consiste in un cammino semplice di archi che non appartengono a cicli trovati precedentemente, più un cammino semplice di archi nei vecchi cicli. Usiamo la visita in profondità per dividere il grafo in cammini semplici che possono essere assemblati in cicli necessari per il test di planarità. Abbiamo bisogno di una seconda visita per trovare i cammini perché la ricerca dei cammini deve essere portata avanti in un ordine speciale se si vuole che il test di planarità sia efficiente. Successivamente descriveremo il processo di ricerca dei cammini in dettaglio e proveremo alcune proprietà dei cammini generati.

Ad esempio supponiamo di aver trovato il primo ciclo c. Esso consisterà in una sequenza di archi dell'albero seguiti da un arco di ritorno in P. Il numero assegnato ad ogni vertice è tale che i vertici sono ordinati lungo il ciclo. Ogni componente che non appartiene al ciclo consisterà in un singolo arco di ritorno oppure in un arco dell'albero, più un sottoalbero con radice w, più tutti gli archi di ritorno che partono dal sottoalbero. Esaminiamo le componenti ed aggiungiamole alla rappresentazione planare in ordine di v decrescente. Ogni componente può essere posizionata all'interno oppure all'esterno di c grazie al Teorema della Curva di Jordan. Quando aggiungiamo una componente alla rappresentazione planare altre parti del grafo dovranno essere spostate dall'interno all'esterno e viceversa (figura 1.7). Continuiamo

Figura 1.7: (a) Conflitto tra due componenti; (b) per immergere la componente S_3 e mantenere la planarità dobbiamo spostare S_1 e immergere S_3 all'interno di c.

ad aggiungere nuove componenti e a spostare le vecchie, se necessario, fino a quando una componente non può più essere aggiunta, oppure l'intero grafo è stato immerso nel piano. Successivamente descriveremo le strutture dati necessarie per memorizzare gli spostamenti di ogni componente. Riportiamo di seguito l'intero algoritmo:

PLANARITY($G = (V, E)$)
algoritmo di planarità
1 Sia E il numero di archi di G
2 **se** $E > 3V - 6$ **allora** G non è planare
3 dividi G in componenti biconnesse
4 **per** ogni componente biconnessa C di G
5 visita C, numera i vertici e trasforma C in un *palm tree* P
6 trova un ciclo c in P
7 costruisci una rappresentazione planare per c
8 **per** ogni componente che si è formata quando c è stato eliminato
9 applica l'algoritmo in modo ricorsivo per determinare se questa componente più il ciclo c è planare

10 **se** questa componente più il ciclo è planare e la componente può essere aggiunta alla rappresentazione planare
11 aggiungila
12 **altrimenti** G non è planare
13 **end**
14 **end**
15 **end**

1.3.2 Costruzione dei cammini

Supponiamo che G sia un grafo biconnesso che è stato esplorato attraverso la visita in profondità per numerare i vertici e generare un *palm tree* P. D'ora in poi identificheremo i vertici con il numero che è stato loro assegnato tramite la procedura DFS.

Se v è un vertice, sia $S_v = \{w | \exists u(v \rightsquigarrow u \text{ e } u^- \rightarrow w)\}$. S_v è l'insieme dei vertici raggiungibili dagli archi di ritorno che escono dai discendenti di v. Per ogni vertice v di G definiamo le seguenti quantità che chiamaremo *low point* di v:

LOWPT$_1(v)$= min $(\{v\} \cup S_v)$ e

LOWPT$_2(v)$= min $(\{v\} \cup (S_v - \{\text{LOWPT}_1(v)\}))$

LOWPT$_1(v)$ e LOWPT$_2(v)$ sono rispettivamente il primo e il secondo vertice in ordine di numerazione raggiungibile da v attraverso un arco di ritorno che esce da un discendente di v. Per convenzione questi due valori sono uguali a v se non sono definiti e si ha LOWPT$_1(v) \neq$ LOWPT$_2(v)$ eccetto il caso in cui LOWPT$_1(v)$=LOWPT$_2(v)$= v. I *low point* di un vertice v dipendono solo dai *low point* dei figli di v e dagli archi di ritorno che partono da v; per questo motivo è facile calcolare i *low point* usando la visita in profondità. Riportiamo di seguito l'algoritmo VISITA per la visita in profondità di un grafo, opportunamente integrato per il calcolo dei *low point*:

VISITA(v, u, m)
il vertice u è il padre di v nell'albero ricoprente che stiamo costruendo

```
1   m := m + 1
2   NUMBER(v):= m
3   LOWPT1(v):= NUMBER(v)
4   LOWPT2(v):= NUMBER(v)
5   for each w ∈ A(v)
6     if NUMBER(w)= 0
7       w è un nuovo vertice
8       marca (v, w) come arco dell'albero
9       DFS(w, v, m)
10        if LOWPT1(w)< LOWPT1(v)
11          LOWPT2(v):= min {LOWPT1(v),LOWPT2(w)}
12          LOWPT1(v):= LOWPT1(w)
13        else if LOWPT1(w)= LOWPT1(v)
14          LOWPT2(v):= min{LOWPT2(v),LOWPT2(w)}
15        else LOWPT2(v):= min{LOWPT2(v),LOWPT1(w)}
16        end if
17      else if NUMBER(w)<NUMBER(v) and w ≠ u
18        marca (v, w) come arco di ritorno
19        if NUMBER(w)<LOWPT1(v)
20          LOWPT2(v):= LOWPT1(v)
21          LOWPT1(v):= NUMBER(w)
22        else if NUMBER(w)> LOWPT1(v)
23          LOWPT2(v):= min{LOWPT2(v),NUMBER(w)}
24        end if
25      end if
26    end for
```

È facile verificare che la visita in profondità così modificata calcola correttamente i *low point* con una complessità $O(|V| + |E|)$ [12, 24, 25]. Inoltre LOWPT_1 può essere usato per verificare se un grafo G è biconnesso come viene descritto in [13, 24].

Lemma 1.6. *Se G è un grafo biconnesso e $v \to w$, allora* $\text{LOWPT}_1(w) < v$ *eccetto il caso in cui $v = 1$ quando si ha* $\text{LOWPT}_1(w) = v = 1$*; inoltre* $\text{LOWPT}_1(1) = 1$.

La dimostrazione di questo Lemma che omettiamo per brevità è riportata in [24].

Figura 1.8: (a) Un grafo G; (b) i valori LOWPT_1 e LOWPT_2 calcolati per ogni vertice v, $\varphi(v,w)$ calcolata per ogni (v,w) e le liste di adiacenza ordinate secondo la funzione φ.

Per generare i cammini ordiniamo le liste di adiacenza del *palm tree* P calcolato su G eseguendo una visita in profondità secondo i *low point* ed eseguiamo un'altra visita in profondità. Sia φ una funzione definita su gli archi (v,w) di P come segue:

$$\varphi(v,w) = \begin{cases} 2w, & \text{se } v^- \rightarrow w; \\ 2\text{LOWPT}_1(w), & \text{se } v \rightarrow w \text{ e } \text{LOWPT}_2(w) \geq v; \\ 2\text{LOWPT}_1(w)+1, & \text{se } v \rightarrow w \text{ e } \text{LOWPT}_2(w) < v. \end{cases}$$

Calcoliamo $\varphi(v,w)$ per ogni arco in P e ordiniamo le liste di adiacenza secondo valori crescenti di φ, usando un *radix sort* otteniamo una complessità di $O(|V|+|E|)$. Riportiamo di seguito l'algoritmo:

```
ORDINA(G = (V, E))
costruzione delle liste di adiacenza ordinate
1   for i := 1 to 2V + 1
2     BUCKET(i) := lista vuota
3   end for
4   for each (v, w) ∈ E
5     calcola φ(v, w)
6     aggiungi (v, w) a BUCKET(φ(v, w))
7   end for
8   for v := 1 to V
9     A(v) := lista vuota
10  end for
11  for i := 1 to 2V + 1
12    for (v, w) ∈ BUCKET(i)
13      aggiungi w alla fine di A(v)
14    end for
15  end for
```

Questo algoritmo ordina le liste di adiacenza di P secondo valori crescenti di φ. La figura 1.8 mostra un grafo G, per ogni vertice v i *low point* e $\varphi(v, w)$ calcolata per ogni (v, w).

A questo punto generiamo i cammini eseguendo una visita in profondità del *palm tree* P: ogni volta che attraverseremo un arco dell'albero lo aggiungeremo al cammino che stiamo costruendo e ogni volta che attraverseremo un arco di ritorno questo diventerà l'ultimo arco del nostro cammino, l'arco successivo sarà il primo di un nuovo cammino. Per questo motivo ogni cammino consiste in una sequenza di *archi dell'albero* seguiti da un *arco di ritorno*.

Riportiamo di seguito l'algoritmo per generare cammini in un *palm tree* biconnesso con le liste di adiacenza ordinate secodo la funzione φ:

```
PATH((G = (V, E))
1   s := 0
2   PATHFINDER(1)

PATHFINDER(v)
1   for each w ∈ A(v)
2     if v → w
3       if s = 0
```

```
4           s := v
5           inizia un nuovo cammino
6         end if
7         aggiungi (u, v) al cammino corrente
8         PATHFINDER(v)
9       else v⁻ → w
10        if s = 0
11          s := v
12          inizia un nuovo cammino
13        end if
14        aggiungi (v, w) al cammino corrente
15        restituisci il cammino corrente
16        s := 0
17      end if
18    end for
```

Questo algoritmo ha una complessità di $O(|V| + |E|)$ perché è una visita in profondità con alcune operazioni aggiuntive per costruire i cammini. In seguito vedremo che le uniche informazioni necessarie per l'algoritmo di planarità sono il vertice iniziale e finale di ogni cammino.

Ora enunceremo alcuni Lemmi tecnici che riassumono le proprietà che caratterizzano i cammini che vengono generati dalla procedura PATHFINDER. Questi risultati ci serviranno per dimostrare la correttezza dell'algoritmo di verifica della planarità.

Lemma 1.7. *Sia $p : s \rightsquigarrow f$ un cammino generato dall'algoritmo* PATHFINDER. *Se consideriamo gli archi di ritorno che non sono stati usati per costruire gli altri cammini quando il primo arco in p è stato attraversato, allora f è il vertice a numerazione più piccola raggiungibile da ogni discendente di s tramite un arco di ritorno. Se $v \neq s$, $v \neq f$ e v giace su p allora f è il vertice a numerazione più piccola raggiungibile da ogni discendente di v attraverso un arco di ritorno (tutti gli archi di ritorno uscenti da discendenti di v non sono ancora stati usati quando il primo arco in p viene attraversato).*

Dim. Questo Lemma è una immediata conseguenza di come abbiamo ordi-

Figura 1.9: (a) Cammini generati dalla procedura PATHFINDER dal grafo in figura 1.6. A: (1,2,3,4,1); B: (4,5,1); C: (5,3); D: (4,2); (b) segmenti rispetto al ciclo iniziale.

nato le liste di adiacenza. □

Lemma 1.8. *Sia $p : s \rightsquigarrow f$ un cammino generato dall'algoritmo* PATHFINDER. *Allora $f \rightsquigarrow s$ appartiene all'albero ricoprente P. Se p è il primo cammino allora p è un ciclo, altrimenti p è un cammino semplice. Se p non è il primo cammino, allora ha esattamente due vertici (f ed s) in comune con i cammini generati precedentemente.*

Dim. Sia $p : s \rightsquigarrow f$ un cammino generato dall'algoritmo PATHFINDER. Se il cammino consiste solo in un arco di ritorno allora il cammino è semplice e $f \rightsquigarrow s$. Se il cammino contiene almeno un arco dell'albero sia $s \rightarrow v$ il primo di questi archi. Allora $f = \text{LOWPT}_1(v)$ per il Lemma 1.7. Se $s = 1$

il cammino è un ciclo, se $s > 1$ allora il cammino è semplice per il Lemma 1.6. In entrambi i casi si ha che $f \rightsquigarrow s$. Se f è stato raggiunto durante la procedura di ricerca dei cammini allora s era già stato raggiunto, quindi si ha che ogni cammino ad eccezione del primo ha esattamante due vertici in comune, f ed s, con i cammini generati precedentemente. □

Lemma 1.9. *Siano $p_1 : s_1 \rightsquigarrow f_1$ e $p_2 : s_2 \rightsquigarrow f_2$ due cammini generati dall'algoritmo* PATHFINDER. *Se p_1 è generato prima di p_2 ed s_1 è un predecessore di s_2 allora si ha $f_1 \leq f_2$.*

Dim. L'arco di ritorno che termina il cammino p_2 esce da un discendente di s_1 e non è stato ancora usato quando viene generato p_1. Allora per il Lemma 1.7 si ha che $f_1 \leq f_2$. □

Lemma 1.10. *Siano $p_1 : s \rightsquigarrow f$ e $p_2 : s \rightsquigarrow f$ due cammini generati dall'algoritmo* PATHFINDER *con lo stesso vertice iniziale e finale. Sia v_1 il secondo vertice di p_1 e sia v_2 il secondo vertice di p_2. Supponiamo che p_1 sia generato prima di p_2, $v_1 \neq f$ e* $\text{LOWPT}_2(v_1) < s$*; allora $v_2 \neq f$ e* $\text{LOWPT}_2(v_2) < s$.

Dim. Il vertice v_1 compare prima del vertice v_2 nella lista di adiacenza di $A(s)$ perché p_1 viene generato prima di p_2. Questo Lemma segue dall'ordine che abbiamo imposto alla lista $A(s)$. □

Il LOWPT_2 è stato inserito nell'algoritmo di ricerca dei cammini a causa di questo Lemma, la cui importanza sarà evidente quando considereremo l'immersione dei cammini nel piano.

Abbiamo bisogno di un'altra definizione prima di considerare l'immersione dei cammini. Se $p : s \rightsquigarrow f$ è un cammino semplice generato dalla procedura PATHFINDER sia $p_0 : s_0 \rightsquigarrow f_0$ il primo cammino generato contenente il vertice s. Se $f_0 < f$ allora p viene chiamato *cammino normale*. Se $f_0 = f$ allora p viene chiamato *cammino speciale*. Il caso $f_0 > f$ non si presenta a causa del Lemma 1.9.

1.3.3 Immersione di cammini nel piano

In questo paragrafo enunceremo due Lemmi che sono fondamentali per dimostrare la correttezza dell'algoritmo di verifica della planarità. Sia G un grafo biconnesso e sia C un insieme di cammini generati dall'algorimo PATHFINDER, verifichiamo la planarità di G tentando di immergere un cammino per volta nel piano. Sia c il primo cammino che, come abbiamo visto, per il Lemma 1.8 è un ciclo; c consiste in un insieme di archi dell'albero $1 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_n$ seguiti da un arco di ritorno $v_n^- \rightarrow 1$. La numerazione dei vertici è tale che $1 < v_1 < ... < v_n$. Quando c viene rimosso G viene suddiviso in diverse componenti connesse, chiamate *segmenti*. Ogni segmento s consiste o in un solo arco di ritorno (v_i, w) oppure in un arco dell'albero (v_i, w) più un sottoalbero con radice w più tutti gli archi di ritorno che partono da questo sottoalbero.

L'ordine in cui i cammini vengono generati è tale che tutti i cammini in un segmento vengano generati prima dei cammini in ogni altro segmento e i segmenti vengano esplorati in ordine decrescente rispetto a v_i. Un segmento deve essere immerso interamente in un lato di c per il Teorema della Curva di Jordan. Un segmento è collegato a c da un arco (v_i, w) che parte da c e da uno o più archi di ritorno uscenti da c. Diremo che il segmento S è immerso a *sinistra* (di c) se l'ordine in cui gli archi vengono disegnati attorno a v_i è $(v_{i-1}, v_i), (v_i, w), (v_i, v_{i+1})$ (figura 1.10(a)). Diremo che il segmento S è immerso a *destra* (di c) se l'ordine in cui gli archi vengono disegnati attorno a v_i è $(v_{i-1}, v_i), (v_i, v_{i+1}), (v_i, w)$ (figura 1.10(b)). Diremo che un arco di ritorno che entra in c è immerso a *sinistra* (*destra*) se il segmento cui appartiene è alla sinistra (destra) di c. Se (x, v_i) è un arco di ritorno che entra in c a sinistra l'ordine in cui gli archi vengono disegnati attorno a v_j è $(v_{j-1}, v_j), (x, v_j), (v_j, v_{j+1})$ per il Lemma 1.3.

Supponiamo che c ed i segmenti esplorati prima di S siano stati immersi nel

Figura 1.10: (a) Il segmento (4,2) è immerso a sinistra di C:(1,2,3,4,1), infatti l'ordine in cui gli archi vengono disegnati attorno a $v_i = 4$ è (3,4) (4,2) (4,5); (b) il segmento (5,3) è immerso a destra di C:(1,2,3,4,1), infatti l'ordine in cui gli archi vengono disegnati attorno a $v_i = 3$ è (2,3) (3,4) (3,5).

piano. Sia $p : v_i \rightsquigarrow v_j$ il primo cammino trovato in S. Il seguente Lemma ci dà una condizione necessaria e sufficiente per immergere p.

Lemma 1.11. *Il cammino $p : v_i \rightsquigarrow v_j$ può essere aggiunto all'immersione planare alla sinistra (destra) di c se e solo se nessun arco di ritorno (x, v_k) immerso precedentemente sulla sinistra (destra) di c soddisfa $v_j < v_k < v_i$.*

Dim. Se nessun arco di ritorno soddisfa questa condizione allora vuol dire che nessuno degli archi che sono stati già immersi entra o esce da c sulla sinistra (destra) tra v_j e v_i. Allora il cammino p può essere immerso a sinistra (destra) di c. Viceversa supponiamo supponiamo di voler immergere p a sinistra, ma

alcuni archi di ritorno (x, v_k), già immersi a sinistra $v_j < v_k < v_i$, entrano in c da sinistra. Si ha che x appartiene a c (sia $x = v_l$) oppure (x, v_k) è una parte del segmento S' che ha come primo arco (v_l, w). Sappiamo che $v_l \geq v_i$ per l'ordine in cui vengono generati i cammini.

Figura 1.11: Illustrazione del Lemma 1.11, Caso 1

Dobbiamo considerare due casi:

1. $v_l > v_i$ Supponiamo che p sia stato immerso a sinistra. Se consideriamo p , un cammino in S' che collega v_l e v_k e il cammino costituito da archi dell'albero da v_j a v_l possiamo costruire tre cammini da v_i a v_k che contraddicono il Lemma 1.3 (figura 1.11). Per questo motivo p non può essere immerso a sinistra.

2. $v_l = v_i$. Sia $p_1 : v_l \rightsquigarrow v_m$ il primo cammino trovato nel segmento S'. Si ha che $v_m \leq v_j$ per il Lemma 1.9. Ci sono due sottocasi:

Figura 1.12: Illustrazione del Lemma 1.11, Caso 2 (a)

(a) $v_m < v_j$ Supponiamo che p sia stato immerso a sinistra. Se consideriamo p , un cammino in S' che collega v_k e v_m e il cammino costituito da archi dell'albero da v_m a v_i possiamo costruire tre cammini da v_k a v_j che contraddicono il Lemma 1.3 (figura 1.12). Per questo motivo p non può essere immerso a sinistra.

(b) $v_m = v_j$. Siano y e w rispettivamente il secondo vertice del cammino p e del cammino p_1. Dal momento che il segmento S' contiene l'arco di ritorno (x, v_k) si ha che $w \neq v_m$ e $\text{LOWPT}_2(w) < v_i$. Confrontando p e p_1 e applicando il Lemma 1.10 si ha che $y \neq v_j$ e $\text{LOWPT}_2(y) < v_i$. Inoltre $\text{LOWPT}_2(y) < v_j$ dal momento che $\text{LOWPT}_1(y) = v_j$ per il Lemma 1.7. Supponiamo che p sia stato immerso a sinistra. Considerando p, p_1, un cammino da un vertice appartenente a p a $\text{LOWPT}_2(y)$, un cammino da un vertice appartenente a p_1 a v_k e un cammino (eventualmente vuoto)

Figura 1.13: Illustrazione del Lemma 1.11, Caso 2 (b)

costuituito da archi dell'albero che collega v_k e $\text{LOWPT}_2(y)$, possiamo costruire tre cammini che contraddicono il Lemma 1.3 (figura 1.13). Per questo motivo p non può essere immerso a sinistra. □

Usiamo il Lemma 1.11 per verificare la planarità nel modo seguente: per prima cosa immergiamo il ciclo c nel piano, poi i segmenti uno alla volta nell'ordine in cui sono stati esplorati durante la procedura di ricerca dei cammini. Per immergere un segmento S troviamo un cammino al suo interno che chiameremo p; scegliamo un lato, ad esempio a sinistra, su cui immergere p e confrontiamo p con gli archi di ritorno immersi precedentemente per determinare se p può essere immerso (figura 1.14). Se questo non è possibile, spostiamo i segmenti che hanno archi di ritorno che condizionano la posizione di p da sinistra a destra. Se p può essere immerso dopo lo spostamento di questi segmenti allora può essere immerso. Non bisogna dimenticare che

Figura 1.14: Illustrazione del Lemma 1.11: (4,2) non può essere immerso a destra perché $v_i = 4$ $v_j = 2$ e $(x, v_k) = (5, 3)$ e si ha $v_j < v_k < v_i$ $2 < 3 < 4$.

nel momento in cui spostiamo questi segmenti da sinistra a destra possiamo essere costretti a spostare altri segmenti, per questo motivo potrebbe essere impossibile immergere p. Se si presenta questa situazione il grafo non è planare. Se p può essere immerso proviamo ad immergere il resto di S usando l'algoritmo in modo ricorsivo. Quindi proveremo ad immergere il segmento successivo.

