Università di Roma “La Sapienza”

C.d.L. in Informatica

Algoritmi e Strutture Dati

Anno accademico 2000-2001

TOUR DI EULERO E SUE APPLICAZIONI

LOWEST COMMON ANCESTOR

EAR DECOMPOSITION

Saverio Caminiti

Katia Chiusolo

Emanuela Galoppi

Monica Petrucci

CAPITOLO 1 2

TECNICA DEL TOUR DI EULERO 2

INTRODUZIONE 2
STRUTTURE UTILIZZATE 2
DESCRIZIONE DELL'ALGORITMO 4
IMPLEMENTAZIONE 6
COMPLESSITÀ 8
ADERENZA AL MODELLO EREW 8
CONSIDERAZIONI FINALI 9

APPLICAZIONI DELLA TECNICA DEL TOUR DI EULERO 10

RADICARE UN GRAFO 10
Implementazione 11
QUATTRO APPLICAZIONI, UNA SOLUZIONE 13
Visita in Postorder 13
Visita in Preorder 14
Livello di un vertice 14
Numero di discendenti 15
Implementazione 15
Aderenza al modello erew 17
LEFT E RIGHT DI UN NODO 17
Implementazione 19
VISITA INORDER 20
Implementazione 21

LCA (LOWEST COMMON ANCESTORS) 22

INTRODUZIONE 22
MINIMO ANTECEDENTE COMUNE 22
DESCRIZIONE DELL' ALGORITMO 24

CAPITOLO 2 28

EAR DECOMPOSITION 28

INTRODUZIONE 28
EAR DECOMPOSITION 28
DESCRIZIONE DELL'ALGORITMO 30
ESEMPIO 34
APPENDICE A: CICLI FONDAMENTALI 38

RIFERIMENTI BIBLIOGRAFICI 41

TECNICA DEL TOUR DI EULERO

INTRODUZIONE

In questo capitolo studieremo una tecnica che ci permetterà d'affrontare alcuni problemi sugli alberi, quali ad esempio le visite, il radicamento, ecc. Tali problemi, nel calcolo seriale, vengono risolti tramite visite, poco efficienti nel calcolo parallelo. Per ottenere risultati migliori, utilizzeremo un approccio differente.

DEF.: TOUR DI EULERO (TDE)
Dato un grafo G, un Tour di Eulero su G è un cammino chiuso che transita su ogni arco del grafo una ed una sola volta.

Non tutti i grafi ammettono un TDE. Tuttavia noi considereremo soltanto una classe di grafi che ne ammettono sempre almeno uno: i grafi connessi i cui vertici hanno grado pari.

In un grafo cosi fatto, è possibile partire da un arbitrario vertice e per ogni nodo raggiunto, essendo il grado pari, possiamo proseguire tramite un arco ancora mai visitato, a meno di esser tornati sul vertice di partenza.

Inoltre, essendo il grafo connesso, sappiamo che tutti gli archi sono raggiungibili dal vertice di partenza.

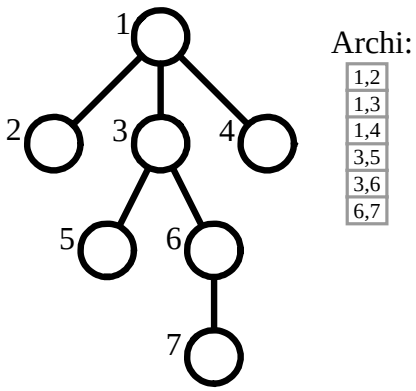
STRUTTURE UTILIZZATE

Il modello d'architettura parallela considerata è una P-RAM a lettura e scrittura esclusive (EREW) con un numero di processori pari a N, dove N é uguale ad n numero dei nodi dell'albero in esame.

Consideriamo ora l'albero (fig.1a), ovvero un grafo connesso aciclico non orientato: se al posto di ogni suo arco non orientato {u, v}, prendiamo una coppia di archi orientati e reciproci (u, v) e (v, u), otteniamo un grafo orientato appartenente alla classe dei grafi connessi i cui vertici hanno grado pari.

Certi dell'esistenza di almeno un TDE su tale grafo, ne costruiremo uno a partire dalla radice e procederemo visitando l'albero in profondità, come nell'esempio riportato in fig.1b.

Introduciamo ora una struttura dati dalla quale si può ottenere il TDE.