Per implementare in modo efficiente questo metodo abbiamo bisogno di una buona struttura dati . Se stiamo per immergere un segmento che inizia dal vertice v_i dobbiamo sapere quali vertici del cammino che va da 1 a v_i hanno archi di ritorno entranti da destra e da sinistra. Per questo motivo useremo due pile: L e R. La pila L conterrà in ordine i vertici v_k tali che $1 \rightsquigarrow v_k \rightsquigarrow v_i$,

$1 < v_k < v_i$ e qualche arco di ritorno già immerso entra in v_k sulla sinistra. Per ogni segmento che ha un arco di ritorno che esce da v_k dobbiamo inserire il vertice v_k nella pila L, qualche volta due archi di ritorno che appartengono allo stesso segmento entrano nello stesso vertice v_k, per questo motivo v_k potrebbe apparire più di una volta nella pila sebbene esso rappresenti un singolo segmento. La pila R ha la stessa funzione di L, ma sul lato destro.

Le pile L ed R devono essere aggiornate in quattro modi:

1. Dopo che tutti i segmenti uscenti da v_{i+1} sono stati esplorati ed immersi, tutte le occorrenze di v_i su L ed R devono essere cancellate per il fatto che i segmenti che sono già stati esplorati partono da vertici non più grandi di v_i. Questo aggiornamento richiede lo spostamento di alcuni elementi in cima ad L ed R.

2. Se $p : s \rightsquigarrow f$ è il primo cammino in un segmento S, e se p è normale allora f deve essere aggiunto alla pila quando p viene immerso. Dal momento che s giace su p, p è normale se e solo se $f > 1$. Per il Lemma 1.11 p può essere immerso a sinistra (destra) quando nessun vertice di L (R) è più grande di f, così f può essere aggiunto in cima ad L (R).

3. L'applicazione ricorsiva dell'algoritmo deve aggiungere gli elementi corrispondenti agli altri cammini nel segmento s.

4. Gli elementi devono essre spostati da una pila all'altra come i rispettivi segmenti. Ad esempio l'immersione di un arco di ritorno sulla sinistra costringe gli archi di ritorno dello stesso segmento ad essere immersi a sinistra per il Lemma 1.3 e potrebbe costringere gli archi di ritorno degli altri segmenti ad essre immersi a destra per il Lemma 1.11.

Definiamo un *blocco* B un insieme massimale di elementi di L ed R a cui corrispondono archi di ritorno tali che la posizione di qualcuno di questi

determina quella di tutti gli altri. Gli elementi dei blocchi cambiano come quelli delle pile, ma i blocchi partizionano sempre gli elementi di una pila.

Inoltre i blocchi hanno una struttura semplice come viene illustrato nel Lemma seguente:

Lemma 1.12. *Sia B un blocco. Allora gli elementi che appartengono a $B \cap L$ ($B \cap R$) sono adiacenti su $L(R)$. Si ha anche che ci sono vertici v_j, v_k su c tali che per $v_l \in L \cup R$:*

1. *se $v_j < v_l < v_k$ allora $v_l \in B$,*
2. *se $v_l < v_j$ oppure $v_l > v_k$ allora $v_l \notin B$.*

La dimostrazione di questo Lemma che omettiamo per brevità è riportata in [11]. Questo Lemma ci mostra come tenere traccia dei blocchi. I blocchi ci danno abbastanza informazioni per spostare con facilità gli elementi da una pila all'altra. Usiamo una lista per memorizzare L ed R. Allora per spostare un blocco di dati da una pila all'altra dobbiamo solamente spostare il puntatore all'inizio e alla fine del blocco. Useremo una pila B per memorizzare le informazioni riguardanti i blocchi. Ogni elemento di B rappresenta un blocco ed è una coppia ordinata (x, y) dove x è il puntatore al primo elemento del blocco su L ed y punta all'ultimo elemento del blocco su R.

Ecco l'algoritmo per immergere nel piano un grafo biconnesso con le liste di adiacenza opportunamente ordinate:

EMBED($G = (V, E)$)
1 $L := R := B :=$ pila vuota
2 trova il primo ciclo c
3 **while** alcuni segmenti non sono stati visitati
4 inizia a cercare un cammino nel segmento successivo S
5 quando torni indietro attraverso l'arco $v \rightarrow w$ cancella gli elementi di L, R e del blocco B che contengono vertici con una numerazione maggiore di v
6 sia $p : s \rightsquigarrow f$ il primo cammino trovato nel segmento S

7 **while** la posizione degli elementi in cima al blocco B determina la posizione di p
8 cancella gli elementi in cima al blocco
9 **if** gli elementi del blocco sono stati immersi a sinistra **then** sposta blocchi di elementi da L ad R e da R a L
10 **if** il blocco ha ancora elementi a sinistra in conflitto con p **then** il grafo non è planare
11 **end while**
12 **if** p è normale **then** aggiungi l'ultimo vertice di p a L
13 aggiungi un nuovo blocco a B che corrisponde a p e ai blocchi che sono stati appena cancellati da B
14 applica l'algoritmo ricorsivamente per immergere gli altri cammini in S
15 *dopo discuteremo i dettagli di questa applicazione ricorsiva. Dopo avere eseguito il passo d tutti i cammini in S che partono da un predecessore di S saranno stati rappresentati su L. Ci sarà anche un nuovo blocco in B che corrisponde a questi cammini*
16 combina la cima dei due blocchi di B
17 **end while**

Lemma 1.13. *L'algoritmo* EMBED *gira fino alla fine se e solo se G è planare altrimenti si interrompe e dichiara il grafo non planare.*

Dim. EMBED è una implementazione diretta dell'algoritmo che abbiamo descritto prima. Ad ogni passo le pile L ed R contengono elementi relativi agli archi di ritorno immersi a sinistra e a destra del ciclo c e la pila B contiene informazioni riguardo la fine di ciascun blocco di dati. I Lemmi 1.11 e 1.12 vengono usati rispettivamente per verificare la planarità e per modificare i blocchi. Supponendo che il passo 14 (l'applicazione ricorsiva dell'algoritmo) sia implementato correttamente, allora è semplice provare per induzione che:

1. l'immersione di qualunque arco di ritorno in un blocco determina completamente l'immersione di tutti gli archi di ritorno appartenenti allo stesso blocco,

2. l'immersione di un arco di ritorno che appartiene ad un blocco non influisce sull'immersione di un arco di ritorno che non appartiene allo stesso blocco.

Per i Lemmi 1.11 e 1.12 l'algorimto EMBED verifica correttamente la planarità. □

Consideriamo l'immersione del secondo cammino e di quelli successivi in un segmento. Supponiamo di aver già immerso il ciclo c, tutti i segmenti prima di S e il primo cammino $p : s \rightsquigarrow f$ in S. È facile vedere che il resto di S può essere aggiunto all'immersione se e solo se S e c insieme danno luogo ad un grafo planare (segue dai risultati di [4]). Il cammino p e il cammino costituito da archi dell'albero che va da f ad s dà luogo ad un ciclo c' usato per l'applicazione ricorsiva dell'algoritmo di immersione. Dopo che p è stato immerso sulla sinistra dall'algoritmo EMBED l'elemento in cima ad L è f. Tutti gli archi di ritorno in S portano a vertici con una numerazione non più piccola di f per il Lemma 1.7. Supponiamo di porre un segnale di fine pila in cima ad R ed applichiamo l'algoritmo ricorsivamente per determinare se il ciclo c' più i segmenti in S che si sono formati quando c' è stato eliminato possono esser immersi nel piano. Se la ricorsione si conclude con successo, le pile L ed R conterranno elementi che corrispondono agli archi di ritorno che finiscono cammini normali in S e la pila B potrebbe contenere alcuni nuovi blocchi. Il resto del ciclo c può essere aggiunto all'immersione di S e c' se e solo se nessun nuovo blocco ha elementi che appartengono sia ad L che ad R. In questo caso ogni nuovo blocco con un elemento appartenente ad R può essere spostato nella pila L in modo tale che nessun nuovo blocco avrà elementi appartenenti ad R.

Per concludere la verifica della planarità di c ed S dobbiamo provare a spostare i nuovi blocchi da L ad R. Dobbiamo combinare tutti i nuovi blocchi in un unico blocco che corrisponde ai cammini in S meno p e dobbiamo cancellare il segnale di fine pila su R. Dopo queste operazioni R sarà ripristinato, gli elementi in cima ad L saranno quelli corrispondenti agli archi di ritorno in S, B conterrà un nuovo blocco che corrisponde agli archi di ritorno in S meno p. Il passo 14 può essere implementato in questo modo:

14: *applica questo algoritmo ricorsivamente per immergere gli altri cammini in S aggiungi un segnale di fine pila ad R*
1 **for each** nuovo blocco $(x, y) \in B$
2 **if** $(x \neq 0)$ **and** $(y \neq 0)$ **then** il grafo non è planare
3 **if** $(y \neq 0)$ **then** sposta gli elementi del blocco in L
4 cancella (x, y) da B
5 **end for**
6 cancella il segnale di fine pila su R
7 aggiungi un blocco a B per rappresentare S meno il cammino p

Lemma 1.14. *Se il passo 14 viene implementato come descritto sopra allora l'algoritmo che verifica la planarità di un grafo è corretto.*

Dim. Questo Lemma segue dal Lemma 1.13 e dal fatto che S meno p può essere aggiunto all'immersione planare se e solo se S più c è planare [4]. Gli elementi che già appartengono ad L, R e B non possono influire sull'applicazione ricorsiva dell'algoritmo, dal momento che R ha un segnale di fine pila e tutti gli elementi di L non sono più grandi di f. Alla fine dell'applicazione ricorsiva dell'algoritmo le informazioni contenute dalle pile L, R e B sono proprio quelle di cui abbiamo bisogno. La figura 1.15 mostra l'applicazione ricorsiva dell'algoritmo di immersione sul grafo in figura 1.6 e i contenuti delle pile L, R e B durante l'esecuzione dell'algoritmo sullo stesso grafo. □

1.3.4 L'algoritmo di verifica della planarità

Dal momento che i cammini vengono immersi quando vengono trovati, l'algoritmo di immersione deve essere combinato con quello di ricerca dei cammini.

Riportiamo di seguito l'implementazione dell'algoritmo che determina se un grafo biconnesso G può essere immerso nel piano. G è rappresentato da un insieme di liste di adiacenza $A(v)$ opportunamente ordinate. Le pile L ed R sono memorizzate come liste utilizzando gli array STACK e NEXT. STACK(i)

Figura 1.15: Applicazione ricorsiva dell'algoritmo di immersione sul grafo in figura 1.6: contenuti delle pile L, R e B durante l'esecuzione della procedura EMBED.

restituisce un elemento della pila e NEXT(i) punta all'elemento successivo della stessa pila. NEXT(0) punta al primo elemento di L. NEXT(-1) punta al primo elemento di R. FREE è l'indice della prima posizione libera di STACK. La variabile p indica il numero del cammino corrente. Se v è un vertice PATH(v) è il numero del primo cammino che contiene v. Se i è il numero di un cammino $f(i)$ indica l'ultimo vertice del cammino. I blocchi sono rappresentati come coppie ordinate di una pila B. Se (x, y) è un elemento di B, allora x indica l'ultimo elemento di L nel blocco e y l'ultimo elemento di R nel blocco. Se $x = 0$ ($y = 0$) il blocco non ha elementi appartenenti ad L (R).

EMBED($G = (V, E)$)
1 *inizializzo i valori*

```
2   NEXT(−1):= 0
3   NEXT(0):= 0
4   FREE:= 1
5   STACK(0):= 0
6   B := pila vuota
7   p := 0
8   s := 0
9   PATH(1):= 1
10   la ricerca inizia dal vertice 1
11   PATHFINDER(1)

PATHFINDER(v)
1   for each w ∈ A(v)
2     if v → w
3       if s = 0
4         s := v
5         p := p + 1
6       end if
7       PATH(w):= p
8       PATHFINDER(w)
9       cancella gli elementi della pila e dei blocchi che corrispondono a vertici a
        numerazione non più bassa di v
10        while (x, y) ∈ B ha (STACK(x)≥ v) or (x = 0) and (STACK(y)≥ v)
        or (y = 0)
11          cancella (x, y) da B
12        if (x, y) ∈ B ha (STACK(x)≥ v) then sostituisci (x, y) con (0, y)
13        if (x, y) ∈ B ha (STACK(y)≥ v) then sostituisci (x, y) con (x, 0)
14        while NEXT(-1)≠ 0 and STACK(NEXT(-1)) ≥ v
15          NEXT(-1):= NEXT(NEXT(-1))
16        while NEXT(0)≠ 0 and STACK(NEXT(0))≥ v
17          NEXT(0):= NEXT(NEXT(0))
18        if PATH(w)≠ PATH(v)
19        tutti i segmenti aventi come primo arco (v, w) sono stati immersi. Nuovi
        blocchi devono essere spostati da destra a sinistra
20          L′ := 0
21          while (x, y) ∈ B ha (STACK(x)> f(PATH(w)))
          or (STACK(y)> f(PATH(w))) and (STACK(NEXT(-1))≠ 0)
22            if (STACK(x)> f(PATH(w))
23              if (STACK(y)> f(PATH(w)) then il grafo non è planare
24              L′ := x
25            else
26              SAVE:= NEXT(L′)
27              NEXT(L′):= NEXT(-1)
28              NEXT(-1):= NEXT(y)
```

```
29              NEXT(y):= SAVE
30              L' := y
31            end if
32            cancella (x, y) da B
33          end while
34          il blocco B deve esser combinato con i nuovi blocchi appena cancellati
35          cancella (x, y) da B
36          if x ≠ 0 then aggiungi (x, y) da B
37          else if (L' ≠ 0) or (y ≠ 0) then aggiungi (L', y) a B
38          cancella il simbolo di fine pila sulla pila destra
39          NEXT(-1):= NEXT(NEXT(-1))
40        end if
41      else v⁻ → w. Il cammino corrente è completo. Il cammino è normale se
        f(PATH(s)) < w
42        if s = 0
43          p := p + 1
44          s := v
45        end if
46        f(p) = w
47        sposta blocchi di elementi dalla pila sinistra a quella destra per immergere
          p
48        L' = 0
49        R' = −1
50        while (NEXT(L')≠ 0) and (STACK(NEXT(L'))> w) or
          (NEXT(R')≠ 0) and (STACK(NEXT(R'))> w)
51          if (x, y) ∈ B ha (x ≠ 0) and (y ≠ 0)
52            if (STACK(NEXT(L'))> w)
53              if (STACK(NEXT(R'))> w) then il grafo non è planare
54              SAVE:= NEXT(R')
55              NEXT(R'):= NEXT(L')
56              NEXT(L'):= SAVE
57              SAVE:= NEXT(x)
58              NEXT(x):= NEXT(y)
59              NEXT(y):= SAVE
60              L' := y
61              R' := x
62            else (STACK(NEXT(R'))> w)
63              L' := x
64              R' := y
65            end if
66          else if (x, y) ∈ B ha (x ≠ 0) (STACK(NEXT(L'))> w)
67            SAVE:= NEXT(x)
68            NEXT(x):= NEXT(R')
```

```
69            NEXT(R'):= NEXT(L')
70            NEXT(L'):= SAVE
71            R' := x
72          end if
73          else if (x, y) ∈ B ha (y ≠ 0) then R' := y
74          end if
75        cancella (x, y) da B
76        end while
77        aggiungi w alla pila sinistra se p è normale
78        if f(PATH(s))< w
79          if L' = 0 then L' = FREE
80          STACK(FREE):= w
81          NEXT(FREE):= NEXT(0)
82          NEXT(0):= FREE
83          FREE:= FREE+1
84        end if
85        aggiungi un nuovo blocco che corrisponde "ai vecchi blocchi". Il nuovo
        blocco sarà vuoto se il segmento contenente il cammino corrente non è
        un arco di ritorno
86        if R' = −1 then R' := 0
87        if (L' ≠ 0) or (R' ≠ 0) or (v ≠ s) then aggiungi (L', R') a B
88        se il segmento contenente il cammino corrente non è un solo arco di
        ritorno, aggiungi un simbolo di fine pila alla pila sinistra
89        if v ≠ s
90          STACK(FREE):= 0
91          NEXT(FREE):= NEXT(-1)
92          NEXT(-1):= FREE
93          FREE:= FREE+1
94        end if
95        s := 0
96      end if
97    end for
```

Lemma 1.15. EMBED *verifica correttamente la planarità di un grafo* G.

Dim. EMBED è una diretta implementazione dell'algoritmo descritto nel paragrafo 1.3.3. □

Lemma 1.16. EMBED *ha una complessità di* $O(|V| + |E|)$ *dove* $|V|$ *è il numero di vertici ed* $|E|$ *il numero di archi di* G.

Dim. L'algoritmo di ricerca dei cammini ha una complessità di $O(|V|+|E|)$ come mostrato nel paragrafo 1.3.2 perché è una visita in profondità con alcune operazioni aggiuntive per costruire i cammini. Le uniche informazioni riguardo ai cammini trovati che vengono usate dall'algoritmo di immersione sono i vertici finali. La parte che riguarda l'immersione nell'algoritmo consiste in una sequenza di operazioni su una pila. Aggiungere o cancellare un elemento da una pila è una operazione che richiede tempo costante. Il numero totale di elementi che vengono memorizzati in L, R o B è $O(|V|+|E|)$ perché il numero totale di cammini è $|E|-|V|+1$. Quindi le operazioni che riguardano le pile hanno una complessità $O(|V|+|E|)$. L'inizializzazione richiede un tempo costante quindi la complessità dell'intero algoritmo è $O(|V|+|E|)$. □

Lemma 1.17. *L'algoritmo di planarità ha una complessità di $O(|V|)$, dove $|V|$ è il numero di vertici di un grafo G.*

Dim. L'algoritmo termina se il numero degli archi di G è superiore a $3|V|-6$, l'operazione di contare gli archi ha una complessità $O(|V|)$. Se G ha $O(|V|)$ archi allora la prima visita in profondità ha una complessità di $O(|V|)$, l'ordinamento degli archi utilizzando i *low point* richiede un tempo $O(|V|)$ e anche l'algoritmo di immersione. É facile vedere che anche lo spazio occupato dall'algoritmo è $O(|V|)$. □

Osserviamo che l'algoritmo di planarità che viene descritto in questo capitolo verifica se un grafo G è planare, ma non costruisce una rappresentazione planare per G. Ad ogni modo questo algoritmo colleziona abbastanza informazioni che permettono di costruire facilmente una rappresentazione planare per G.

Capitolo 2

Immersione di grafi planari su griglie bidimensionali

In questo capitolo considereremo il problema di costruire un disegno rettilineo sulla griglia Z^2 di un grafo planare G con n vertici. Le possibili soluzioni di questo problema dipendono dalle limitazioni che si vogliono porre al disegno.

2.1 Preliminari

Sia $G = (V, E)$ un grafo planare, costruire una *immersione rettilinea* di G vuol dire immergere G nel piano in modo tale che i vertici vengono mandati in punti della griglia e ogni arco viene mandato in una linea spezzata che consiste in una sequenza alternata di segmenti orizzontali e verticali. Dal momento che due archi differenti non possono avere in comune un punto interno, una condizione che deve essere soddisfatta da G è che ogni vertice abbia grado massimo 4. Un grafo planare connesso che verifica questa condizione viene chiamato *standard*. Tutti i grafi che prenderemo in considerazione d'ora in poi saranno grafi standard. Ogni punto interno di un arco, che è una intersezione di un segmento orizzontale con un segmento verticale, è chiamato

bend (*piega*). Un arco $k - bend$ è un arco con esattamente k bend.

Il numero di bend di un arco e l'area occupata dal disegno sono valori importanti per l'applicazione del disegno rettilineo sulla griglia Z^2. Ci sono due problemi di ottimizzazione relativi al numero di bend di una immersione rettilinea di un grafo: il problema *min-max* e quello *min-sum*. Il problema $min - max$ [1, 3] consiste nel trovare il più piccolo k tale che ogni grafo può essere immerso in modo rettilineo nella griglia con al massimo k bend per ogni arco. Un grafo che può essere immerso nella griglia con al massimo r bend per ogni arco viene chiamato $r - immergibile$ e l'immersione viene chiamata $r - immersione$ di G.

Teorema 2.1. *Ogni grafo standard è 3-immergibile.*

Figura 2.1: L'ottaedro.

Teorema 2.2. *Ogni grafo standard è 2-immergibile ad eccezione dell'ottaedro.*

Teorema 2.3. *Ogni grafo standard è 3-immergibile in modo tale che il numero totale di bend sia al massimo 2n.*

Le dimostrazioni di questi Teoremi che omettiamo per brevità sono riportate in [19]. Il problema *min-sum* [26] consiste nel trovare tra tutte le possibili immersioni rettilinee quella il cui numero totale di piegature sia minimo. Il problema di minimizzare l'area occupata dal disegno può essere definito in modo simile [22, 29]. La caratteristica più importante dell'algoritmo che viene presentato in questo capitolo è che l'immersione rettilinea che verrà costruita avrà al massimo due bend per ogni arco con l'eccezione dell'ottaedro, dove ci saranno al massimo due archi con tre bend. Nel paragrafo 2.2 descriveremo un algoritmo che ha una complessità lineare nello spazio e nel tempo che riceve in input un grafo standard biconnesso con le liste di adiacenza opportunamente ordinate e produce una immersione con al massimo tre bend per ogni arco di G. Il numero totale di bend è al massimo $2n + 4$ e l'area occupata è $O(n^2)$. Nel paragrafo 2.3 daremo una versione modificata dell'algoritmo presentato in precedenza che modifica gli archi con due bend di un certo tipo. Nel paragrafo 2.4 proveremo che per ogni grafo standard biconnesso, ad eccezione dell'ottaedro, esiste una immersione rettilinea tale che l'algoritmo presentato precedentemente produce un disegno con al massimo due bend per ogni arco. Nel paragrafo 2.5 estenderemo i risultati enunciati a tutti i grafi standard.

2.2 Algoritmo per grafi biconnessi che produce al massimo tre bend per ogni arco

Per prima cosa introduciamo i concetti di orientazione bipolare e di numerazione unilaterale che valgono per ogni grafo biconnesso G. Dare una *orientazione bipolare* a G vuol dire assegnare una direzione ad ogni arco di G in modo tale che il grafo risultante sia aciclico con una sola sorgente ed un solo pozzo.

Una *numerazione unilaterale* di G è una numerazione dei vertici da 1 a n in modo tale che ogni vertice $i = 2, ..., n-1$ è adiacente sia a qualche vertice $h < i$ sia a qualche vertice $j > i$. Un grafo con questa numerazione viene chiamato *unilateralmente numerato*.

Una numerazione dei vertici di G da 1 ad n concorda con una orientazione aciclica di G se ogni arco orientato esce da un vertice a numerazione più bassa ed entra in un vertice a numerazione più alta, Una numerazione che verifica questa proprietà viene chiamata numerazione st; una numerazione st esiste e può essere trovata in $O(n)$ grazie all'ordinamento topologico [24] .

Figura 2.2: (a) Un grafo planare biconnesso G; (b) una orientazione bipolare di G.

In questo paragrafo $G = (V, E)$ è un grafo planare biconnesso; supponiamo di conoscere l'immersione planare di G. Immergere un grafo planare vuol dire costruire le liste di adiacenza (*liste di rotazione*) in modo tale che per ogni vertice w tutti i vertici adiacenti a w compaiano secondo l'ordine in cui vengono disegnati. Una immersione di questo tipo può essere realizzata con

Figura 2.3: (a) Una numerazione st di G; (b) le liste di rotazione di G.

una complessità lineare in n usando uno degli algoritmi descritti in [5, 8, 11, 16, 17, 18].

Si dice che due immersioni di G sono *equivalenti* se tutte le loro liste di rotazione coincidono. D'ora in poi denoteremo l'immersione planare di G (ovvero le liste di adiacenza opportunamente ordinate per immergere G) con G. Sia w un vertice sulla faccia esterna di G. Allora ci saranno due archi sulla faccia esterna che hanno w in comune. Esattamente uno di questi due archi, e, ha la proprietà che l'altro arco, e', è il successivo nella lista di rotazione di w. Allora e verrà chiamato l'*arco più a destra* e e' l'*arco più a sinistra* di w. È noto che per ogni grafo biconnesso esiste una numerazione st e può essere trovata in un tempo lineare [7, 9, 23]. Inoltre la sorgente s ed il pozzo t possono essere scelti arbitrariamente. Ora, senza perdita di generalità, possiamo supporre che G abbia una orientazione bipolare in modo tale che la sorgente s ed il pozzo t siano stati scelti sulla faccia esterna di G e che i suoi vertici siano stati unilateralmente numerati. Per ogni vertice v sia

$d^-(v)$ il numero di archi entranti e $d^+(v)$ il numero di archi uscenti. Sotto le ipotesi appena enunciate valgono le seguenti proposizioni:

Proposizione 2.4. *In ogni immersione planare di G, per ogni vertice w tutti gli archi entranti (uscenti) sono consecutivi nella lista di rotazione di w.*

La dimostrazione di questa Proposizione che omettiamo per brevità è riportata in [27]. Per descrivere l'algoritmo in modo più semplice è conveniente inserire in ogni arco orientato (u, v) un punto interno w (*vertice virtuale*) in modo tale da dividere l'arco (u, v) in due *semi-archi* (u, w) e (w, v). Il vertice virtuale w sarà numerato con lo stesso numero del vertice v. Sia G' il grafo ottenuto da G in questo modo. È semplice ottenere una immersione con al massimo tre bend per arco di G da una immersione con al massimo tre bend per arco di G'. Data una numerazione unilaterale di G, per ogni $i = 1, ..., n$ sia $V_i = \{1, ..., i\}$, sia G_i il sottografo di G indotto da V_i e sia C_i il cociclo di V_i in G, ovvero l'insieme di tutti gli archi di G che hanno un vertice finale in V_i.

Proposizione 2.5. *In ogni immersione planare di G, per ogni $i = 1, ..., n$ tutti gli archi di C_i appartengono alla faccia esterna di G_i (figura 2.4).*

La dimostrazione di questa Proposizione che omettiamo per brevità è riportata in [8]. Questa proposizione implica che per ogni arco (u, v) in C_i, con $u \in V_i$, il suo vertice virtuale w e i due semi-archi (u, w) e (w, v) appartengono alla faccia esterna di G_i. Allora si ha che tutti i semi-archi (u, w), $u \in V_i$ possono essere inseriti in una lista di rotazione secondo l'ordine che hanno sulla faccia esterna di G_i. L'ordine dei semi-archi impone un ordine ai vertici virtuali: la lista corrispondente dei vertici virtuali viene chiamata la *frontiera* F_i di G_i. La proposizione seguente è importante per stabilire la correttezza dell'algoritmo che verrà presentato successivamente.

Proposizione 2.6. *Per ogni $i = 1, ..., n$ tutti i vertici virtuali che portano il numero $i + 1$ occupano posizioni consecutive in F_i.*

Figura 2.4: Illustriamo la proposizione 2.5: (a) numerazione unilaterale del grafo in figura 2.2 (a); (b) G_3; (c) $i = 3$, $V_3 = \{1, 2, 3\}$, $C_3 = \{(1, 6), (2, 5), (3, 4), (1, 4)\}$.

Dim. Tutti i vertici virtuali che sono numerati $i + 1$ appartengono a F_i perché ognuno di questi vertici è adiacente a qualche vertice che ha una numerazione più bassa grazie alla numerazione unilaterale. Supponiamo che in F_i ci siano tre vertici virtuali w_h, w_{h+1} e w_k $(k > h + 1)$ tali che w_h e w_k portino il numero $i+1$ mentre w_{h+1} no. Siano $(u_h, w_h), (u_{h+1}, w_{h+1})$ e (u_k, w_k) rispettivamente i corrispondenti semi-archi con $u_h, u_{h+1}, u_k \in V_i$. Allora sia u_h che u_k devono appartenere alla faccia esterna di G_i poiché G è planare. Sia $P(u_h, u_k)$ il cammino ottenuto partendo da u_h e attraversando la faccia esterna di G_i in senso orario fino a raggiungere il vertice u_k. Sia $v = i + 1$; consideriamo la curva chiusa $C = vP(u_h, u_k)v$. Esiste un unico semi-arco (w_{h+1}, z) uscente da w_{h+1}. Per la planarità di G e per la proposizione 2.4, z appartiene all'interno di C (vedi figura 2.5). Allora il vertice a numerazione più alta di G all'interno di $C(z)$ deve essere un pozzo; questa affermazione contraddice il fatto che l'unico pozzo di G appartiene alla faccia esterna di

Figura 2.5: Dimostrazione della proposizione 2.6.

G. □

Data una immersione rettilinea di un grafo standard, una *linea di supporto* dell'immersione è ogni linea orizzontale o verticale tra alcuni vertici. Descriviamo un algoritmo che trova una immersione rettilinea che produce al massimo tre bend per arco, successivamente modificheremo questo algoritmo in modo tale che ogni arco avrà al massimo due bend:

TWO-BEND
1 **for** $i = 1$ **to** n
2 immergi tutti i semi-archi entranti nel vertice $v = i$ secondo il valore di $d^-(v)$ come mostrato in figura 2.6 (a), posizionando il vertice i sulla linea orizzontale $y = i$.
3 immergi tutti i semi-archi uscenti dal vertice $v = i$ secondo il valore di $d^+(v)$ come mostrato in figura 2.6 (b).
4 **end for**

In particolare osserviamo che l'istruzione della riga 2 posiziona il vertice 1

Figura 2.6: Immergiamo tutti i semi-archi (a) entranti in v e (b) uscenti da v.

sulla linea orizzontale $y = 1$ mentre l'istruzione della riga 3 immerge tutti i semi-archi uscenti dal vertice 1. La figura 2.6 descrive l'esecuzione di un singolo passo dell'algoritmo TWO-BEND. In (a) i vertici virtuali (bianchi) sono stati immersi precedentemente mentre il vertice v (nero) e i semi-archi entranti in v vengono immersi nel passo corrente. In (b) vengono immersi tutti i semi-archi uscenti da v e i loro vertici finali virtuali. Se la sorgente ha grado 4 allora il semi-arco più a destra uscente da s, detto h_s, ha due bend. Se la radice ha grado 4, allora il semi-arco più a destra entrante in t, detto h_t ha due bend.

Per ogni semi-arco (v, w) uscente da v con una o due bend, l'ascissa del vertice

Figura 2.7: Illustriamo l'esecuzione dell'algoritmo TWO-BEND sul grafo in figura 2.2 (a): (a) $F_1 = \{6, 2, 3, 4\}$; (b) $F_2 = \{6, 5, 3, 3, 4\}$; (c) $F_3 = \{6, 5, 4, 4\}$; (d) $F_4 = \{6, 5, 5, 6\}$; (e) $F_5 = \{6, 6, 6\}$; (f) immersione planare di G.

virtuale w è definita in modo tale che la linea di supporto verticale passante per w è equidistante dalla linea di supporto più vicina alla sua sinistra e alla sua destra (se una delle due linee non c'è, allora w viene posizionato a distanza 1 dall'altra linea). L'ordinata di w è irrilevante: possiamo prenderla uguale a $i+\frac{1}{2}$. Una volta ottenuta una immersione di G', possiamo cambiare le ascisse dei vertici in modo tale che la distanza tra le linee verticali di supporto consecutive sia 1. La figura 2.7 mostra ad ogni passo i l'immersione rettilinea corrente generata dall'algoritmo TWO-BEND.

Teorema 2.7. *Dato un grafo planare G unilateralmente numerato, l'algoritmo costruisce una immersione planare con al massimo tre bend per arco.*

Dim. La planarità del disegno segue dalle proposizioni 2.4-2.6. Inoltre la costruzione garantisce che l'immersione finale è equivalente all'immersione data in input e la faccia esterna del grafo G non cambia. Il disegno è rettilineo. Il fatto che ogni arco abbia al massimo tre bend segue dalla costruzione. Infatti ogni arco che viene disegnato dall'algoritmo ha al massimo un bend, ad eccezione degli archi h_s ed h_t incidenti nei vertici s e t rispettivamente. Quindi ogni arco che non contiene h_s nè h_t ha al massimo due bend, la forma di ogni arco di questo tipo è una di quelle mostrate in figura 2.8. Componendo h_s o h_t con un semi-arco 0-bend otteniamo un arco con due bend (figura 2.9 (a)), in tutti gli altri casi otteniamo un arco con tre bend (figura 2.9 (b)). □

Corollario 2.1. *Se s e t hanno grado 4 e (s,t) è un arco di G allora l'immersione rettilinea di (s,t) ha tre piegature.*

Dim. Questo corollario segue dal fatto che (s,t) è sia l'arco più a destra uscente da s che l'arco più a destra entrante in t e in entrambi i casi è ottenuto componendo un semi-arco con due bend con un semi-arco con un bend (figura 2.10). □

Figura 2.8: (a) 0-bend; (b) 1-bend; (c) vertical handle; (d) zig-zag

Figura 2.9: (a) Composizione di h_s con un semi-arco 0-bend; (b) composizione di h_s con un semi-arco 1-bend.

Lemma 2.8. *Per ogni $i = 1, ..., n$ la cardinalità di F_i è al massimo $n + 2$.*

Dim. Sia n_k il numero di vertici il cui numero di archi entranti è $k = 0, 1, 2, 3, 4$. Dal momento che $n_0 = n_4 = 1$ e $n_1 = n_3$ si ha che $n = 2+2n_1+n_2$ dove n è il numero di archi di G e quindi $2n_1 \leq n - 2$. Osserviamo che al passo $i = 1$ la cardinalità di F_i è al massimo 4, questo valore aumenta ogni volta di 2 quando viene immerso un vertice che ha un arco entrante e non aumenta mai in nessun altro caso. La cardinalità di F_i si mantiene sempre minore di $4 + 2n_1 \leq n + 2$. □

Teorema 2.9. *L'algoritmo* TWO-BEND *è lineare nel tempo e nello spazio.*

Figura 2.10: Illustriamo il corollario 2.1.

Dim. Il tempo di esecuzione dell'algoritmo è banalmente $O(n)$; inoltre l'algoritmo ha una complessità lineare nello spazio grazie al Lemma 2.8. □

Teorema 2.10. *Il numero totale di bend è* $2n + 4$.

Dim. Il numero totale di linee di supporto orizzontali prodotte dall'algoritmo è uguale al numero di vertici più due, dal momento che abbiamo introdotto una linea orizzontale in più sia per la radice che per il pozzo. Inoltre ogni linea ha al più due bend. □

Teorema 2.11. *L'area occupata dal disegno è al massimo* $(n + 1)^2$.

Dim. L'altezza del disegno prodotto dall'algoritmo è $n + 1$ dal momento che il numero di linee di supporto orizzontali è $n + 2$. Inoltre la larghezza è uguale alla cardinalità della frontiera meno 1, questo valore è al massimo $n + 1$ per il Lemma 2.8. □

2.3 Eliminazione degli archi del tipo zig-zag

In questo paragrafo mostreremo come eliminare gli archi del tipo zig-zag facendo alcune modifiche all'algoritmo TWO-BEND presentato precedente-

mente.

Per eliminare le "horizontal handles" inseriamo due nuovi vertici s_0 e t_0 all'interno dei segmenti orizzontali di h_s e di h_t rispettivamente. Sia $G_0 = (V_0, E_0)$ il grafo risultante con sorgente s_0 e pozzo t_0. Chiameremo *normali* gli archi con un bend o con "vertical handles".

Osserviamo che gli archi del tipo zig-zag sono formati da due segmenti orizzontali e da un segmento verticale. Un *cammino zig-zag* è un cammino che consiste solo in segmenti del tipo zig-zag. Presi comunque due vertici a, b di G_0 esiste al più un cammino zig-zag che li collega. Se questo cammino esiste lo chiameremo $Z(a, b)$ ed i vertici a, b saranno chiamati *equivalenti*. Assumiamo per convenzione che ogni vertice è equivalente a se stesso. Allora si ha che la relazione che abbiamo appena definito è una relazione riflessiva, simmetrica e transitiva: le corrispondenti classi di equivalenza $V_1, ..., V_q$ (*insiemi zig-zag)* formano una partizione di V_0.

Descriveremo una procedura che elimina tutti gli archi del tipo zig-zag, sostituendoli con archi 0-bend. La strategia è quella di assegnare a tutti i vertici che appartengono allo stesso insieme zig-zag la stessa ordinata mentre l'ascissa rimane invariata.

Diremo che V_j è *sopra* V_i ($V_j > V_i$) se esiste almeno un arco (u, v) con $u \in V_i$ e $v \in V_j$.

Lemma 2.12. *La relazione "stare sopra" non è simmetrica.*

Dim. Supponiamo per assurdo che esistano due archi (u, v), (w, z) tali che $u, z \in V_i$ e $v, w \in V_j$. Senza perdita di generalità supponiamo $x_u < x_z$. Consideriamo le due semirette

$$r_u = \{(x, y_u) : -\infty < x \leq x_u\} \text{ e } r_z = \{(x, y_z) : x_z \leq x < +\infty\}.$$

Figura 2.11: La linea spezzata L.

Figura 2.12: Dim del Lemma 2.12 caso 1.

Consideriamo la linea spezzata $L = r_u Z(u, z) r_z$ e le due regioni aperte R^+ e R^- che si trovano sopra e sotto L rispettivamente (figura 2.11). Per l'algoritmo TWO BEND v deve appartenere a R^+ e w deve appartenere ad R^-; di conseguenza $v \neq w$. Dobbiamo distinguere due casi.

1. $Z(v, w)$ *è incidente da destra.* In questo caso dal momento che l'immersione rettilinea di G_0 è planare e l'arco (u, v) è normale, si ha $x_w > x_z$. A questo punto abbiamo bisogno di un "horizontal handles" per collegare w e z, ma questa è una contraddizione perché (w, z) è anche un

Figura 2.13: Dim. del Lemma 2.12, caso 2.

arco normale.

2. $Z(v, w)$ *è incidente da sinistra*. In questo caso, dal momento che l'immersione rettilinea di G_0 è planare e l'arco (u, v) è normale, si ha $x_w < x_v \leq x_u < x_z$. Abbiamo quindi bisogno di un "horizontal handles" per collegare w e z e questa è una contraddizione. □

Questo Lemma implica che se esiste almeno un arco che esce da V_i ed entra in V_j allora tutti gli archi che collegano V_i e V_j escono da V_i ed entrano in V_j. Per trovare l'ordinata di ogni vertice usiamo il metodo seguente.

Costruiamo un grafo $D = (N, A)$ dove $N = \{V_1, ..., V_q\}$ e $(u, v) \in A$ se in G_0 esiste un arco uscente da $u \in V_i$ ed entrante in $v \in V_j$. Questo grafo D può essere generato in un tempo lineare. Per il Lemma 2.12 D possiede una orientazione bipolare; sia V_1 la sorgente. Definiamo il *livello* di un vertice V_i, indicato con livello(V_i), il massimo numero di archi di un cammino da V_1 a V_i. Il livello di un vertice del grafo D può essere calcolato in un tempo lineare usando questa relazione:

$$\text{livello}(V_1) = 0;\ \text{livello}(V_j) = \max\{\text{livello}(V_i) + 1 : V_iV_j \in A\}$$

Dal momento che $\{V_1, ..., V_q\}$ è una partizione dell'insieme dei vertici di G_0, l'ordinata di tutti i verici di G può essere calcolata in questo modo:

$$y(w) = y(s_0)+\text{livello}(V_j) \text{ per ogni } w \in V_j, j = 1, ..., q$$

Osserviamo che queste modifiche dell'algoritmo TWO-BEND hanno una complessità lineare nel tempo e nello spazio. La figura 2.14 (a) mostra il grafo D corrispondente all'immersione rettilinea in figura 2.7, ogni vertice è un insieme zig-zag numerato con il suo livello; la figura 2.7 (b) mostra l'immersione rettilinea modificata.

Figura 2.14: (a) Il grafo D che corrisponde al grafo in figura 2.7; (b) l'immersione rettilinea di G dove sono stati modificati gli archi del tipo zig-zag.

2.4 Algoritmo per grafi biconnessi che produce al massimo due bend per ogni arco

In questo paragrafo mostreremo che, se G non è l'ottaedro, l'algoritmo presentato nelle pagine precedenti produce una immersione con al massimo due bend per ogni arco da un generica immersione e una numerazione unilaterale di G. Per provare questo risultato abbiamo bisogno di alcune definizioni ed alcuni Lemmi. Una *coppia separatrice* di un grafo generico biconnesso $G = (V, E)$ è una coppia di vertici $\{p, q\}$ tali che il sottografo indotto da $V - \{p, q\}$ non è connesso. Dato un grafo planare biconnesso G si ha che ogni immersione di G può essere ottenuta da ogni altra immersione attraverso un numero finito di riflessioni attorno a coppie separatrici (figura 2.15; per una definizione formale di riflessione vedi [5]).

Figura 2.15: Una riflessione attorno alla coppia separatrice $\{p, q\}$.

Lemma 2.13. *Sia $\{p,q\}$ una coppia separatrice di G e sia (p, q) un arco appartenente alla faccia esterna di una data immersione planare di G. Allora si ha che, ad eccezione dell'arco (p, q), ogni arco che appartiene alla faccia esterna rimane sulla faccia esterna dopo una riflessione attorno a $\{p,q\}$.*

Sia $B(G)$ il grafo blocco di G, ovvero il grafo i cui vertici sono i blocchi (le componenti biconnesse) di G e due vertici sono adiacenti se e solo se i

corrispondenti blocchi hanno un punto di articolazione in comune. $B(G)$ è un albero e le sue foglie vengono chiamate *foglie del blocco*.