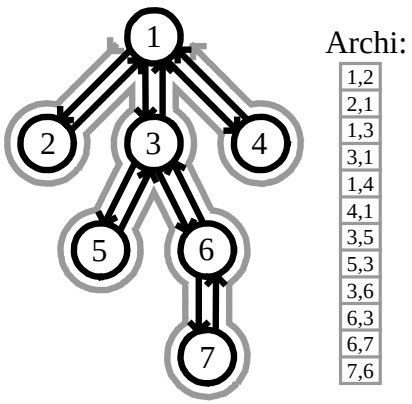


FIG.1a: l'albero di partenza **FIG.1b**: il Tour di Eulero

Memorizziamo il grafo tramite liste d'adiacenza circolari e per ogni arco (u, v) manteniamo un'informazione aggiuntiva: il puntatore al proprio arco reciproco (v, u). Mostriamo nella fig.2 la struttura che desideriamo ottenere.

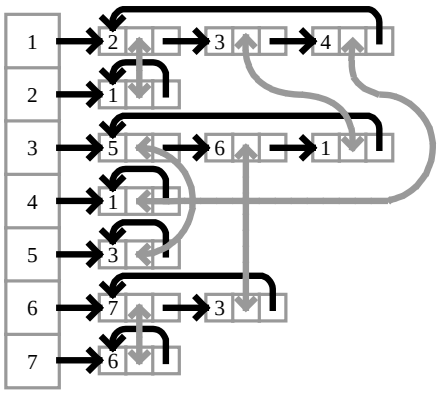


FIG.2: struttura delle liste circolari di adiacenza

A partire da questa struttura, siamo in grado di discendere l'albero grazie alle liste circolari che ne permette la lettura in profondità e grazie al nextdell'arco reciproco, abbiamo la garanzia di lasciare il nodo solo dopo aver visitato interamente il sottoalbero corrispondente.

Vediamo ora, dato il vettore che contiene l'elenco di tutti gli archi, come ottenere le liste circolari.

Associamo ad ogni arco un processore; notiamo subito che il numero d'archi ottenuti è esattamente 2(n-1), poiché come già sappiamo un albero ha esattamente n-1 archi.

Nonostante il numero degli archi sia maggiore del numero dei processori, è sufficiente gestire l'algoritmo in due passi consecutivi, senza gravare sulla complessità asintotica dell'algoritmo.

Vediamo ora l'algoritmo vero e proprio che ci permette di trovare il TDE in tempo O(log N).

DESCRIZIONE DELL'ALGORITMO

La costruzione del TDE, si compone di due parti:

- nella prima creeremo le liste circolari che ci occorrono per collegare ogni nodo al proprio padre e ai propri figli;
- nella seconda parte doteremo ogni arco di un puntatore al proprio arco reciproco (creando di fatto il doppio orientamento di cui abbiamo già parlato).

Da principio ordineremo il vettore degli archi in ordine lessicografico, in modo da ottenere che tutti gli archi che partono da uno stesso nodo, diventino elementi consecutivi nel vettore.

A questo punto si può procedere con un confronto eseguito con tutti i processori in parallelo, fra ogni arco (u, v), e l'arco (w, k) posto subito sopra di esso nel vettore; se il nodo u dell'arco (u, v) è diverso dal nodo w dell'arco (w, k), il processore associato sa che l'arco di cui è competente, è "capo-blocco" e si attiverà.

In questo modo siamo in grado di limitare i blocchi all'interno del vettore.

Ciascuno di questi processori provvederà a scrivere in un apposito vettore che l'arco (u, v) da loro considerato è l'inizio della lista di adiacenza relativa al nodo u.

In un passo seguente basterà che ogni processore imposti il puntatore all'elemento successivo; ad eccezione di quelli alla fine di un blocco che punteranno al primo arco del proprio blocco.

In questo modo otteniamo le liste circolari che volevamo.

Ora dobbiamo creare i puntatori all'arco reciproco.

Si può osservare, che l'informazione relativa a questo arco è stata creata da noi, quindi al

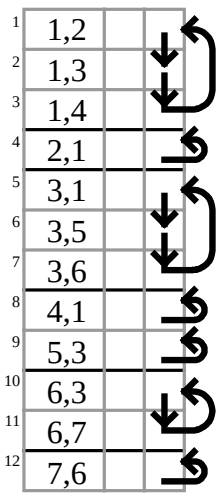


FIG.3: vettore degli archi in cui si realizzano le liste circolari

momento in cui viene riempito il vettore, per ogni nodo si potrebbe inserire l'arco e il puntatore al proprio reciproco.

Il problema che si crea in questo caso è che una volta ordinato il vettore in modo lessicografico, questo puntatore diventa inconsistente, poiché indica un elemento del vettore in cui non è più contenuto l'arco reciproco.

Bisognerebbe quindi inserire l'informazione dopo l'ordinamento; ma anche in questo caso, come dire ad ogni arco dove trovare nel vettore il proprio arco reciproco, senza doverlo scorrere tutto?