Lemma 2.14. *Se G è un grafo biconnesso ed x un suo vertice qualunque allora esiste almeno un vertice y adiacente ad x tale che $\{x,y\}$ non è una coppia separatrice.*

Dim. Il grafo $G' = G - x$ è connesso e se T è un qualunque blocco di foglie di G', allora x deve essere adiacente a qualche vertice y di T che non è un punto di articolazione per G', altrimenti G non sarebbe biconnesso. Quindi $\{x, y\}$ non è una coppia separatrice. □

Lemma 2.15. *Sotto le ipotesi del Lemma 2.13 esiste un vertice r all'interno del dominio della faccia esterna di G tale che $\{p,r\}$ non è una coppia separatrice. Inoltre se il grado di p è al massimo 4 facendo al più due riflessioni otteniamo una immersione planare di G tale che l'arco (p, r) appartiene alla faccia esterna.*

Dim. Segue dal Lemma 2.14 e dal fatto che ogni vertice ha al massimo grado 4. □

Lemma 2.16. *Sia G un grafo biconnesso standard con almeno una faccia diversa da un triangolo. Partendo da una immersione planare generica di G la cui faccia esterna consiste almeno in quattro archi e operando al più quattro riflessioni, otteniamo una immersione planare di G e quattro vertici distinti s, t, u, v, tali che (s, v) e (u, t) appartengono alla faccia esterna e nè $\{s,v\}$ nè $\{u,t\}$ sono una coppia separatrice di G.*

Dim. Siano s e t due vertici non adiacenti appartenenti alla faccia esterna C di G. Sia (s, x) l'arco più a destra uscente da s e sia (y, t) l'arco più a destra entrante in t. Se $\{s, x\}$ non è una coppia separatrice per G sia $v = x$. Altrimenti per il Lemma 2.14 esiste un vertice v all'interno del

dominio di C tale che $\{s, v\}$ non è una coppia separatrice. Per il Lemma 2.15 facendo al più due riflessioni otteniamo una immersione planare per G tale che (s, v) appartiene alla faccia esterna. Supponiamo che l'immersione planare di G abbia questa proprietà. Osserviamo che per il Lemma 2.13 l'arco (y, t) rimane sulla faccia esterna. Se $\{y, t\}$ non è una coppia separatrice, sia $u = y$. Altrimenti per il Lemma 2.14 esiste un vertice u adiacente a t all'interno del dominio della faccia esterna (e quindi diverso da s, t, v) tale che $\{u, t\}$ non è una coppia separatrice. Per il Lemma 2.15 facendo al più due riflessioni possiamo spostare l'arco (u, t) sulla faccia esterna. Per il Lemma 2.13 l'arco (s, v) rimane sulla faccia esterna e la tesi è dimostrata (figura 2.16). □

Figura 2.16: Un grafo per il quale sono necessarie quattro riflessioni.

Se G è un grafo biconnesso e (x, y) è un arco di G, la *contrazione* G_x^y è il

grafo ottenuto da G rimuovendo x e collegando ad y tutti i vicini di x che non erano già collegati ad y in G.

Teorema 2.17. *Sia G un grafo biconnesso standard, con almeno una faccia diversa da un triangolo. Allora G ha una immersione ed una numerazione unilaterale tale che gli archi $(1,2)$ e $(n-1,n)$ appartengono entrambi alla faccia esterna.*

Dim. Scegliamo i vertici s, t, u, v come nel Lemma 2.16. Consideriamo il grafo $G' = (G_s^v)_t^u$. Dal momento che $\{s, v\}$, $\{u, t\}$ non sono coppie separatrici di G e $\{u, t\}$ mantiene questa proprietà dopo la contrazione dell'arco (s, v) il grafo G' è biconnesso e si ha che esiste una orientazione bipolare per G' con sorgente v e pozzo u. Quindi i vertici di G' possono essere unilateralmente numerati da 2 a $n-1$. Ora, se il vertice s viene numerato 1 e il vertice t viene numerato n otteniamo una numerazione unilaterale di G con la proprietà cercata. Osserviamo che in questa numerazione i vertici s, v, u, t sono numerati rispettivamente $1, 2, n-1, n$. □

Teorema 2.18. *Se G è un ottaedro, allora esistono una immersione di G e una numerazione unilaterale di partenza tali che l'algoritmo produce una immersione con al massimo due bend per arco.*

Dim. Se G non è l'ottaedro, l'arco (s, t) appartiene a G e per il corollario 2.1 ha al massimo tre bend. Tra tutti i grafi standard, gli unici che hanno solo facce triangolari, oltre all'ottaedro, sono $K_3, K_4, K_5 - e$, per ogni arco $e \in K_5$. Nei primi due casi, ogni vertice ha grado minore di 4 e l'algoritmo non produce nessun arco con tre bend. Nel terzo caso esiste un vertice di grado 3 sulla faccia esterna. Allora possiamo scegliere s, u e t sulla faccia esterna in modo tale che s è il vertice di grado 3, (s, t) è l'arco più a destra uscente da s e (u, t) è l'arco più a destra entrante in t. Allora esiste una numerazione unilaterale tale che s, u e t vengano numerati $1, 4$ e 5 rispettivamente. Allora si ha che sia l'arco (s, t) che l'arco (u, t) hanno due bend (figura 2.18).

Figura 2.17: (a) K_3; (b) K_4; (c) $K_5 - e$.

Per ogni altro grafo standard esiste almeno una faccia diversa da un triangolo. Se partiamo da una immersione di G tale che la faccia esterna ha almeno quattro archi, allora per il Teorema 2.17 esistono una immersione e una numerazione unilaterale tale che i due archi con "horizontal handles" sono $(1, 2)$ e $(n-1, n)$. Dal momento che il vertice 2 ha solo un arco entrante e il vertice $n-1$ ha solo un arco uscente (per la numerazione unilaterale) gli archi $(1, 2)$ e $(n-1, n)$ hanno due bend. Dunque l'immersione che abbiamo costruito produce al massimo due bend per arco. □

2.5 Il caso generale

Se vogliamo generalizzare l'algoritmo presentato nella sezione 2.2 ad un grafo connesso (non necessariamente biconnesso) standard G le difficoltà principali sono due: in primo luogo potrebbe non esistere una orientazione bipolare, quindi dobbiamo considerare una orientazione aciclica più generale con un'unica sorgente e più pozzi; in secondo luogo alcune volte nel caso in cui ci sono due sottografi connessi H_1 e H_2 separati da un punto di articolazione w dobbiamo immergere H_2 all'interno di una faccia C di H_1; quando questo succede dobbiamo assicurarci che almeno un vertice di C venga immerso

Figura 2.18: Immersione del grafo $K_5 - e$.

dopo che tutti i vertici di H_2 sono stati immersi. Esaminiamo separatamente queste difficoltà. Abbiamo bisogno di alcune definizioni. Data una immersione di G, un blocco di G è detto *terminale* se la sua faccia esterna contiene al massimo un punto di articolazione per G. Ovviamente tutti i blocchi di foglie sono terminali, ma non è vero il viceversa.

È noto che un grafo G ammette una orientazione bipolare se e solo se il suo grafo blocco è un cammino [23]. Dal momento che dobbiamo occuparci di grafi connessi generici dobbiamo introdurre una orientazione più generale che, data una immersione Γ per G soddisfa le seguenti proprietà:

1. L'orientazione indotta in ogni blocco è bipolare e sia la sorgente che il pozzo appartengono alla faccia esterna del blocco;

2. G possiede un'unica sorgente s, inoltre s appartiene alla faccia esterna di G e non è un punto di articolazione per G; sia B_1 il blocco che contiene s;

3. tutti i pozzi di G appartengono a blocchi terminali, B_1 non contiene pozzi di G quando ha esattamente un punto di articolazione sulla faccia esterna.

Una orientazione che verifica (1)-(3) viene chiamata *divergente*. Una orientazione divergente è aciclica e nessun punto di articolazione di G è un pozzo per G. Osserviamo che ogni numerazione dei vertici che concorda con una orientazione divergente ha la proprietà che ogni vertice $j > 1$ è sempre adiacente a qualche vertice $i < j$. Ora parleremo di come trovare una immersione che produce al massimo due o tre bend per arco di un grafo connesso arbitrario. L'algoritmo TWO BEND presentato nella sezione 2.2 produce una immersione di questo tipo se riceve in input una opportuna immersione ed una appropriata orientazione divergente. Il caso più semplice è quello in cui l'immersione di G ha la seguente proprietà: la faccia esterna di G è l'unione delle facce esterne dei blocchi di G. In questo caso si ha che tutti i blocchi terminali sono blocchi di foglie. Per ottenere una immersione di G che produce al massimo tre bend per arco abbiamo bisogno di una generica orientazione divergente per G, Γ, e di una opportuna numerazione dei vertici. Una orientazione divergente per G può essere costruita in un tempo $O(n)$. Infatti possiamo costruire in un tempo $O(n)$ l'albero blocco $B(G)$ la cui radice sarà uno dei blocchi, detto B_1. Partendo da B_1 e procedendo secondo l'ordine dato dalla visita in profondità troviamo in ogni blocco una orientazione bipolare che verifica le proprietà (1)-(3). Tramite questa orientazione otteniamo una numerazione dei vertici coerente. A questo punto per trovare una immersione di G che produca al massimo tre bend per arco usiamo sostanzialmente l'algoritmo presentato nella sezione 2.2 con l'unica differenza che ci potrebbero essere più pozzi e l'algoritmo termina solo dopo che l'ultimo pozzo è stato immerso. Dal momento che valgono ancora le proposizioni 2.4-2.6 l'algoritmo è corretto. Inoltre, se G non è l'ottaedro, l'algoritmo produce una immersione di G con al massimo due bend per arco partendo da una immersione e da

una orientazione divergente con questa proprietà:

4. Se la sorgente s ha grado 4, allora esiste un successore di s che appartiene alla faccia esterna di B_1 e ha solo un arco entrante; se un pozzo t_i di G appartenente ad un blocco terminale B_i ha grado 4, allora esiste un predecessore di t_i che appartiene alla faccia esterna di B_i e che ha un solo arco uscente.

Il Lemma 2.15 e 2.16 ci assicurano che una immersione e una orientazione che godano di queste proprietà possono essere trovate in un tempo $O(n)$. A questo punto procedendo come nella dimostrazione del Teorema 2.18 possiamo trovare una opportuna numerazione dei vertici che dà luogo ad una immersione con al massimo due bend per arco. Prima di discutere il caso generale abbiamo bisogno di alcune definizioni. Sia C un ciclo di G, un *ponte* H (rispetto a C) è una componente connessa di $G - C$ più tutti gli archi di G che collegano questa componente a C. Se H appartiene all'interno del dominio di C allora H viene chiamato *ponte interno*. I vertici di H che appartengono a C sono chiamati *vertici di collegamento* di H. Se H ha solo un vertice di collegamento w, allora w è un punto di articolazione per G. In ogni immersione di G, w appartiene almeno a due facce. Se disponiamo H all'interno di ogni altra faccia incidente w otteniamo una immersione di G che non è equivalente all'immersione data. Ogni operazione di questo tipo viene chiamata *riposizionamento* di H.

Data una immersione di G, un ciclo viene detto *minimale* quando ognuno dei suoi ponti interni ha un unico vertice di collegamento. Osserviamo che se H è un ponte interno e ha un vertice di collegamento, allora H è anche un ciclo minimale. Ora dimostreremo che l'algoritmo TWO BEND presentato nella sezione 2.2 produce una immersione di G con al massimo due bend per arco se riceve in input un'opportuna immersione ed un'appropriata numerazione dei vertici di G. Sia $B(G)$ l'albero blocco di G e siano $\{B_1, B_2, ..., B_k\}$ i suoi

insiemi di vertici. L'algoritmo di immersione è basato sul seguente risultato.

Teorema 2.19. *Partendo da una generica immersione, possiamo trovare una immersione* Γ *e una orientazione divergente tale che vale la proprietà (4) ed è soddisfatta la seguente proprietà:*

5. Per ogni ciclo minimale C *di* G *e per ogni ponte interno* H *di* C*, l'unico vertice di collegamento con* H *ha un successore in* C*.*

Dim. Daremo una dimostrazione algoritmica. Troviamo i punti di articolazione e i blocchi di G e costruiamo l'albero blocco $B(G)$. A questo punto generiamo una immersione per ogni blocco e alla fine una immersione per G ordinando, per ogni punto di articolazione w di G le liste di adiacenza di w che corrispondono ai blocchi separati da w. Scegliamo sulla faccia esterna di G un vertice s che non è un punto di articolazione per G. Se s ha grado 4 verifichiamo che almeno uno dei suoi vicini che appartiene alla faccia esterna non formi con s una coppia separatrice; se non è così possiamo trovare un altro vertice al più dopo due riflessioni per il Lemma 2.15. Sia B_1 il blocco che contiene s e la radice $B(G)$. Esploriamo $B(G)$ con una visita in profondità: sia B_i il blocco corrente. Definiamo la sorgente s_i di B_i in questo modo: se $i = 1$ allora $s_i = s$; se $i \neq 1$ allora s_i è l'unico punto di articolazione di G che separa B_i da suo padre in $B(G)$. Consideriamo la faccia esterna C_i di B_i. Se C_i contiene almeno un punto di articolazione w per G, $w \neq s_i$, allora scegliamo w come pozzo t_i di B_i. Se C non contiene punti di articolazione di G scegliamo come pozzo un vertice generico u diverso da s_i tale che se u ha grado 4 almeno uno dei suoi vicini che appartiene alla faccia esterna C_i non forma con u una coppia separatrice; se questo non è possibile, allora per il Lemma 2.15 possiamo trovare un vertice che gode di questa proprietà dopo al più due riflessioni. Se dopo una riflessione alcuni punti di articolazione w appartengono alla nuova faccia esterna di B_i allora scegliamo w come pozzo t_i. Alla fine, per ogni vertice w di B_i che è un punto di articolazione per G

riposizioniamo, se necessario, i ponti collegati solo a w all'interno di una faccia appropriata incidente a w in modo tale che valga la proprietà (5). Infatti ogni ponte H di questo tipo è contenuto all'interno del dominio di un unico ciclo minimale C. Consideriamo il blocco B che contiene C. Se w appartiene alla faccia esterna di B (figura 2.19 (a)), allora H può essere riposizionato all'interno della faccia esterna di B; altrimenti (figura 2.19 (b))w non può coincidere con il pozzo di B che appartiene alla faccia esterna di B. □

Figura 2.19: Riposizionamento di ponti interni.

Una numerazione dei vertici di G da 1 a n è detta *annidata* (rispetto ad una data immersione di G) se per ogni ciclo minimale C e per ogni ponte interno H di C esiste almeno un vertice i di C tale che $i > j$ per ogni vertice j di H.

Corollario 2.2. *Data una immersione* Γ *e una orientazione aciclica come nel Teorema 2.19, allora esiste una numerazione annidata rispetto a* Γ *che concorda con l'orientazione.*

Dim. Una volta che conosciamo l'orientazione possiamo facilmente ottenere una numerazione in questo modo: introduciamo un superpozzo t all'interno della faccia esterna di G. Ogni pozzo t_i può essere di due tipi: (a) appartiene alla faccia esterna di G; (b) appartiene alla faccia esterna di qualche ponte interno H di un unico ciclo minimale C. Nel caso (a) aggiungiamo un falso arco da t_i a t. Nel caso (b) aggiungiamo un falso arco da t_i ad un succes-

sore dell'unico vertice di collegamento di H. Un successore di questo tipo esiste sempre per il Teorema 2.19. Dal momento che il grafo così ottenuto è biconnesso, esiste una numerazione unilaterale dei suoi vertici tale che s è numerato 1 e t è numerato $n + 1$ e questa numerazione induce una numerazione annidata dei vertici di G che concorda con l'orientazione data di G. □

Per il grafo in figura 2.20 (a) la figura 2.20 (b) mostra una immersione, una orientazione aciclica e una numerazione annidata che soddisfano il teorema 2.19 e il Corollario 2.2.

Teorema 2.20. *Dato un grafo standard generico possiamo trovare una immersione, una orientazione divergente ed una numerazione annidata in tempo lineare.*

Dim. I punti di articolazione, i blocchi e l'albero blocco di G possono essere generati in un tempo lineare [24]. Le facce di una immersione data possono essere trovate con la stessa complessità [5]. Per trovare un'opportuna immersione di G abbiamo bisogno di considerare, per ogni blocco B_i, tutti i punti di articolazione di G che appartengono a B_i in modo tale da scegliere la sorgente e il pozzo del blocco, trovare una orientazione bipolare per B_i e riposizionare i ponti interni in modo opportuno. Per fare quest'ultima operazione abbiamo bisogno di informazioni riguardo gli archi incidenti il punto di articolazione w che abbiamo preso in considerazione: le loro rotazioni e i loro blocchi. Osserviamo che, una volta che conosciamo l'orientazione di ogni blocco attraverso la lista ordinata dei successori di ogni vertice possiamo ottenere facilmente una orientazione di G che soddisfa la proprietà (5) ordinando, per ogni vertice w, le liste di adiacenza dei successori di w che corrispondono ai blocchi separati da w. Dal momento che esistono al più due ponti interni collegati ad un punto di articolazione w e ogni operazione di riposizionamento viene eseguita e richiede un tempo costante, ne segue

che il tempo totale per trovare una immersione e una orientazione è lineare. Una volta che conosciamo l'orientazione la costruzione del grafo aumentato e la sua numerazione unilaterale possono essere ottenute in un tempo lineare. Una volta ottenuta una numerazione annidata, l'algoritmo che immerge G producendo al massimo due bend per arco è identico a quello della sezione 2.2 eccetto per il fatto che ci sono più pozzi. □

Teorema 2.21. *Data una numerazione annidata l'algoritmo* TWO BEND *descritto nella sezione 2.2 produce una immersione planare con al massimo due bend per arco.*

Dim. La planarità è assicurata dalla numerazione annidata. Osserviamo che le proposizioni 2.4-2.6 valgono ancora. Inoltre la proprietà (4) assicura che l'immersione produce al più due bend per arco. □

Il nostro risultato finale riguarda un limite massimo per il numero totale β di bend. Senza perdita di generalità possiamo supporre che G abbia almeno un punto di articolazione sulla sua faccia esterna. Allora possiamo ottenere in un tempo lineare, dopo aver riposizionato alcuni ponti, una immersione in cui la faccia esterna di G contenga la faccia esterna di alcuni blocchi terminali. Assumiamo che il vertice 1 appartenga ad uno di questi blocchi.

Teorema 2.22. *Per un generico grafo standard con n vertici, il numero totale di bend dell'immersione prodotta dall'algoritmo soddisfa la disuguaglianza*

$$\beta \leq (\tfrac{7}{3})n \quad (2.1)$$

Dim. Sia G un grafo standard con n vertici. Sia q la cardinalità massima di un blocco terminale di G e sia b_h il numero di questo blocco con cardinalità h, $h = 3, ..., q$.

Per il Corollario 1.1 si ha che in ogni grafo planare il numero di archi è al più $3n - 6$, quindi in ogni blocco terminale con cardinalità $h \leq 5$ ci sono almeno due vertici di grado al più 3. Dal momento che ogni vertice di grado al più 3 produce al massimo un bend e poiché in ogni blocco terminale esiste al più un "horizontal handle" il contributo di ogni blocco terminale che ha cardinalità $h \leq 5$ è al più di $2h$ bend. Allora si ha che

$$\beta \leq 2n + 2(b_6 + ... + b_q). \quad (2.2)$$

Trattiamo separatamente il caso in cui G consiste in due blocchi di 6 vertici con un vertice in comune. In tutti gli altri casi si ha

$$6(b_6 + ... + b_q) \leq n$$

che insieme alla (2.2) implica (2.1). Trattiamo il caso di cui abbiamo parlato prima. In primo luogo il vertice in comune deve avere grado 2 all'interno di ogni blocco. Dal momento che (4,4,4,4,4,2) non può essere la sequenza dei gradi dei vertici di un grafo planare, eccetto che per il caso in cui entrambi i blocchi siano cicli, G deve avere almeno due vertici di grado 3. Per il Teorema 2.10 si ha che $\beta \leq n_3 + 2n_4 + 4$. Poiché $2 \leq n_3 \leq 11 - n_4$ si ha che $\beta \leq 24 \leq (\frac{7}{3})n$. Allora la (2.1) vale anche in questo caso. □

Figura 2.20: (a)Un grafo standard G e la sua immersione iniziale; (b) immersione, orientazione divergente e numerazione annidata richieste dall'algoritmo di immersione rettilinea.

Capitolo 3

Immersione di grafi su griglie multidimensionali

3.1 Introduzione

Fino a questo punto abbiamo affrontato il problema di immergere grafi planari in griglie bidimensionali. Questa nozione di immersione è particolarmente utile per la fabbricazione di chip VLSI (*Very Large Scale Integration*) e nel disegno di circuiti stampati. Bisogna considerare che nel caso in cui il grafo che vogliamo immergere non sia planare dobbiamo usare una griglia multidimensionale. In pratica gli archi che si intersecano devono appartenere a livelli diversi; per questo nel momento in cui appartengono allo stesso livello è necessario che uno di loro cambi di livello prima di incrociare l'altro. Questo cambiamento di livello avviene realizzando un *contact cut* tra i due livelli. Solitamente l'uso di troppi contact cut porta ad un aumento dell'area occupata dall'immersione del grafo e della probabilità di produrre chip difettosi.

Nella sezione 3.2 introdurremo due modelli di disegno che corrispondono ai chip VLSI e ai circuiti stampati e discuteremo alcune proprietà elementari e

alcune differenze tra questi due modelli. Introdurremo anche il concetto di immersione rispetto ad una disposizione prestabilita dei vertici sulla griglia. Nelle sezioni 3.3 e 3.4 enunceremo una serie di risultati relativi rispettivamente all'area occupata da un grafo immerso secondo il primo modello e all'area occupata da una classe di grafi immersa in base al secondo modello.

3.2 Modelli

In questo Capitolo una immersione di un grafo G in una griglia ad un livello è una applicazione che manda i vertici di G in segmenti orizzontali (possibilmente della stessa lunghezza di un nodo della griglia) e gli archi in cammini sulla griglia. I cammini che rappresentano archi non si intersecano eccetto che per i vertici in comune (vedi figura 3.1). Osserviamo che questo tipo di immersione è differente da quella definita nel capitolo 2, infatti in quel caso i vertici vengono mandati in nodi della griglia (figura 3.2).

Figura 3.1: Immersione su una griglia di un grafo G.

Consideriamo nelle pagine seguenti l'immersione su più livelli di un grafo G. Il primo modello multidimensionale, il modello SAL (*Single Active Layer*), consiste in due livelli che chiameremo verde e blu. Tutti i vertici di G sono

Figura 3.2: (a) Un grafo planare G; (b)immersione su una griglia di G secondo la definizione data nel capitolo 2.

immersi sul livello verde e gli archi sono rappresentati da cammini nella griglia che iniziano e finiscono sul livello verde, ma possono cambiare di livello ad ogni *contact cut*. Su ogni livello i cammini non si intersecano. Questo modello riprende la tecnologia utilizzata per costruire i circuiti VLSI dove i livelli blu e verde rappresentano rispettivamente il livello di metallo e quello di polisilicio di un chip VLSI. La figura 3.3 (a) mostra una immersione del grafo K_5 (il grafo completo con 5 vertici) secondo questo modello. I segmenti blu e verde che rappresentano gli archi sono disegnati rispettivamente con linee tratteggiate e non tratteggiate, le contact cut sono rappresentate da piccoli quadrati.

Il modello $k - PCB$ (*Printed Circuit Board*) invece consiste in una griglia di k livelli, dove k è un intero positivo, in cui ogni vertice di G viene immerso nella stessa posizione su ogni livello e gli archi vengono immersi come cammini nella griglia che possono cambiare di livello ad ogni contact cut. Un cammino che rappresenta un arco può iniziare e finire ad ogni livello, ma su ogni livello

Figura 3.3: Immersione di K_5 secondo il modello SAL (a) e secondo il modello 2-PCB(b).

i cammini non si intersecano eccetto che per i vertici in comune. Osserviamo che una immersione secondo il modello 1-PCB è una immersione sulla griglia ad un livello e che per ogni k il modello $k - PBC$ è diverso dal modello SAL. Il modello $k - PCB$ corrisponde ad un circuito stampato di k livelli in cui i pin dei chip sono presenti su ogni livello. La figura 3.3 (b) mostra una immersione del grafo K_5 secondo il modello 2-PCB. Dato un grafo G la *thickness* (*spessore*) di G è il più piccolo numero k tale che G sia l'unione di k grafi planari. Con l'espressione "unione di grafi planari" intendiamo che gli archi di G possono essere partizionati in k insiemi in modo tale che il grafo indotto da ogni insieme sia planare. Osserviamo ad esempio che il grafo K_5 ha thickness uguale a 2.

Lemma 3.1. *Un grafo G può essere immerso secondo il modello k-PCB senza l'uso di contact cut se ha thickness al più k.*

Definiamo più precisamente la nozione di "disposizione prestabilita". Una *disposizione* è una funzione che manda i vertici di un grafo G in un insieme

di segmenti orizzontali disgiunti appartenenti alla griglia rettangolare R. Diremo che una immersione di un grafo G in R *rispetta una disposizione* σ se i vertici sono immersi secondo l'immagine di σ. Spesso siamo interessati a trovare una *disposizione universale*, ovvero una disposizione σ che manda i vertici di un grafo G in un insieme N di n segmenti orizzontali in una griglia rettangolare R con la proprietà che, per qualche classe H di grafi con n vertici, se G è un grafo in H allora esiste una immersione di G che rispetta σ. Diremo che un algoritmo di immersione per una classe di grafi *rispetta una disposizione prestabilita* se esiste un insieme di disposizioni universali per quella classe tale che per ogni grafo G che appartiene alla classe e per ogni disposizione σ appartenente all'insieme delle disposizioni universali, l'algoritmo di immersione produce una immersione di G che rispetta σ. Osserviamo che algoritmi di immersione che rispettano una disposizione prestabilita possono essere molto utili in pratica se non richiedono un'area troppo grande. I risultati enunciati nella sezione 3 insieme al limite superiore dimostrato in [2] mostrano che per $k = 1, 2$ l'area occupata dall'immersione di un grafo con n nodi di thickness k è $O(n^3)$ se l'algoritmo rispetta una dispozione prestabilita altrimenti è $O(n^2)$. Chiudiamo questa sezione con alcune osservazioni sulle differenze tra il modello SAL e quello $k - PCB$. In primo luogo solo grafi planari possono essere immersi secondo il modello SAL senza usare contact cut mentre possiamo immergere grafi di thickness k secondo il modello $k-PCB$. Inoltre si può dimostrare che alcune importanti reti di collegamento possiedono una immersione senza contact cut secondo il modello 2-PCB che non produce un aumento dell'area rispetto al modello standard. Invece questo tipo di reti di collegamento richiede un numero di contact cut uguale a $\Omega(n \log^2 n)$ nel modello SAL.

3.3 Area occupata da un grafo immerso secondo il modello SAL

Molti risultati che sono stati dimostrati per l'immersione su griglie ad un livello di grafi planari valgono anche per il modello SAL. Ad esempio un risultato di Leiserson [15] fornisce una semplice tecnica per immergere ogni grafo con m archi secondo il modello SAL con area uguale a $O(m^2)$ e numero di contact cut pari a $O(m)$. In più il risultato di Leiserson mostra che ogni grafo planare con n vertici possiede una immersione secondo il modello SAL con $O(n \log^2 n)$. Leighton [14] dimostra un limite inferiore per l'area occupata da una immersione secondo il modello SAL di $\Omega(n \log n)$. Si può dimostrare che per ogni C con $1 \leq C \leq n/\log^2 n$, ogni grafo planare con n vertici può essere immerso secondo il modello SAL con un'area uguale a $O(n^2/C)$ e con un numero di contact cut uguale a $O(C)$. In più per ogni n e $C \leq n/4$ esiste un grafo planare con n nodi per il quale ogni immersione secondo il modello SAL con C contact cut occupa un'area pari a $\Omega(n^2/C)$. Il limite superiore è ottenuto usando un algoritmo che si basa su tre differenti tecniche di immersione. Il limite inferiore è un semplice corollario di un risultato di Shiloach [22, 29]. Un limite superiore simile è dimostrato in [6]. Questo risultato deriva da un importante Teorema che riportiamo di seguito. Ricordiamo che si dice che un grafo planare $G = (V, E)$ ha *gauge* al massimo G se esiste una immersione planare che ha la proprietà che per ogni vertice $v \in V$ e per ogni $w \in V$, w appartenente alla faccia esterna, esiste un cammino $p : v \rightsquigarrow w$ di lunghezza al massimo G.

Teorema 3.2. *Ogni grafo planare con n nodi di gauge G possiede una immersione su una griglia quadrata di lato pari a* $O((ng)^{\frac{1}{2}})$ *e quindi area uguale a* $O(ng)$.

Figura 3.4: Esempio di grafo planare G con gauge uguale a 2.

3.4 Area occupata da un grafo immerso secondo il modello $k - PCB$

In questo paragrafo enunceremo alcuni risultati relativi all'immersione di grafi secondo il modello $k - PCB$. Questi risultati si basano su un algoritmo di immersione per grafi planari su griglie bidimensionali che garantisce al massimo quattro piegature per arco e che tutti i vertici appartengano alla stessa linea e tutti gli archi che collegano i vertici vengono disegnati sopra o sotto questa linea (figura 3.5).

Figura 3.5: Immersione di un grafo planare.

Facendo alcune modifiche a questo algoritmo se ne può ottenere un altro che produce una immersione per un grafo G con n nodi con area pari a $O(n^3)$ e che rispetta una disposizione prestabilita dei nodi sulla griglia. Questa area è asintoticamente ottimale (un limite inferiore viene dimostrato in [2]). Si può dimostrare che per immergere grafi con n vertici di thickness 2 è necessaria un'area uguale a $O(n^2)$. Per $k \geq 3$ un grafo con n nodi di thickness k può essere immerso su k livelli in un'area pari a $O(n^3)$. Questo algoritmo deve ricevere in input la decomposizione di un grafo in k grafi planari e trovare questa decomposizione è un problema NP-completo [21]. Per grafi con n vertici di grado al massimo $2k$ si può dimostrare l'esistenza di un algoritmo che non ha bisogno di ricevere in input la decomposizione in grafi planari del grafo e produce un'area $O(n^3)$. Entrambi questi algoritmi rispettano una disposizione prestabilita dei nodi nella griglia. Il limite inferiore dimostrato in [2] mostra che l'area occupata dalla rappresentazione del grafoè minimale, anche se c'è la possibilità che ci sia un alto numero di contact cut. Per finire si può anche dimostrare che un grafo planare con n nodi e grado limitato occupa un'area pari a $(O(n^{\frac{3}{2}}(\log n)^2)$ su due livelli.

Osserviamo che tutti questi algoritmi non usano contact cut e questa è una proprietà molto richiesta perché un elevato numero di contact cut può causare la produzione di chip difettosi.

Capitolo 4

Appendice A: Esempi commentati

Verifica della planarità

Figura 4.1: L'algoritmo EMBED per la verifica della planarità di un grafo (implementato dal programma `planare.c`) legge in input un file che contiene le liste di adiacenza del grafo G. In questo esempio $G = K_{3,3}$, il grafo completo bipartito. Per prima cosa viene contato il numero degli archi $|E|$ di G e se $|E| > 3|V| - 6$ allora il programma si interrompe e il grafo é dichiarato non planare per il corollario 1.1. Nel caso in figura si ha $|E| = 9$ e $|V| = 6$ quindi il programma continua la verifica della planaritá.

Figura 4.2: Tramite il metodo di visita in profondità viene esplorato il grafo e gli archi vengono suddivisi in due classi: un insieme di *archi dell'albero* che definiscono *l'albero ricoprente* T di $K_{3,3}$ (le frecce continue) ed un insieme di *archi di ritorno* (u, v) tali che esiste un cammino da u a v in T. (le frecce tratteggiate). Il grafo $K_{3,3}$ partizionato in questo modo é chiamato *palm tree*.

Figura 4.3: Ordiniamo le liste di adiacenza del *palm tree* P calcolato su G in questo modo: integrando opportunamente l'algoritmo della visita in profonditá di un grafo per ogni vertice v calcoliamo i valori $\text{LOWPT}_1(v)$ e $\text{LOWPT}_2(v)$ che sono rispettivamente il primo e il secondo vertice in ordine di numerazione raggiungibile da v attraverso un arco di ritorno che esce da un discendente di v. Successivamente calcoliamo $\varphi(v, w)$ per ogni arco (v, w) in P ($\varphi(v, w)$ é definita nel paragrafo 1.3.2) e ordiniamo le liste di adiacenza di P secondo valori crescenti di φ. Riportiamo i valori LOWPT_1, LOWPT_2, φ e le liste di adiacenza ordinate secondo la funzione φ.

Figura 4.4: A questo punto generiamo i cammini eseguendo una visita in profonditá del *palm tree* P: ogni volta che attraversiamo un arco dell'albero lo aggiungiamo al cammino che stiamo costruendo e ogni volta che attraversiamo un arco di ritorno questo diventerá l'ultimo arco del nostro cammino, l'arco successivo sará il primo di un nuovo cammino. Per questo motivo ogni cammino consiste in una sequenza di archi dell'albero seguiti da un arco di ritorno.

Figura 4.5: Sia c il primo cammino che, come abbiamo visto, per il Lemma 1.8 é un ciclo; c consiste in un insieme di archi dell'albero $1 \to v_1 \to v_2 \to ... \to v_n$ seguiti da un arco di ritorno $v_n^- \to 1$. La numerazione dei vertici é tale che $1 < v_1 < ... < v_n$. Quando c viene rimosso G viene suddiviso in diverse componenti connesse, chiamate *segmenti*. Ogni segmento s consiste o in un solo arco di ritorno (v_i, w) oppure in un arco dell'albero (v_i, w) piú un sottoalbero con radice w piú tutti gli archi di ritorno che partono da questo sottoalbero.

Figura 4.6: Riportiamo il contenuto delle pile L, R e B dopo aver immerso il ciclo c ed il segmento S_1, a questo punto l'algoritmo prova ad immergere il segmento S_2 a sinistra

Figura 4.7: Dal momento che c'é una intersezione tra S_2 ed S_1 l'algoritmo immerge il segmento S_2 a destra.

Figura 4.8: Dopo aver immerso il segmento S_2 l'algoritmo prova ad immergere il segmento S_3 a sinistra ma c'é una intersezione tra S_3 ed S_1, successivamente l'algoritmo prova ad immergere S_3 a destra ma c'é una intersezione tra S_3 ed S_2 quindi il grafo é non planare.

Immersione sulla griglia bidimensionale

Figura 4.9: L'algoritmo Two Bend per l'immersione su una griglia bidimensionale di un grafo planare G (implementato dal programma `disegna.c`) legge in input un file che contiene le liste di adiacenza del grafo G opportunamente ordinate.

Figura 4.10: Tramite il metodo di visita in profondità viene esplorato il grafo; per ogni vertice v calcoliamo il valore LOWPT_1. Diamo una *orientazione bipolare* a G, ovvero assegnamo una direzione ad ogni arco di G in modo tale che il grafo risultante sia aciclico con una sola sorgente ed un solo pozzo. Per prima cosa orientiamo la sorgente e il pozzo, successivamente per orientare l'arco (u, w) consideriamo p=$\text{LOWPT}_1(w)$ e q, dove q è il successore di $\text{LOWPT}_1(w)$ nel cammino che va da p a w: se $p \to q$ allora $w \to u$, se $q \to p$ allora $u \to w$.

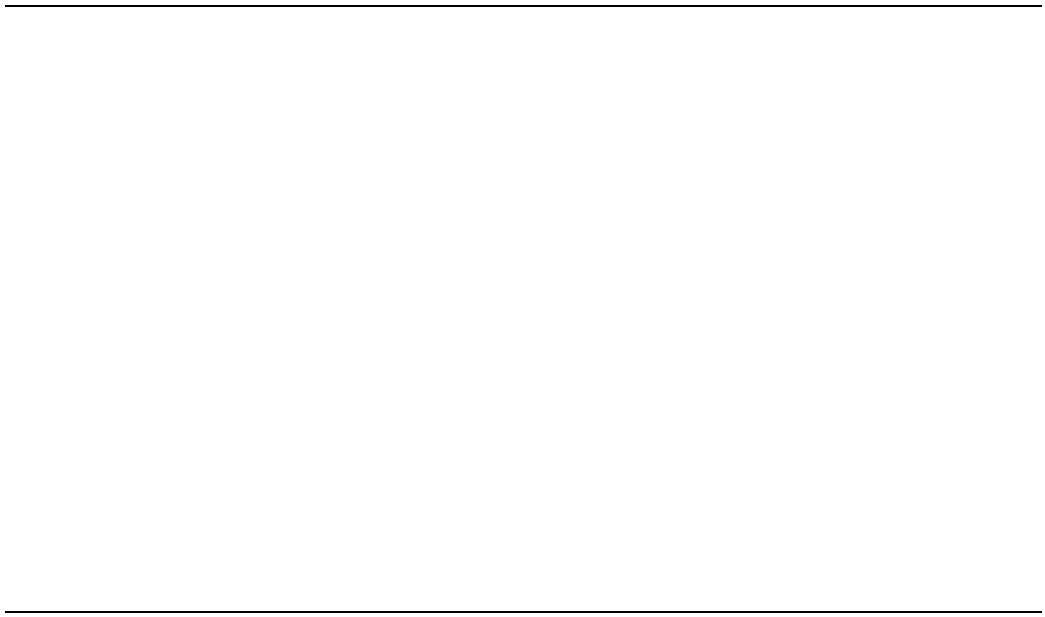

Figura 4.11: Diamo una numerazione da 1 a n ai vertici di G in modo tale che per ogni arco orientato (u, v) si ha che la numerazione di u è più bassa della numerazione di v.

Figura 4.12: Mostriamo ad ogni passo l'esecuzione dell'algoritmo TWO-BEND. Passo $i = 1$: $F_1 = \{7,2,3\}$

Figura 4.13: Passo $i = 2$: $F_2 = \{7,6,4,3,3\}$

Figura 4.14: Passo $i = 3$: $F_3 = \{7,6,4,4,5\}$

Figura 4.15: Passo $i = 4$: $F_4 = \{7,6,6,5,5\}$

Figura 4.16: Passo $i = 5$: $F_5 = \{7,6,6,6\}$

Figura 4.17: Passo $i = 6$: $F_6 = \{7,7\}$

Figura 4.18: Immersione sulla griglia bidimensionale del grafo in figura prodotta dall'algoritmo TWO-BEND.