A questo proposito vale la pena osservare più da vicino la reale implementazione delle strutture che stiamo usando.

Fino a questo momento abbiamo descritto il vettore degli archi come un vettore i cui elementi sono formati da tre campi (fig.4):

- una che contiene le etichette delle coppie di nodi che identificano gli archi;
- una per i puntatori all'arco reciproco;
- una per i puntatori che occorrono alla costruzione delle liste circolari.

(from, to)	rev	next

FIG.4: struttura che contiene la terna di campi di un arco

Subito al primo esame, si potrebbe facilmente intuire che a creare problemi è la rigidità della struttura appena descritta.

Una soluzione può essere quella di ordinare, al posto di un vettore di terne, un vettore ausiliare contenente puntatori agli archi. Questo nuovo vettore ci consentirà di ordinare il modo lessicografico gli archi, senza però modificarne la reale locazione fisica. Il puntatore relativo all'arco reciproco non perderebbe la coerenza e potrebbe essere inserito nel vettore insieme all'arco stesso, così come avevamo già proposto.

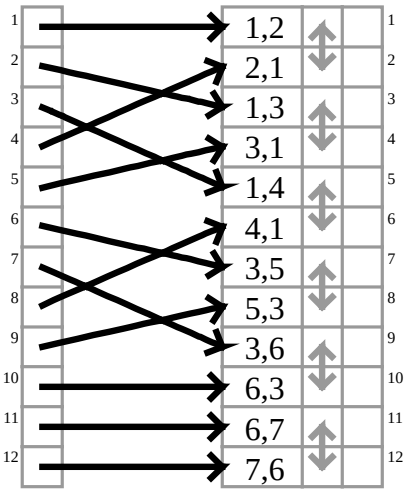


FIG.5: vettore ausiliare per l'ordinamento

Nella figura 5, alla sinistra del vettore degli archi, è rappresentata la struttura appena introdotta.

Le strutture descritte, opportunamente lette ci restituiscono il TDE come rappresentato nella fig.1b.

IMPLEMENTAZIONE

Vediamo ora come sia possibile implementare quanto finora detto verificando, che il modello EREW non sia violato.

Iniziamo considerando l'input:

Dell'albero sul quale costruiamo il tour di Eulero, ciò che ci serve è un vettore contenente gli (n-1) archi non orientati. Chiamiamo questo vettore inputEdges.

1. La prima cosa da fare è passare dall'albero al grafo orientato. Procederemo quindi alla creazione di due archi orientati in sostituzione di quelli in input. Li memorizzeremo in un vettore di 2(n-1) record, nei quali manterremo la coppia ordinata di vertici, il puntatore al reciproco e un puntatore all'arco successivo nella lista circolare. Chiamiamo tale vettore edges.

```
for i=1 to n-1 pardo
  (u, v) = inputEdges[i];
  edges[i] = ((u, v), n-1+i, null);
  edges[n-1+i] = ((v, u), i, null);
  sortEdges[i] = i;
  sortEdges[n-1+i] = n-1+i;
```

Le ultime due istruzioni servono ad inizializzare il vettore di puntatori che poi ordineremo; si noti che i puntatori sono in realtà gli indici degli archi nel vettore edges.

2. Il secondo passo consiste appunto nell'ordinare il vettore sortEdges in modo da ottenere l'ordine lessicografico degli archi.

```
lexicographicsSort(sortEdges);
```

L'analisi dell'implementazione di un algoritmo efficiente di ordinamento su una macchina parallela, esula dallo scopo di questo capitolo; si noti tuttavia che la chiave di confronto tra gli elementi non sarà il valore stesso memorizzato in ogni locazione del vettore, ma la coppia di vertici dell'arco da essa puntato.

3. Dobbiamo ora dividere in blocchi il vettore ordinato degli archi, per capire quale sia il primo arco d'ogni lista circolare d'adiacenza. Per fare ciò lavoreremo in due passi sequenziali, prima sui primi n-1 archi e poi sui restanti n-1, ricordiamo infatti

che abbiamo solo n processori.

```
for passo=0 to 1 do
  for p=1 to n-1 pardo
    /* indice dell'arco da confrontare in sortEdges */
    i = p + passo*(n-1);
    /* se è il primo arco */
    if (i == 1) then adjList[1] = 1;
    else
      /* confrontare l'i-esimo arco con il precedente */
      e = sortEdges[i];
      prevEdge = sortEdges[i-1];
      if (edges[e].from <> edges[prevEdge].from) then
        adjList[edges[e].from] = e;
```

Appare necessario chiarire che adjList è il vettore che, nella i-esima locazione contiene il puntatore all'arco a capo della i-esima lista circolare.