Capitolo 5

Appendice B: Listati dei programmi

5.1 Verifica della planarità

```
/*
** planare.c
**
** Programma per la verifica della planarita' di un grafo
** G=(V,E) biconnesso.
**
** Il programma legge in input un file che contiene le liste
** di adiacenza di un grafo G.
**
** Il programma gira fino alla fine solo se il grafo e' planare
** altrimenti si interrompe e dichiara grafo non planare.
*/

#include <stdio.h>
#include <stdlib.h>

#define MAXN 100
#define MAX 50

struct nodo{
  int info;
  struct nodo *next;
```

```
};

/*
 * Questa struttura serve a memorizzare una lista di archi: se p
 * e' di tipo "struct nodo1" allora dato un arco di (u,v) avro'
 * p->info1=u, p->info2=v.
 */

struct nodo1{
  int info1;
  int info2;
  struct nodo1 *next;
};

/*
 * Questa struttura serve a memorizzare una lista di archi che
 * sono stati classificati in archi dell'albero e archi di
 * ritorno: se p e' di tipo "struct nodo2" allora dato un arco
 * di (u,v) avro' p->info1=u, p->info2=v e p->info3=0 se (u,v)
 * e' un arco di ritorno oppure p->info3=1 se (u,v) e' un arco
 * dell'albero.
 */

struct nodo2{
  int info1;
  int info2;
  int info3;
  struct nodo2 *next;
};

/*
 * leggi_grafo(l)
 *
 * legge in input il file denominato "nomefile". La prima
 * contiene il numero di vertici di G; le altre righe contengono
 * ognuna i vertici appartenenti ad una lista di adiacenza
 * separati da uno spazio, ogni lista di adiacenza termina con -1.
 * La funzione memorizza le liste di adiacenza di G in l e conta
 * il numero degli archi: se |E|>=3|N|-6 allora il grafo non e'
 * planare per il Corollario 1.1.
 */

int leggi_grafo(struct nodo **l){
```

```
  int n,a,i,cont=0;
  char nomefile[50];
  struct nodo *p;
  FILE *h;

  printf("Nome del file di dati: ");
  scanf("%s", nomefile);
  h = fopen(nomefile, "rt");
  fscanf(h, "%d", &n);
  for (i=1; i<=n; i++){
    fscanf(h, "%d", &a);
    *(l+i) = NULL;
    while (a != -1){
      cont = cont+1;
      p=malloc(sizeof(struct nodo));
      p->info=a;
      p->next=*(l+i);
      *(l+i)=p;
      fscanf(h,"%d",&a);
    }
  }
  cont=cont/2;
  if(cont>3*n-6){
    printf("\nIl grafo non e' planare\n");
    exit(1);
  }
  fclose(h);
  return(n);
}

/*
 * visita2(v,m,l,pi,number,l1,l2,arc)
 *
 * legge in input il vertice v che deve essere visitato,
 * m e' l'ultimo numero assegnato ad un vertice, l contiene
 * le liste di adiacenza di G, se i e' un vertice pi[i] e'
 * il padre di i e number[i] e' il numero che e' stato
 * assegnato ad i, low1[i] e low2[i] sono rispettivamente i
 * valori LOWPT1[i] e LOWPT2[i], in arc memorizzo gli archi
 * di G che sono stati classificati in archi dell'albero e
 * archi di ritorno. La funzione scorre la lista di adiacenza
 * di v classificando gli archi in archi di ritorno e archi
 * dell'albero, calcola LOWPT1 e LOWPT2 e assegna a v
```

```
 * il numero m che corrisponde all'ordine in cui v viene
 * visitato rispetto agli altri vertici. Se viene trovato un
 * vertice inesplorato la funzione viene richiamata in maniere
 * ricorsiva su questo vertice. In arc memorizzo la lista
 * degli archi.
 */

struct nodo2 *visita(int v,int *m,struct nodo **l,int *pi,
  int *number,int *l1,int *l2,struct nodo2 *arc){

  struct nodo *p;
  struct nodo2 *q;
  *m=*m+1;
  *(number+v)=*m;
  *(l1+v)=*m;
  *(l2+v)=*m;
  p=*(l+v);
  while(p!=NULL){
    if(*(number+p->info)==0){
      q=malloc(sizeof(struct nodo2));
      q->info1=v;
      q->info2=p->info;
      q->info3=1;
      q->next=arc;
      arc=q;
      *(pi+p->info)=v;
      arc=visita(p->info,m,l,pi,number,l1,l2,arc);
      if(*(l1+p->info)<*(l1+v)){
        *(l2+v)=min(*(l1+v),*(l2+p->info));
        *(l1+v)=*(l1+p->info);
      }else if(*(l1+p->info)==*(l1+v)){
        *(l2+v)=min(*(l2+v),*(l2+p->info));
      }else{
        *(l2+v)=min(*(l2+v),*(l1+p->info));
      }
    } else {
      if(*(number+p->info)<*(number+v) && p->info!=*(pi+v)){
        q=malloc(sizeof(struct nodo2));
        q->info1=v;
        q->info2=p->info;
        q->info3=0;
        q->next=arc;
        arc=q;
```

```
        if(*(number+p->info)<*(l1+v)){
          *(l2+v)=*(l1+v);
          *(l1+v)=*(number+p->info);
        }else if(*(number+p->info)>*(l1+v)){
          *(l2+v)=min(*(l2+v),*(number+p->info));
        }
      }
    }
    p=p->next;
  }
  return(arc);
}

/*
 * calcola(p,number,low1,low2)
 *
 * legge in input l'arco (u,v)=(p->info2,p->info3), gli array
 * number, low1 e low2 contenenti rispettivamente le informazioni
 * riguardanti il numero assegnato ad ogni vertice e il valore
 * LOWPT1 e LOWPT2. La funzione calcola il valore di funzione fi
 * definita nel paragrafo 1.3.2
 */

int calcola(struct nodo2 *p,int *number,int *low1,int *low2){
  if(p->info3==0){
    return(2* *(number+p->info2));
  }else if(p->info3==1 && *(low2+p->info2)>=*(number+p->info1)){
    return(2* *(low1+p->info2));
  }else{
    return(2* *(low1+p->info2)+1);
  }
}

/*
 * ordina(n,arc,number,l1,l2,newlist)
 *
 * legge in input il numero di vertici di G, la lista contenente
 * gli archi di G classificati in archi dell'albero e archi di
 * ritorno, se i e' un vertice number[i] e' il numero che e' stato
 * assegnato ad i, low1[i] e low2[i] sono rispettivamente i valori
 * LOWPT1[i] e LOWPT2[i]. La funzione ordina le liste di adiacenza
 * di G secondo la funzione fi e le memorizza in newlist.
 */
```

```
void ordina(int n,struct nodo2 *arc,int *number,int *l1,int *l2,
  struct nodo1 **newlist){

  int i,fi,fine=2*n+1;
  struct nodo1 *r, *ultimo[MAXN];
  struct nodo2 *bucket[MAXN],*q,*p;
  for(i=1;i<=fine;i++)
    bucket[i]=NULL;
  p=arc;
  while(p!=NULL){
    fi=calcola(p,number,l1,l2);
    q=malloc(sizeof(struct nodo2));
    q->info1=*(number+p->info1);
    q->info2=*(number+p->info2);
    q->info3=p->info3;
    q->next=bucket[fi];
    bucket[fi]=q;
    p=p->next;
  }
  for(i=1;i<=n;i++){
    *(newlist+i)=NULL;
    ultimo[i]=NULL;
  }
  for(i=1;i<=fine;i++){
    q=bucket[i];
    while(q!=NULL){
      if(*(newlist+q->info1)==NULL){
        r=malloc(sizeof(struct nodo1));
        r->info1=q->info2;
        r->info2=q->info3;
        r->next=*(newlist+q->info1);
        *(newlist+q->info1)=r;
        ultimo[q->info1]=r;
      }else{
        r=malloc(sizeof(struct nodo1));
        r->info1=q->info2;
        r->info2=q->info3;
        r->next=NULL;
        ultimo[q->info1]->next=r;
        ultimo[q->info1]=r;
      }
      q=q->next;
```

```
    }
  }
  return;
}

/*
 * dfs(n,l,newlist)
 *
 * legge in input il numero di vertici e le liste di adiacenza di G.
 * La funzione visita in profondita' il grafo G e richiama la
 * procedura che ordina le liste di adiacenza secondo la funzione
 * fi definita nel paragrafo 1.3.2.
 */

void dfs(int n,struct nodo **l,struct nodo1 **newlist){
  int number[MAXN],i,time=0,pi[MAXN],l1[MAXN],l2[MAXN];
  struct nodo2 *arc;
  arc=NULL;
  for(i=1;i<=n;i++){
    number[i]=0;
    pi[i]=-1;
  }
  for(i=1;i<=n;i++){
    if(number[i]==0){
      arc=visita(i,&time,l,&pi[0],&number[0],&l1[0],&l2[0],arc);
    }
  }
  ordina(n,arc,&number[0],&l1[0],&l2[0],newlist);
  return;
}

/*
 * parthfinder(s,v,l,cont,fre,vicino,{\sc stack},path,f,vicinob,stackb,freb)
 *
 * legge in input s che e' il vertice iniziale del cammino corrente
 * (se s=0 iniziamo un nuovo cammino), v e il vertice di cui stiamo
 * scorrendo la lista di adiacenza, cont indica il numero del
 * cammino corrente. Le pile L ed R sono memorizzate come liste
 * utilizzando gli array {\sc stack} e vicino, {\sc stack}(i) restituisce un
 * elemento della pila e vicino(i) punta all'elemento successivo
 * della stessa pila; vicino[0] punta al primo elemento di L,
 * vicino[MAX] punta al primo elemento di R. La variabile fre e'
 * l'indice della prima posizione libera di {\sc stack}. La pila B e'
```

```
 * memorizzata allo stesso modo tramite gli stackb, vicinob e freb.
 * I blocchi sono rappresentati come coppie ordinate di B: se (x,y)
 * e' un elemento di B, allora x indica l'ultimo elemento di L nel
 * blocco e y l'ultimo elemento di R nel blocco, se x=0 (y=0) il
 * blocco non ha elementi appartenenti ad L (R). Se v e' un vertice
 * path[v] e' il numero del primo cammino che contiene v. Se i e'
 * il numero di un cammino f[i] indica l'ultimo vertice del cammino.
 *
 * La funzione si interrompe se il grafo non e' planare.
 */

void pathfinder(int *s,int v,struct nodo1 **l,int *cont,int *fre,
  int *vicino,int *{\sc stack},int *path,int *f,int *vicinob,int *stackb,
  int *freb){

  struct nodo1 *p;
  int a,c,save,x,y,app,flag;
  p=*(l+v);
  while(p!=NULL){
    if(p->info2==1){
      if(*s==0){
        *s=v;
        /* inizia un nuovo cammino: v e' il primo vertice */
        (*cont)++;
      }
      *(path+p->info1)=*cont;
      /* aggiungi (v,p->info1) al cammino corrente */
      pathfinder(s,p->info1,l,cont,fre,vicino,{\sc stack},path,f,vicinob,
        stackb,freb);
      app=*vicinob;
      flag=0;
      /* scorri il blocco fino a quando ci sono elementi che */
      /* corrispondono a vertici non piu' piccoli di v e     */
      /* cancellali.                                         */
      while(app!=-1 && flag==0){
        if(((*({\sc stack}+*(stackb+app*MAXN+1))>=v) ||
          (*(stackb+app*MAXN+1)==0)) &&
          ((*({\sc stack}+*(stackb+app*MAXN+2))>=v) ||
          (*(stackb+app*MAXN+2)==0))){
          *(vicinob)=*(vicinob+*vicinob);
        }else{
          flag=1;
        }
```

```
  app=*(vicinob+app);
}
if(*vicinob!=-1 && *({\sc stack}+*(stackb+*vicinob*MAXN+1))>=v){
  *(stackb+*vicinob*MAXN+1)=0;
} else {
  if(*vicinob!=-1 && *({\sc stack}+*(stackb+*vicinob*MAXN+2))>=v){
    *(stackb+*vicinob*MAXN+2)=0;
  }
}
while(*vicino!=0 && *({\sc stack}+*vicino)>=v){
    *vicino=*(vicino+*vicino);
}
while(*(vicino+MAX)!=0 && *({\sc stack}+*(vicino+MAX))>=v){
    *(vicino+MAX)=*(vicino+*(vicino+MAX));
}
if(*(path+p->info1)!=*(path+v)){
  /* tutti i segmenti con primo arco (v,p->info1) sono  */
  /* stati immersi. I nuovi blocchi devono essere       */
  /* spostati da destra a sinistra                      */
  a=0;
  app=*vicinob;
  flag=0;
  while(app!=-1 && flag==0){
    if((*({\sc stack}+*(stackb+app*MAXN+1))>*(f+*(path+p->info1))) ||
      (*({\sc stack}+*(stackb+app*MAXN+2))>*(f+*(path+p->info1))) &&
      (*({\sc stack}+*(vicino+MAX))!=0)){
      if(*({\sc stack}+*(stackb+app*MAXN+1))>*(f+*(path+p->info1))){
        if(*({\sc stack}+*(stackb+app*MAXN+2))>*(f+*(path+p->info1))){
          printf("\nIl grafo non e' planare\n");
          exit(1);
        }
        a=*(stackb+app*MAXN+1);
      }else{
        save=*(vicino+a);
        *(vicino+a)=*(vicino+MAX);
        *(vicino+MAX)=*(vicino+*(stackb+app*MAXN+2));
        *(vicino+*(stackb+app*MAXN+2))=save;
        a=*(stackb+app*MAXN+2);
      }
      *(vicinob)=*(vicinob+*vicinob);
    }else{
      flag=1;
    }
```

```
      /* cancella l'elemento in cima a B */
      app=*(vicinob+app);
    }
    app=*vicinob;

    /* il blocco appartenente a B deve essere combinato */
    /* con i nuovi blocchi appena cancellati            */
    if(app!=-1){
      x=*(stackb+app*MAXN+1);
      y=*(stackb+app*MAXN+2);
      *(vicinob)=*(vicinob+*vicinob);
      if(x!=0){
        /* aggiungi l'elemento (x,y) a B*/
        *(stackb+*freb*MAXN+1)=x;
        *(stackb+*freb*MAXN+2)=y;
        *(vicinob+*freb)=*vicinob;
        *vicinob=*freb;
        (*freb)++;
      }else if((a!=0) || (y!=0)){
        /* aggiungi l'elemento (a,y) a B*/
        *(stackb+*freb*MAXN+1)=a;
        *(stackb+*freb*MAXN+2)=y;
        *(vicinob+*freb)=*vicinob;
        *vicinob=*freb;
        (*freb)++;
      }
    }
    /* cancella il segnale di fine pila sulla pila destra */
    *(vicino+MAX)=*(vicino+*(vicino+MAX));
  }
}else{
  if(*s==0){
    *s=v;
    /* inizia un nuovo cammino: v e' il primo vertice */
    (*cont)++;
  }
  /* aggiungi (v,p->info1) al cammino corrente */
  /* restituisci il cammino corrente */
  *(f+*cont)=p->info1;
  a=0;
  c=MAX;

  /* sposta i blocchi da sinistra a destra per immergere il */
```

```
/* cammino corrente a sinistra                                  */
while(((*(vicino+a)!=0) && (*({\sc stack}+*(vicino+a))>p->info1)) ||
  ((*(vicino+c)!=0) && (*({\sc stack}+*(vicino+c)))>p->info1)){
  app=*vicinob;
  if((*(stackb+app*MAXN+1)!=0) && (*(stackb+app*MAXN+2)!=0)){
    if(*({\sc stack}+*(vicino+a))>p->info1){
      if(*({\sc stack}+*(vicino+c))>p->info1){
        printf("\nIl grafo non e' planare\n");
        exit(1);
      }
      save=*(vicino+c);
      *(vicino+c)=*(vicino+a);
      *(vicino+a)=save;
      save=*(vicino+*(stackb+app*MAXN+1));
      *(vicino+*(stackb+app*MAXN+1))=*(vicino+*(stackb+app*MAXN+2));
      *(vicino+*(stackb+app*MAXN+2))=save;
      a=*(stackb+app*MAXN+2);
      c=*(stackb+app*MAXN+1);
    }else{
      a=*(stackb+app*MAXN+1);
      c=*(stackb+app*MAXN+2);
    }
  }else if(*(stackb+app*MAXN+1)!=0){
    save=*(vicino+*(stackb+app*MAXN+1));
    *(vicino+*(stackb+app*MAXN+1))=*(vicino+c);
    *(vicino+c)=*(vicino+a);
    *(vicino+a)=save;
    c=*(stackb+app*MAXN+1);
  }else if(*(stackb+app*MAXN+2)!=0){
    c=*(stackb+app*MAXN+2);
  }
  /* cancella l'elemento in cima a B*/
  *(vicinob)=*(vicinob+*vicinob);
}

/* aggiungi p->info1 alla pila sinistra se il cammino */
/* corrente e' normale                                */
if(*(f+*(path+*s))<p->info1){
  if(a==0){
    a=*fre;
  }
  *({\sc stack}+*fre)=p->info1;
  *(vicino+*fre)=*vicino;
```

```
        *vicino=*fre;
        (*fre)++;
      }

      /* aggiungi un nuovo blocco corrispondente ai vecchi blocchi */
      /* combinati. Il nuovo blocco potrebbe essere vuoto se il    */
      /* segmento contenente il cammino corrente non e' solo un    */
      /* arco di ritorno.                                          */
      if(c==MAX){
        c=0;
      }
      if((a!=0)||(c!=0)||(v!=*s)){
        /* aggiungi l'elemento (a,c) a B */
        *(stackb+*freb*MAXN+1)=a;
        *(stackb+*freb*MAXN+2)=c;
        *(vicinob+*freb)=*vicinob;
        *vicinob=*freb;
        (*freb)++;
      }

      /* se il segmento contenente il cammino corrente non e' solo */
      /* un arco di ritorno aggiungi un segnale di fine pila ad R. */
      if(v!=*s){
        *({\sc stack}+*fre)=0;
        *(vicino+*fre)=*(vicino+MAX);
        *(vicino+MAX)=*fre;
        (*fre)++;
      }
      *s=0;
    }
    p=p->next;
  }
  return;
}

/*
 * cammino (n,l)
 *
 * legge in input il numero di vertici e le liste di adiacenza
 * di G. La funzione inizilizza i valori che verranno utilizzati
 * in pathfinder e seleziona il vertice 1 come vertice iniziale
 * per la ricerca dei cammini.
 */
```

```
void cammino(int n,struct nodo1 **l){
    int s=0,cont=0,vicino[MAXN],{\sc stack}[MAXN],path[MAXN],fre=1;
    int f[MAXN],vicinob[MAXN],stackb[MAXN][MAXN],freb=1;
    path[1]=1;
    {\sc stack}[0]=0;
    vicino[0]=0;
    vicino[MAX]=0;
    vicinob[0]=-1;
    pathfinder(&s,1,l,&cont,&fre,&vicino[0],&{\sc stack}[0],&path[0],
      &f[0],&vicinob[0],&stackb[0][0],&freb);
    return;
}

/*
 * funzione principale
 */
int main(void){
  int n;
  struct nodo *lista[MAXN];
  struct nodo1 *newlist[MAXN];
  n=leggi_grafo(&lista[0]);
  dfs(n,&lista[0],&newlist[0]);
  cammino(n,&newlist[0]);
  return(1);
}
```

5.2 Immersione su griglie

```
# include<stdio.h>
# include<stdlib.h>
# include "calcola.c"
# define MAX 100

/*
** disegna.c
**
** Programma che costruisce un disegno rettilineo sulla griglia
** bidimensionale di un grafo planare G con n vertici.
**
** Il programma legge in input un file che contiene le liste di
```

```
** adiacenza di G.
**
** Il programma produce in output due file: uno contenente le 4
** coordinate di alcuni segmenti che costituiscono gli archi del
** grafo o parti degli archi del grafo; il secondo contenente le
** coordinate e le etichette dei vertici del grafo stesso.
*/

/*
 * visita(a,indice,u,l,n,d,pi,colore,time,low)
 *
 * legge in input il vertice u che deve essere visitato e le liste
 * di adiacenza di G. La funzione scorre la lista di adiacenza di
 * v fino a quando non trova un arco inesplorato (v,w) allora la
 * funzione viene richiamata su w.
 */

void visita(int *a,int *indice,int u,struct nodo **l,int n,int *d,
int *pi,int *colore,int *time,int *low){

   struct nodo *v;
   *(colore+u)=1;
   (*time)++;
   *(d+u)=*time;
   *(low+u)=*(d+u);
   *(low+u+MAX)=u;
   v=*(l+u);
   while(v!=NULL){
     if(*(colore+v->info)==0){
         *(pi+v->info)=u;
          visita(a,indice,v->info,l,n,d,pi,colore,time,low);
          if(*(low+u)>*(low+v->info)){
            *(low+u)=*(low+v->info);
            *(low+u+MAX)=*(low+v->info+MAX);
         }
      }else if(*(colore+v->info)==1 && (v->info)!=*(pi+u)){
         if(*(low+u)>*(d+v->info)){
            *(low+u)=*(d+v->info);
            *(low+u+MAX)=v->info;
         }
      }
      v=v->next;
   }
```

```
   *(colore+u)=2;
   *(a+*indice)=u;
   *(indice)=*(indice)-1;
   return;
}

/*
 * dfs(a,l,n,d,pi,low)
 *
 * legge in input le liste di adiacenza e il numero di vertici
 * di G. L'array a contiene i vertici di G secondo l'ordinamento
 * topologico, indice indica la prima posizione libera nell'array a,
 * se u e' un vertice allora pi[u] e' il padre di u, low[u] contiene
 * il valore lowpt1[u] e low[u][MAX]=u. La funzione visita  in
 * profondita' il grafo G e calcola l'ordinamento topologico; per ogni
 * vertice v calcola pi[v], low[v] e d[v].
 */

void dfs(int *a,struct nodo **l,int n,int *d,int *pi,int *low){
   int u,colore[MAX],time=0,indice;
   indice=n;
   for(u=1;u<=n;u++){
      colore[u]=0;
      *(pi+u)=-1;
   }
   for(u=1;u<=n;u++){
      if(colore[u]==0){
         visita(a,&indice,u,l,n,d,pi,&colore[0],&time,low);
      }
   }
   return;
}

/*
 * succ(p,w,pi)
 *
 * legge in input i vertici p,w e l'array pi. La funzione verifica
 * se p e' un antenato di w; se lo e' restituisce 1 altrimenti
 * restituisce 0.
 */

int succ(int p,int w,int *pi){
   int q,flag=0;
```

```
   while(flag==0){
      if(*(pi+*(pi+w))==p){
         q=*(pi+w);
         flag=1;
      }else{
         w=*(pi+w);
      }
   }
   return(q);
}

/*
 * elimino(l1,a,b)
 *
 * legge in input le liste di adiacenza e i vertici a e b;
 * la funzione cancella a dalla lista di b.
 */

void elimino(struct nodo **l1,int a,int b){
   int flag=0;
   struct nodo *l,*prec_v,*v;
   l=*(l1+b);
   if((l->info)==a){
      v=l->next;
      free(l);
      *(l1+b)=v;
   }else{
      v=l->next;
      prec_v=l;
      while(flag==0){
         if((v->info)==a){
            prec_v->next=v->next;
            free(v);
            v=prec_v->next;
            flag=1;
         }else{
            prec_v=v;
            v=v->next;
         }
      }
   }
   return;
}
```

```
/*
 * controlla(p,q,l2,u,w)
 *
 * legge in input le liste di adiacenza e i vertici p,q,u,w.
 * La funzione orienta l'arco (w,u) confrontandolo con l'arco (p,q):
 * se p->q (flag=1) si ha w->u; se q->p (flag=0) si ha u->w.
 * Nel primo caso  elimina w dalla lista2 di u, nel secondo elimina u
 * dalla lista2 di w.
 */

void controlla(int p,int q,struct nodo **l2,int u,int w){
   int flag=0;
   struct nodo *v;
   v=*(l2+p);
   while(flag==0 && v!=NULL){
      if((v->info)==q){
         flag=1;
      }
      v=v->next;
   }
   if(flag==0){
      elimino(l2,u,w);
   }else{
      elimino(l2,w,u);
   }
   return;
}


/*
 * numera(a,l1,l2,n,low,pi,d)
 *
 * legge in input le liste di adiacenza l1 e il numero di vertici di G.
 * l1 ed l2 contengono le liste di adiacenza di G ma la funzione
 * modifica l2 in modo tale che alla fine l2 conterra' le liste di
 * adiacenza del grafo orientato. La funzione esegue la numerazione st
 * di G, sceglie come sorgente il vertice 1 e come pozzo il secondo
 * vertice che viene esplorato nella visita in profondita', a[2].
 */

void numera(int *a,struct nodo **l1,struct nodo **l2,int n,int *low,
```

```
int *pi,int *d){

   int u,p,q,i,po;
   struct nodo *v;
   po=*(a+2);
   *(l2+po)=NULL;
   for(i=3;i<=n;i++){
      u=*(a+i);
      v=*(l1+*(a+i));
      while(v!=NULL){
         if(v->info==1){
            elimino(l2,1,u);
         }else if(*(d+(v->info))>*(d+u)){
               p=*(low+v->info+MAX);
               q=succ(p,v->info,pi);
               controlla(p,q,l2,u,v->info);
         }
         v=v->next;
      }
   }
   return;
}

/*
 * visita2(o,indi,u,l,n,colore)
 *
 * legge in input il vertice u che deve essere visitato e le liste
 * di adiacenza del grafo G orientato secondo la numerazione s-t.
 * La funzione scorre la lista di adiacenza di v fino a quando non
 * trova un arco inesplorato (u,v->info) allora la funzione viene
 * richiamata su v->info.
 */

void visita2(int *o,int *indi,int u,struct nodo **l,int n,int *colore){
   struct nodo *v;
   *(colore+u)=1;
   v=*(l+u);
   while(v!=NULL){
      if(*(colore+v->info)==0){
         visita2(o,indi,v->info,l,n,colore);
      }
      v=v->next;
   }
```

```
   *(colore+u)=2;
   *(o+*indi)=u;
   *indi=*indi-1;
   return;
}

/*
 * dfs2(o,l,n)
 *
 * legge in input le liste di adiacenza e il numero di vertici del
 * grafo orientato. La funzione calcola l'ordinamento topologico
 * dei vertici e lo memorizza nell'array o.
 */

void dfs2(int *o,struct nodo **l,int n){
   int u, colore[MAX],indi;
   indi=n;
   for(u=1;u<=n;u++){
      colore[u]=0;
   }
   for(u=1;u<=n;u++){
      if(colore[u]==0){
         visita2(o,&indi,u,l,n,&colore[0]);
      }
   }
   return;
}

/*
 * rinumero(n,posi,l1,l3,occ)
 *
 * legge in input le liste di adiacenza del grafo orientato G e
 * l'ordinamento topologico dei vertici. La funzione assegna ad
 * ogni vertice u il numero che corrisponde all'ordine in cui v
 * e' stato visitato in dfs2, le nuove liste di adiacenza vengono
 * memorizzate in l3.
 */

void rinumero(int n,int *posi,struct nodo **l2,struct nodo **l3,
int *occ){

   int i,u;
   struct nodo *p,*v;
```

```
   for(i=1;i<=n;i++){
      u=*(posi+i);
      *(occ+u)=0;
      *(l3+u)=NULL;
      v=*(l2+i);
      while(v!=NULL){
         p=malloc(sizeof(struct nodo));
         p->info=*(posi+v->info);
         v=v->next;
         p->next=*(l3+u);
         *(l3+u)=p;
         *(occ+u)=*(occ+u)+1;
      }
   }
   return;
}

/*
 * numerazione(l1,l2,l3,occ,n)
 *
 * legge in input le liste di adiacenza e il numero di vertici di G.
 * La funzione seleziona un vertice sorgente ed un vertice pozzo ed
 * impone una orientazione ad ogni arco di G, le nuove liste di
 * adiacenza vengono memorizzate in l2; successivamente i vertici
 * vengono rinumerati secondo l'orinamento topologico e questa volta
 * le nuove liste di adiacenza vengono meorizzate in l3.
 */

void numerazione(struct nodo **l1,struct nodo **l2,struct nodo **l3,
int *occ,int n){

   int i,app[MAX],d[MAX],low[MAX][MAX],ord[MAX],pi[MAX],posi[MAX];
   dfs(&app[0],l1,n,&d[0],&pi[0],&low[0][0]);
   numera(&app[0],l1,l2,n,&low[0][0],&pi[0],&d[0]);
   dfs2(&ord[0],l2,n);
   for(i=1;i<=n;i++){
         posi[ord[i]]=i;
   }
   rinumero(n,&posi[0],l2,l3,occ);
   return;
}

/*
```

```
 * leggi_grafo(l,m,deg)
 *
 * la funzione legge in input il file denominato nomefile. La prima
 * riga contiene il numero di vertici di G; le altre righe contengono
 * ognuna i vertici appartenenti ad una lista di adiacenza separati
 * da uno spazio, ogni lista di adiacenza termina con -1. La funzione
 * memorizza le liste di adiacenza di G in l ed m e restituisce
 * il numero di vertici di G.
 */

int leggi_grafo(struct nodo **l,struct nodo **m,int *deg){
   int n,a,i;
   char nomefile[50];
   struct nodo *p,*q;
   FILE *h;

   printf("Nome del file di dati: ");
   scanf("%s", nomefile);
   h=fopen(nomefile,"rt");
   fscanf(h,"%d",&n);
   for(i=1;i<=n;i++){
      fscanf(h,"%d",&a);
      *(l+i)=NULL;
      *(m+i)=NULL;
      *(deg+i)=0;
      while(a!=-1){
         p=malloc(sizeof(struct nodo));
         p->info=a;
         p->next=*(l+i);
         *(l+i)=p;
         q=malloc(sizeof(struct nodo));
         q->info=a;
         q->next=*(m+i);
         *(m+i)=q;
         fscanf(h,"%d",&a);
      }
   }
   fclose(h);
   return(n);
}

/*
 * funzione principale
```

```
*/

int main(void){
   int n,occ[MAX],deg[MAX];
   struct nodo *lista1[MAX],*lista2[MAX],*lista3[MAX];
   n=leggi_grafo(&lista1[0],&lista2[0],&deg[0]);
   numerazione(&lista1[0],&lista2[0],&lista3[0],&occ[0],n);
   disegna(&lista3[0],n,&occ[0],&deg[0]);
   return(1);
}

# include<stdio.h>
# include<stdlib.h>
# define MAX 100

/*
 * calcola.c
 */

struct nodo{
   int info;
   struct nodo *next;
};

/*
 * questa struttura serve a memorizzare una lista di vertici con le
 * loro coordinate: se p->info1=v si ha che p->info2 e p->info3 sono
 * rispettivamente l'ascissa e l'ordinata del vertice v. Uso una lista
 * con puntatore all'elemento sia precedente che successivo perche' ho
 * bisogno di percorrere la lista in entrambi i sensi.
 */

struct nodo3{
   int info1;
   int info2;
   int info3;
   struct nodo3 *next;
   struct nodo3 *prec;
};


/*
 * potenz(a)
```

```
 *
 * legge in input un numero intero a. La funzione restituisce 2^a
 */

int potenz(int a){
   int i,r=1;
   if(a==0){
      return(r);
   }else{
      for(i=1;i<=a;i++){
         r=r*2;
      }
   }
   return(r);
}

/*
 * stampa_matrice(a,n,h)
 * legge in input una matrice a ed un file h. La funzione memorizza
 * nel file h le informazioni contenute dalla matrice a.
 */

void stampa_matrice(int *a, int n, FILE *h){
   int j;
   for(j=1;j<=n;j++){
      fprintf(h,"%d %d %d\n",j,*(a+j),*(a+MAX+j));
   }
   return;
}

/* crea_lista1(f,l,i,m,po,punt,deg,altezza,coord,h)
 *
 * legge in input la frontiera corrente f, le liste di adiacenza di G,
 * il vertice i che deve essere immerso ed ha un arco uscente.
 * La funzione crea una lista m che contiene la lista di adiacenza di i,
 * definisce in modo opportuno le coordinate dei vertici adiacenti ad i
 * come mostrato in figura 2.6 (a) e le memorizza in f. Il primo e
 * l'ultimo elemento di m puntano rispettivamente all'elemento che
 * precede e all'elemento che segue i nella lista f. L'array dei
 * puntatori ai vertici e quello contenente informazioni riguardo al
 * numero di archi entranti nel vertice i vengono aggiornati.
 * La funzione memorizza nel file h le coordinate dei segmenti
 * corrispondenti agli archi che stiamo immergendo e restituisce il
```

```
 * puntatore al primo elemento di m.
*/

struct nodo3 *crea_lista1(struct nodo3 *f,struct nodo **l,int i,
struct nodo3 *m, int po,struct nodo3 **punt,int *deg,int *altezza,
int *coord,FILE *h){

   struct nodo *p;
   struct nodo3 *G,*e;
   p=*(l+i);
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   if(*(punt+i)!=NULL){
      e=*(punt+i);
      G->info2=e->info2;
      e=e->next;
      G->next=e;
      if(e!=NULL){
         e->prec=G;
      }
   }else{
      G->next=NULL;
   }
   G->info3=*altezza;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   e=*(punt+i);
   if(e!=NULL){
      G->prec=e->prec;
   }else{
      G->prec=NULL;
   }
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   return(m);
}

/* crea_lista2(f,l,i,m,po,punt,deg,altezza,coord,h)
 *
 * legge in input la frontiera corrente f, le liste di adiacenza di G,
 * il vertice i che deve essere immerso ed ha due archi uscenti.
 * La funzione crea una lista m che contiene la lista di adiacenza di i,
```

```
 * definisce in modo opportuno le coordinate dei vertici adiacenti ad i
 * come mostrato in figura 2.6 (a) e le memorizza in f. Il primo e
 * l'ultimo elemento di m puntano rispettivamente all'elemento che
 * precede e all'elemento che segue i nella lista f. L'array dei
 * puntatori ai vertici e quello contenente informazioni riguardo al
 * numero di archi entranti nel vertice i vengono aggiornati. Le
 * variabili x e y sono rispettivamente le linee di supporto verticale
 * piu' vicine a sinistra e a destra del vertice p->info. La linea di
 * supporto verticale passante per p->info e' equidistante da queste due
 * linee x ed y, se una delle due linee non c'e' allora la funzione
 * posiziona p->info a distanza 1 dall'altra. La funzione memorizza nel
 * file h le coordinate dei segmenti corrispondenti agli archi che
 * stiamo immergendo e restituisce il puntatore al primo elemento di m.
 */

struct nodo3 *crea_lista2(struct nodo3 *f,struct nodo **l,int i,
struct nodo3 *m, int po,struct nodo3 **punt,int *deg,int *altezza,
int *coord,FILE *h){

   struct nodo *p;
   struct nodo3 *G,*e;
   int x,y;
   p=*(l+i);
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   if(*(punt+i)!=NULL){
      e=*(punt+i);
      G->next=e->next;
      x=e->info2;
      e=e->next;
      if(e!=NULL){
         y=e->info2;
      }else{
         y=0;
      }
   }else{
      x=0;
      G->next=NULL;
   }
   if(i==1){
      G->info2=*(coord+i)+po;
   }else if(x==0){
      G->info2=y-4;
```

```
   }else if(y==0){
      G->info2=x+4;
   }else{
      G->info2=(x+y)/2;
   }
   G->info3=*altezza;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,*(coord+MAX+i));
   fprintf(h,"\n%d %d",G->info2,*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   G->prec=NULL;
   if(m!=NULL){
      m->prec=G;
   }
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   p=p->next;
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   G->next=m;
   e=*(punt+i);
   if(e!=NULL){
      G->prec=e->prec;
   }else{
      G->prec=NULL;
   }
   if(m!=NULL){
      m->prec=G;
   }
   G->info2=*(coord+i);
   G->info3=*altezza;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   return(m);
}

/* crea_lista3(f,l,i,m,po,punt,deg,altezza,coord,h)
 *
 * legge in input la frontiera corrente f, le liste di adiacenza di G,
```

```
 * il vertice i che deve essere immerso ed ha tre archi uscenti.
 * La funzione crea una lista m che contiene la lista di adiacenza di i,
 * definisce in modo opportuno le coordinate dei vertici adiacenti ad i
 * come mostrato in figura 2.6 (a) e le memorizza in f. Il primo e
 * l'ultimo elemento di m puntano rispettivamente all'elemento che
 * precede e all'elemento che segue i nella lista f. L'array dei
 * puntatori ai vertici e quello contenente informazioni riguardo al
 * numero di archi entranti nel vertice i vengono aggiornati. Le
 * variabili x e y sono le linee di supporto verticali piu' vicine al
 * vertice p->info: quando immergo il primo vertice della lista di
 * adiacenza di i y e' la linea di supporto piu' vicina a sinistra di
 * p->info e x e' la linea di supporto piu' vicina a destra di p->info,
 * quando inserisco il terzo vertice della lista di adiacenza di i ho
 * il contrario. La linea di supporto verticale passante per p->info e'
 * equidistante da queste due linee x ed y, se una delle due linee non
 * c'e' allora la funzione posiziona p->info a distanza 1 dall'altra.
 * La funzione memorizza nel file h le coordinate dei segmenti
 * corrispondenti agli archi che stiamo immergendo e restituisce il
 * puntatore al primo elemento di m.
*/

struct nodo3 *crea_lista3(struct nodo3 *f,struct nodo **l,int i,
struct nodo3 *m,int po,struct nodo3 **punt,int *deg,int *altezza,
int *coord,FILE *h){

   struct nodo *p;
   struct nodo3 *G,*e;
   int x,y;
   p=*(l+i);
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   if(*(punt+i)!=NULL){
      e=*(punt+i);
      G->next=e->next;
      x=e->info2;
      e=e->next;
      if(e!=NULL){
         y=e->info2;
      }else{
         y=0;
      }
   }else{
      G->next=NULL;
```

```
}
if(i==1){
   G->info2=*(coord+i)+po;
}else if(x==0){
   G->info2=y-4;
}else if(y==0){
   G->info2=x+4;
}else{
   G->info2=(x+y)/2;
}
G->info3=*altezza;
e=*(punt+i);
if(e!=NULL){
   e=e->prec;
   if(e!=NULL){
      y=e->info2;
   }else{
      y=0;
   }
}else{
   y=0;
}
fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
fprintf(h," %d %d",G->info2,*(coord+MAX+i));
fprintf(h,"\n%d %d",G->info2,*(coord+MAX+i));
fprintf(h," %d %d",G->info2,G->info3);
G->prec=NULL;
if(m!=NULL){
   m->prec=G;
}
*(deg+p->info)=*(deg+p->info)+1;
*(punt+p->info)=G;
m=G;
p=p->next;
G=malloc(sizeof(struct nodo3));
G->info1=p->info;
G->next=m;
G->prec=NULL;
if(m!=NULL){
   m->prec=G;
}
G->info2=*(coord+i);
G->info3=*altezza;
```

```
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   p=p->next;
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   G->next=m;
   e=*(punt+i);
   if(e!=NULL){
      G->prec=e->prec;
   }else{
      G->prec=NULL;
   }
   if(m!=NULL){
      m->prec=G;
   }
   if(i==1){
      G->info2=*(coord+i)-po;
   }else if(x==0){
      G->info2=y+4;
   }else if(y==0){
      G->info2=x-4;
   }else{
      G->info2=(x+y)/2;
   }
   G->info3=*altezza;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,*(coord+MAX+i));
   fprintf(h,"\n%d %d",G->info2,*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   return(m);
}

/* crea_lista4(f,l,i,m,po,punt,deg,altezza,coord,h)
 *
 * legge in input la frontiera corrente f, le liste di adiacenza di G,
 * il vertice i che deve essere immerso ed ha quattro archi uscenti.
 * La funzione aggiorna la frontiera f inserendo al posto di i la sua
```

```
 * lista di e definendo le coordinate dei vertici in modo opportuno
 * come mostrato in figura 2.6 (a). Osserviamo che l'unico vertice
 * che puo' avere quattro archi uscenti e' la sorgente 1 e quando la
 * funzione inserisce la lista di adiacenza di 1 nella frontiera la
 * frontiera e' vuota. L'array dei puntatori ai vertici e quello
 * contenente informazioni riguardo al numero di archi entranti nel
 * vertice i vengono aggiornati. La funzione memorizza nel file h le
 * coordinate dei segmenti corrispondenti agli archi che stiamo
 * immergendo e restituisce il puntatore al primo elemento di m.
 */

struct nodo3 *crea_lista4(struct nodo3 *f,struct nodo **l,int i,
struct nodo3 *m,int po,struct nodo3 **punt,int *deg,int *altezza,
int *coord,FILE *h){

   struct nodo *p;
   struct nodo3 *G;
   p=*(l+i);
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   G->info2=*(coord+i)+po;
   G->info3=*altezza;
   G->next=NULL;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",*(coord+i),*(coord+MAX+i)-1);
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i)-1);
   fprintf(h," %d %d",G->info2,*(coord+MAX+i)-1);
   fprintf(h,"\n%d %d",G->info2,*(coord+MAX+i)-1);
   fprintf(h," %d %d",G->info2,G->info3);
   G->prec=NULL;
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   p=p->next;
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   G->next=m;
   G->prec=NULL;
   if(m!=NULL){
      m->prec=G;
   }
   G->info2=*(coord+i)+po/2;
   G->info3=*altezza;
```

```
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,*(coord+MAX+i));
   fprintf(h,"\n%d %d",G->info2,*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   p=p->next;
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   G->next=m;
   G->prec=NULL;
   if(m!=NULL){
      m->prec=G;
   }
   G->info2=*(coord+i);
   G->info3=*altezza;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   p=p->next;
   G=malloc(sizeof(struct nodo3));
   G->info1=p->info;
   G->next=m;
   G->prec=NULL;
   if(m!=NULL){
      m->prec=G;
   }
   G->info2=*(coord+i)-po;
   G->info3=*altezza;
   fprintf(h,"\n%d %d",*(coord+i),*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,*(coord+MAX+i));
   fprintf(h,"\n%d %d",G->info2,*(coord+MAX+i));
   fprintf(h," %d %d",G->info2,G->info3);
   *(deg+p->info)=*(deg+p->info)+1;
   *(punt+p->info)=G;
   m=G;
   return(m);
}

/*
```

```
 * occ1(f,po,l,i,punt,deg,altezza,coord,h)
 *
 * legge in input la frontiera corrente f, le liste di adiacenza di G
 * ed il vertice i che deve essere immerso. In questo caso i ha un arco
 * uscente e la funzione costruisce una lista m contenente la lista di
 * adiacenza di i e la inserisce nella frontiera al posto di i. La
 * funzione restituisce il puntatore al primo elemento di f.
 */

struct nodo3 *occ1(struct nodo3 *f,int po,struct nodo **l,int i,
struct nodo3 **punt,int *deg,int *altezza,int *coord,FILE *h){

   struct nodo3 *q,*m;
   m=NULL;
   m=crea_lista1(f,l,i,m,po,punt,deg,altezza,coord,h);
   if(f==NULL){
      f=m;
   }else{
      q=*(punt+i);
      q=q->prec;
      if(q!=NULL){
         q->next=m;
      }else{
         f=m;
      }

   }
   return(f);
}

/*
 * occ2(f,po,l,i,punt,deg,altezza,coord,h)
 *
 * la funzione occ2 e' simile alla funzione occ1, l'unica differenza
 * e' che in questo caso il vertice i ha due archi uscenti.
 */

struct nodo3 *occ2(struct nodo3 *f,int po,struct nodo **l,int i,
struct nodo3 **punt,int *deg,int *altezza,int *coord,FILE *h){

   struct nodo3 *G,*q,*m;
   m=NULL;
   m=crea_lista2(f,l,i,m,po,punt,deg,altezza,coord,h);
```

```
   G=m;
   if(f==NULL){
      f=m;
   }else{
      q=*(punt+i);
      q=q->prec;
      if(q!=NULL){
         q->next=m;
      }else{
         f=m;
      }
   }
   return(f);
}

/*
 * occ3(f,po,l,i,punt,deg,altezza,coord,h)
 *
 * la funzione occ3 e' simile alla funzione occ1, l'unica differenza
 * e' che in questo caso il vertice i ha tre archi uscenti.
 */

struct nodo3 *occ3(struct nodo3 *f,int po,struct nodo **l,int i,
struct nodo3 **punt,int *deg,int *altezza,int *coord,FILE *h){

   struct nodo3 *q,*m,*G;
   m=NULL;
   m=crea_lista3(f,l,i,m,po,punt,deg,altezza,coord,h);
   G=m;
   if(f==NULL){
      f=m;
   }else{
      q=*(punt+i);
      q=q->prec;
      if(q!=NULL){
         q->next=m;
      }else{
         f=m;
      }
   }
   return(f);
}
```

```
/*
 * occ4(f,po,l,i,punt,deg,altezza,coord,h)
 *
 * la funzione occ4 e' simile alla funzione occ1, l'unica differenza
 * e' che in questo caso il vertice i ha quattro archi uscenti.
 */


struct nodo3 *occ4(struct nodo3 *f,int po,struct nodo **l,int i,
struct nodo3 **punt,int *deg,int *altezza,int *coord,FILE *h){

   f=crea_lista4(f,l,i,f,po,punt,deg,altezza,coord,h);
   return(f);
}

/*
 * deg1(f,punt,i,coord,h)
 *
 * legge in input la frontiera corrente f, l'array dei puntatori ed
 * il vertice che deve essere immerso i che ha un arco entrante.
 * La funzione calcola le coordinate dei segmenti che rappresentano
 * gli archi entranti in i come mostrato in figura 2.6 (b) e le
 * memorizza nel file h. La funzione restituisce il puntatore
 * al primo elemento di f.
*/

struct nodo3 *deg1(struct nodo3 *f,struct nodo3 **punt, int i,
int *coord,FILE *h){

   struct nodo3 *p;
   p=*(punt+i);
   *(coord+i)=p->info2;
   *(coord+MAX+i)=p->info3;
   return(f);
}

/*
 * elimina_succ(f,punt,i,h)
 *
 * legge in input la frontiera corrente f, l'array dei puntatori e il
 * vertice che deve essere immerso i. La funzione elimina l'occorrenza
 * di i in f successiva a punt[i], calcola le ccordinate dei segmenti
 * che rappresentano gli archi entranti in i come mostrato in
```

```
 * figura 2.6 (b) e le memorizza nel file h. La funzione restituisce
 * il puntatore al primo elemento di f.
 */

struct nodo3 *elimina_succ(struct nodo3 *f,struct nodo3 **punt, int i,
FILE *h){

   struct nodo3 *p,*q;
   p=*(punt+i);
   q=p->next;
   fprintf(h,"\n%d %d",p->info2,p->info3);
   fprintf(h," %d %d",q->info2,p->info3);
   fprintf(h,"\n%d %d",q->info2,q->info3);
   fprintf(h," %d %d",q->info2,p->info3);
   p->next=q->next;
   free(q);
   q=p->next;
   if(q!=NULL){
      q->prec=*(punt+i);
   }
   return(f);
}

/*
 * elimina_prec(f,punt,i,h)
 *
 * legge in input la frontiera corrente f, l'array dei puntatori e il
 * vertice che deve essere immerso i. La funzione elimina l'occorrenza
 * di i in f precedente a punt[i], calcola le coordinate dei segmenti
 * che rappresentano gli archi entranti in i come mostrato in
 * figura 2.6 (b) e le memorizza nel file h. La funzione restituisce
 * il puntatore al primo elemento di f.
 */

struct nodo3 *elimina_prec(struct nodo3 *f,struct nodo3 **punt,
int i,FILE *h){

   struct nodo3 *p,*q;
   p=*(punt+i);
   if(f->info1==i){
      q=p->prec;
      fprintf(h,"\n%d %d",p->info2,p->info3);
      fprintf(h," %d %d",q->info2,p->info3);
```

```
      fprintf(h,"\n%d %d",q->info2,q->info3);
      fprintf(h," %d %d",q->info2,p->info3);
      free(q);
      f=p;
   }else{
      q=p->prec;
      fprintf(h,"\n%d %d",p->info2,p->info3);
      fprintf(h," %d %d",q->info2,p->info3);
      fprintf(h,"\n%d %d",q->info2,q->info3);
      fprintf(h," %d %d",q->info2,p->info3);
      p->prec=q->prec;
      free(q);
      q=p->prec;
      q->next=p;
      q=q->next;
   }
   return(f);
}

/*
 * deg2(f,punt,i,coord,h)
 *
 * legge in input la frontiera corrente f, l'array dei puntatori ed il
 * vertice che deve essere immerso i che ha due archi entranti. La
 * funzione  elimina una delle due occorrenze di i nella frontiera f e
 * memorizza le coordinate di i in coord. La funzione restituisce il
 * puntatore al primo elemento di f.
*/

struct nodo3 *deg2(struct nodo3 *f,struct nodo3 **punt, int i,
int *coord,FILE *h){

   struct nodo3 *q,*p;
   p=*(punt+i);
   q=p->next;
   if( q!=NULL && q->info1==i){
      /* l'occorrenza di i in f */
      /* e' successiva a punt[i] */
      *(coord+i)=q->info2;
      *(coord+MAX+i)=p->info3;
      f=elimina_succ(f,punt,i,h);
   }else{
      /* l'occorrenza di i in f */
```

```
      /* e' precedente a punt[i] */
      *(coord+i)=p->info2;
      *(coord+MAX+i)=p->info3;
      f=elimina_prec(f,punt,i,h);
   }
   p=*(punt+i);
   p->info2=*(coord+i);
   return(f);
}

/*
 * deg3(f,punt,i,coord,h)
 *
 * legge in input la frontiera corrente f, l'array dei puntatori ed
 * il vertice che deve essere immerso i che ha tre archi entranti.
 * La funzione  elimina due delle tre occorrenze di i nella frontiera
 * f e memorizza le coordinate di i in coord. La funzione restituisce
 * il puntatore al primo elemento di f.
*/

struct nodo3 *deg3(struct nodo3 *f,struct nodo3 **punt, int i,
int *coord,FILE *h){

   struct nodo3 *p,*q;
   int x;
   p=*(punt+i);
   q=p->next;
   if( q!=NULL && q->info1==i){
      x=q->info2;
      f=elimina_succ(f,punt,i,h);
      q=p->next;
      if( q!=NULL && q->info1==i){
         /* le due occorrenze di i in f */
         /* sono successive a punt[i] */
         *(coord+i)=x;
         *(coord+MAX+i)=p->info3;
         f=elimina_succ(f,punt,i,h);
      }else{
         /* le due occorrenze di i in f */
         /* sono una precedente  e */
          /* una successiva a punt[i] */
         *(coord+i)=p->info2;
         *(coord+MAX+i)=p->info3;
```

```
         f=elimina_prec(f,punt,i,h);
      }
   }else{
      /* le due occorrenze di i in f */
      /* sono precedenti a punt[i] */
      q=p->prec;
      *(coord+i)=q->info2;
      *(coord+MAX+i)=p->info3;
      f=elimina_prec(f,punt,i,h);
      f=elimina_prec(f,punt,i,h);
   }
   p=*(punt+i);
   p->info2=*(coord+i);
   return(f);
}

/*
 * deg4(f,punt,i,coord,h)
 *
 * legge in input la frontiera corrente f, l'array dei puntatori ed
 * il vertice che deve essere immerso i che ha due archi entranti.
 * La funzione  elimina tre delle quattro occorrenze di i nella
 * frontiera f e memorizza le coordinate di i in coord. La funzione
 * restituisce il puntatore al primo elemento di f.
*/

struct nodo3 *deg4(struct nodo3 *f,struct nodo3 **punt, int i,
int *coord,FILE *h){

   int hh;
   struct nodo3 *p;
   p=*(punt+i);
   hh=p->info3;
   p=f;
   p=(p->next)->next;
   *(coord+i)=p->info2;
   *(coord+MAX+i)=hh;
   p=f;
   fprintf(h,"\n%d %d",p->info2,p->info3);
   fprintf(h," %d %d",p->info2,hh+1);
   fprintf(h,"\n%d %d",p->info2,hh+1);
   fprintf(h," %d %d",*(coord+i),hh+1);
   fprintf(h,"\n%d %d",*(coord+i),hh+1);
```

```
   fprintf(h," %d %d",*(coord+i),hh);
   p=p->next;
   fprintf(h,"\n%d %d",p->info2,p->info3);
   fprintf(h," %d %d",p->info2,hh);
   fprintf(h,"\n%d %d",p->info2,hh);
   fprintf(h," %d %d",*(coord+i),hh);
   p=p->next;
   fprintf(h,"\n%d %d",p->info2,p->info3);
   fprintf(h," %d %d",p->info2,hh);
   p=p->next;
   fprintf(h,"\n%d %d",*(coord+i),hh);
   fprintf(h," %d %d",p->info2,hh);
   fprintf(h,"\n%d %d",p->info2,p->info3);
   fprintf(h," %d %d",p->info2,hh);
   return(f);
}

/*
 * disegna(l,n,occ,deg)
 *
 * legge in input le liste di adiacenza di G opportunamente ordinate.
 * Se i e' un vertice deg[i] e' il numero di archi entranti in i, occ[i]
 * il numero degli archi uscenti da i; po=2^(n-1) definisce i segmenti
 * orizzontali dei semi-archi uscenti dal vertice i=1; altezza e'
 * l'ordinata del vertice corrente; coord contiene le informazioni
 * relative alle coordinate dei vertici: se i e' un vertice allora si
 * ha che coord[1][v] e coord[2][v] sono rispettivamente l'ascissa e
 * l'ordinata di v; f e' una lista che contiene i vertici appartenenti
 * alla frontiera; punt e' un array di puntatori ai vertici che
 * appartengono ad f: se i e' un vertice punt[i] punta ad i nella lista f.
 * La funzione calcola le coordinate dei segmenti che rappresentano archi
 * o parti di archi di G e le coordinate dei vertici e li memorizza
 * rispettivamente nel file nomefile1 e nel file nomefile2.
 */

void disegna(struct nodo **l,int n,int *occ,int *deg){
   int i,po=potenz(n-1),altezza=1,coord[2][MAX];
   struct nodo3 *f,*punt[MAX];
   FILE *h,*h2;
   char nomefile1[50], nomefile2[50];
   coord[0][1]=potenz(n-2);
   coord[1][1]=1;
   f=NULL;
```

```
for(i=1;i<=n;i++){
   punt[i]=NULL;
}
printf("Nome del file dove vuoi memorizzare le coordinate dei segmenti: ");
scanf("%s", nomefile1);
printf("Nome del file dove vuoi memorizzare le coordinate dei vertici: ");
scanf("%s", nomefile2);
h=fopen(nomefile1,"wt");
h2=fopen(nomefile2,"wt");
for(i=1;i<=n;i++){
   /* immergo tutti i semi-archi entranti nel vertice v=i secondo */
   /* il valore deg[v] come mostrato in figura 2.6 (a) */
   switch(*(deg+i)){
      case(1):
         f=deg1(f,punt,i,&coord[0][0],h);
         break;
      case(2):
         f=deg2(f,punt,i,&coord[0][0],h);
         break;
      case(3):
         f=deg3(f,punt,i,&coord[0][0],h);
         break;
      case(4):
         f=deg4(f,punt,i,&coord[0][0],h);
         break;
      default:
      break;
   }
   altezza=altezza+1;
   /* immergo tutti i semi-archi uscenti dall vertice v=i secondo */
   /* il valore occ[v] come mostrato in figura 2.6 (b) */
   switch(*(occ+i)){
   case(1):
      f=occ1(f,po,l,i,&punt[0],deg,&altezza,&coord[0][0],h);
      break;
   case(2):
      f=occ2(f,po,l,i,&punt[0],deg,&altezza,&coord[0][0],h);
      break;
   case(3):
      f=occ3(f,po,l,i,&punt[0],deg,&altezza,&coord[0][0],h);
      break;
   case(4):
      f=occ4(f,po,l,i,&punt[0],deg,&altezza,&coord[0][0],h);
```

```
         break;
      default:
         break;
      }
   }
   stampa_matrice(&coord[0][0],n,h2);
   return;
}

/*
**   grafotex.c
**
**   Programma per il tracciamento di un grafo (planare ed immerso
**   su una griglia) mediante le istruzioni LaTeX.
**
**   Il programma legge in input il contenuto di due file:
**   a) il primo contiene le 4 coordinate di alcuni segmenti che
**   costituiscono   gli archi del grafo o parti degli archi del grafo;
**   b) il secondo contiene le coordinate e le etichette dei
**   vertici del grafo stesso.
**
**   Il programma produce in output un file in formato LaTeX con
**   una figura con il grafo.
**
*/

#include <stdlib.h>
#include <stdio.h>

#define MAX_STRINGA 255
#define MAX_N 100
#define MAX_M 300
#define MAX_X 150
#define MAX_Y 200
#define UNITA 0.65

struct segmento {
   int x1, y1, x2, y2;
   struct segmento *next;
};

struct vertice {
   int x, y, label;
```

```
   struct vertice *next;
};

/*
 *   leggi_archi(nome_file)
 *
 *   legge in input il file denominato nome_file. Ogni riga del
 *   file contiene quattro numeri interi separati da spazi.
 *   I quattro numeri rappresentano le coordinate di un segmento.
 *   La funzione memorizza tali coordinate in una lista di strutture
 *   di tipo "segmento" e restituisce il puntatore al primo
 *   elemento.
 */

struct segmento *leggi_archi(char nome_file[]) {
   FILE *f;
   int x1, y1, x2, y2;
   struct segmento *s, *primo=NULL;

   if (f = fopen(nome_file, "rt")) {
      while (!feof(f)) {
         fscanf(f, "%d %d %d %d", &x1, &y1, &x2, &y2);
         s = malloc(sizeof(struct segmento));
         s->next = primo;
         s->x1 = x1;
         s->y1 = y1;
         s->x2 = x2;
         s->y2 = y2;
         primo = s;
      }
      fclose(f);
   }
   return(primo);
}

/*
 *   leggi_vertici(nome_file)
 *
 *   legge in input il file denominato nome_file. Ogni riga del file
 *   contiene tre numeri interi che rappresentano rispettivamente
 *   l'etichetta di un vertice del grafo, la sua coordinata x e la
 *   sua coordinata y. La funzione restituisce il puntatore
 *   al primo elemento della lista di strutture di tipo "vertice"
```

```
 *   in cui sono state memorizzate le etichette e le coordinate dei
 *   vertici.
 */

struct vertice *leggi_vertici(char nome_file[]) {
   FILE *f;
   int l, x, y;
   struct vertice *primo=NULL, *v;

   if (f = fopen(nome_file, "rt")) {
      while (!feof(f)) {
         fscanf(f, "%d %d %d", &l, &x, &y);
         v = malloc(sizeof(struct vertice));
         v->label = l;
         v->x = x;
         v->y = y;
         v->next = primo;
         primo = v;
      }
      fclose(f);
   }
   return(primo);
}

/*
 *   crea_grafo(a, v, nome_file)
 *
 *   Costruisce un documento LaTeX (sul file nome_file) con una
 *   figura in cui sono rappresentati i segmenti presenti nella
 *   lista puntata da a ed i vertici presenti nella lista puntata
 *   da v.
 */

int crea_grafo(struct segmento *a, struct vertice *v, char nome_file[]) {
   int rc = 1, max_x=0, max_y=0, cx1, cx2, cy1, cy2;
   FILE *f;
   struct segmento *pa;


   if (f = fopen(nome_file, "wt")) {
      fprintf(f, "\\documentclass[a4]{report}\n");
      fprintf(f, "\\begin{document}\n");
      fprintf(f, "\\begin{figure}[h]\n");
```

```
fprintf(f, "\\unitlength=%fmm\n", UNITA);
fprintf(f, "\\begin{center}\n");
fprintf(f, "\\begin{picture}(%d,%d)\n", MAX_X, MAX_Y);
pa = a;
while (pa != NULL) {
   if (pa->x1 > max_x)
      max_x = pa->x1;
   if (pa->x2 > max_x)
      max_x = pa->x2;
   if (pa->y1 > max_y)
      max_y = pa->y1;
   if (pa->y2 > max_y)
      max_y = pa->y2;
   pa = pa->next;
}
while (a != NULL) {
   cx1 = (a->x1 * MAX_X)/max_x;
   cx2 = (a->x2 * MAX_X)/max_x;
   cy1 = (a->y1 * MAX_Y)/max_y;
   cy2 = (a->y2 * MAX_Y)/max_y;

   /* linea verticale verso l'alto */
   if (a->y1 < a->y2 && a->x1 == a->x2) {
      fprintf(f, "\\put(%d,%d){\\line(0,1){%d}}\n", cx1, cy1, cy2-cy1);
   }
   /*   linea verticale verso il basso */
   if (a->y1 > a->y2 && a->x1 == a->x2) {
      fprintf(f, "\\put(%d,%d){\\line(0,-1){%d}}\n", cx1, cy1, cy1-cy2);
   }
   /*   linea orizzontale verso destra */
   if (a->x1 < a->x2 && a->y1 == a->y2) {
      fprintf(f, "\\put(%d,%d){\\line(1,0){%d}}\n", cx1, cy1, cx2-cx1);
   }
   /*   linea orizzontale verso sinistra */
   if (a->x1 > a->x2 && a->y1 == a->y2) {
      fprintf(f, "\\put(%d,%d){\\line(-1,0){%d}}\n", cx1, cy1, cx1-cx2);
   }
   a = a->next;
}
while (v != NULL) {
   cx1 = (v->x * MAX_X)/max_x;
   cy1 = (v->y * MAX_Y)/max_y;
   fprintf(f, "\\put(%d,%d){\\circle*{1.0}}\n\\put(%d,%d){\\small %d}\n", cx1, cy1,
```

```
         cx1+1, cy1+1, v->label);
         v = v->next;
      }
      fprintf(f, "\\end{picture}\n\\end{center}\n\\end{figure}\n\n");
      fprintf(f, "\\end{document}\n");
      fclose(f);
   } else {
      rc = 0;
   }
   return(rc);
}

/*
 *   funzione principale
 */

int main(void) {
   char file1[MAX_STRINGA], file2[MAX_STRINGA], fileout[MAX_STRINGA];
   struct segmento *primo_arco;
   struct vertice *primo_vertice;

   fprintf(stderr, "Nome del file con le coordinate dei segmenti: ");
   scanf("%s", file1);
   fprintf(stderr, "Nome del file con le coordinate dei vertici:  ");
   scanf("%s", file2);
   fprintf(stderr, "Nome del file di output in formato LaTeX:     ");
   scanf("%s", fileout);

   if (!(primo_arco = leggi_archi(file1))) {
      fprintf(stderr, "Errore nella lettura del file '%s'.\n", file1);
      fflush(stderr);
      return(0);
   }
   if (!(primo_vertice = leggi_vertici(file2))) {
      fprintf(stderr, "Errore nella lettura del file '%s'.\n", file2);
      fflush(stderr);
      return(0);
   }

   if (!crea_grafo(primo_arco, primo_vertice, fileout)) {
      fprintf(stderr, "Errore nella creazione del grafo (file '%s').\n", fileout);
      fflush(stderr);
      return(0);
```

```
    }
    fprintf(stderr, "Il grafo e' stato creato con successo (file '%s').\n", fileout);
    fflush(stderr);
    return(1);
}
```

Bibliografia

[1] A. Aggarwal, M. Klawe, D. Lichtenstein, N. Linial, A. Wigderson, *Multi-layer grid embedding*, Proc. IEEE 26th Ann. Symp. Foundations of Computer Science, 1985, 186-196.

[2] A. Aggarwal, M. Klawe, D. Lichtenstein, N. Linial, A. Wigderson, *A Lower Bound on the Area of Permutation Layouts*, Algorithmica 7 (1992) 133-145.

[3] A. Aggarwal, M. Klawe, P. Shor, *Multi-layer grid embedding for VLSI*, Algorithmica 6 (1991) 129-151.

[4] L. Auslander, S. V. Parter, *On embedding graphs in the plane*, J. Math. and Mech. 10, 3 (May 1961), 517-523.

[5] N. Chiba, T. Nishizeki, *Planar Graphs: Theory and Algorithms*, North-Holland, Amsterdam, 1988.

[6] D. Dolev, F. T. Leighton, H. Trickey, *Planar Embedding of Planar Graphs*, Advances in Computing Research, Vol. 2, JAI Press, Greenwich, CT, 1984, 147-161.

[7] J. Ebert, *st-numbering the vertices of biconnected graphs*, Computing 30 (1983) 19-33.

[8] S. Even, *Graphs Algorithms*, Computer science Press, Potomac, 1979.

[9] S. Even, R. A. Tarjan, *Computing and st-numbering*, Theoret. Comput. Sci. 2 (1976) 339-344.

[10] D. W. Hall, G. Spencer, *Elementary topology*, Wiley,New York, 1955.

[11] J.Hopcroft, R. A. Tarjan, *Efficient planarity testing*, J. Comput. Math. 21 (1974) 549-568.

[12] J.Hopcroft, R. A. Tarjan, *Dividing a graph into triconnected components*, SIAM J. Comput. 2,3 (Sept. 1973), 135-158.

[13] J.Hopcroft, R. A. Tarjan, *Efficinet algorithms for graph manipulation*, Comm. ACM 16, 6 (June 1973), 372-378.

[14] F. T. Leighton, *Layouts for the Shuffle-Exchange Graph and Lower Bound Techniques for VLSI*, Ph. D. Thesis, Department of Mathematics, Massachusetts Institute of Technology, 1981.

[15] C. E. Leiserson, *Area Efficient Graph Layouts (for VLSI)*, Proc. 21st Annu. IEEE Symp. on Foundations Of Computer Science, 1980, 270-281.

[16] A. Lempel, S. Even, I. Cederbaum, *An algorithm for planarity testing of graphs*, P. Rosenstiehl (Ed.), Thery of Graphs, Internat. Symp., Rome, July 1966, 215-232.

[17] Y. P. Liu, *On the linearity of testing planarity of a graph*, Combin. Optim. CORR 84-5. University of Waterloo, 1984.

[18] Y. P. Liu, *A new approach to the linearity of testing planarity of graphs*, Acta Math. Appl. Sinica 4 (1988) 257-265.

[19] Y. Liu, A. Morgana, B. Simeone, *General theoretical results on rectilinear embeddability of graphs*, Acta Math. Appl. Sinica 7 (1991) 187-192.

[20] Y. Liu, A. Morgana, B. Simeone, *A linear algorithm for 2-bend embeddings of planar graphs in the two-dimensional grid*, Discrete Appl. Math. 81 (1998), 69-91.

[21] J. Makowsky, *On the Complexity of Certain Decision Procedures for Propositional Calculus and Random Graphs* undated preprint, Free University, Berlin and Hebrew University, Jerusalem.

[22] Y. Shiloach, *Linear and planar arrangement of graphs*, Ph. D. Dissertation, Weizmann Institute, Rehovat, Israel (1976).

[23] J. L. Szwarcfiter, R. Persiano, A. Oliverira *Orientations with single source and sink*, Discrete Appl. Math. 10 (1985) 313-321.

[24] R. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput. 1,2 (June 1972), 146-159.

[25] R. Tarjan, *An efficient planarity algorithm*, STAN-CS-244-71, Comput. Sci. Dep.,Stanford U. Nov. 1971.

[26] R. Tamassia, I. G. Tollis, *Planar grid embedding in linear time*, IEEE Trans. Circuit Systems 36 (1989) 1230-1234 .

[27] R. Tamassia, I. G. Tollis, *A unified approach to visibility representation of planar graphs*, Discrete Comput. Geom. 1 (1986) 321-341.

[28] W. J. Thron, *Introduction to the Theory of Functions of a Complex Variable* , Wiley,New York, 1953.

[29] L. G. Valiant, *Universality Considerations in VLSI*, IEEE Trans. Comput. 30 (1981) 135-140.