4. Per completare la nostra struttura dobbiamo ora realizzare le liste circolari, facendo puntare ogni arco al successivo secondo l'ordine lessicografico. Fanno eccezione quegli archi che sono alla fine di una lista, ovvero tutti gli e per i quali, essendo e' l'arco successivo di e, si ha che i vertici di partenza di e e di e' sono differenti.

```
for passo=0 to 1 do
  for p=1 to n-1 pardo
    /* indice dell'arco da gestire */
    i = p + passo*(n-1);
    if (i == 2*(n-1)) then         /* l'ultimo arco */
      edges[sortEdges[i]].next = adjList[n];
    else
      e = sortEdges[i];            /* l'i-esimo arco */
      nextEdge = sortEdges[i+1];   /* l'(i+1)-esimo arco */
      if (edges[e].from <> edges[nextEdge].from) then
        edges[e].next = adjList[edges[e].from];
      else edges[e].next = nextEdge;
```

Ricordiamo che per ogni arco è memorizzata una terna del tipo ((from, to), rev, next), dove (from, to) sono gli estremi dell'arco, rev è il puntatore all'arco reciproco e nextè il puntatore al successivo nella lista circolare di adiacenza.

5. Per concludere vediamo come trasformare questa struttura in un'unica lista circolare rappresentante il tour di Eulero: sappiamo che dato un arco e il suo successore nel TDE è il next del suo reciproco. Possiamo quindi scrivere quest'informazione direttamente in un vettore TDE, il cui i-esimo elemento è l'indice dell'arco successore dell'i-esimo arco nel tour.

```
for i=1 to n-1 pardo
  TDE[i] = edges[edges[i].rev].next;
  TDE[edges[i].rev] = edges[i].next;
```

COMPLESSITÀ

Calcoliamo ora la complessità dell'algoritmo, analizzandolo puntualmente.

Il primo punto descritto è una operazione compiuta in parallelo da tutti i processori, che come abbiamo già visto sono la metà del numero degli archi. Il caricamento dei dati nel vettore è quindi realizzato in soli due passi. **O(1)**.

Il secondo punto si riferisce ad un generico algoritmo d'ordinamento, arricchito solamente di qualche accortezza relativa all'accesso ai dati da confrontare. Ciò non modifica la complessità asintotica dell'algoritmo di base. Anche se non lo abbiamo ancora visto, un algoritmo efficiente d'ordinamento di N elementi costa **O(log N)** con N processori.

Il terzo punto, relativo ai confronti tra coppie d'archi all'interno del vettore, viene anch'esso compiuto in tempo costante, poiché tutti i processori riescono a confrontare tutte le coppie, in soli due passi. Questo appare evidente dal ciclo sequenziale nella parte dell'algoritmo in esame. **O(1)**.

Il quarto punto si riferisce alla costruzione delle liste circolari, create sfruttando l'ordine del vettore e la sua divisione in blocchi. Per questo motivo ogni processore sa quando e come reperire in tempo costante l'indirizzo del puntatore che deve allocare. **O(1)**.

Il quinto ed ultimo punto, è anch'esso eseguito in tempo costante, poiché realizzato in parallelo da tutti i processori. **O(1)**.

L'analisi puntuale, evidenzia subito che la complessità dell'algoritmo dipende unicamente dall'operazione d'ordinamento, poiché tutte le altre operazioni vengono realizzate in tempo costante. Quindi la complessità dell'algoritmo che calcola il TDE è **O(log N)**.

ADERENZA AL MODELLO EREW

Per mostrare che l'algoritmo descritto non viola il modello dato, osserviamo che nella

maggior parte dei casi, quando i processori lavorano in parallelo, accedono, sia in lettura sia in scrittura, a zone di memoria completamente distinte.

Degno di nota è solo il punto 3, dove ciascun processore confronta il suo arco con il precedente, potrebbe sembrare quindi che in due leggeranno la stessa zona di memoria. Tuttavia, poiché i processori lavorano in modo sincrono, prima leggono tutti l'informazione del proprio arco e dopo quella del precedente: questo ci assicura che non avvengano mai accessi concorrenti. Valgono analoghe considerazioni per quanto concerne il punto 4 durante la creazione delle liste circolari.

Il modello EREW è stato quindi rispettato.

CONSIDERAZIONI FINALI

Prima di terminare questa sezione è opportuno far notare che il TDE calcolato, è vettore non ordinato che per ogni arco i fornisce l'indice dell'arco successivo nel tour.

Nelle applicazioni tuttavia questa struttura può risultare poco efficiente. Ad esempio nel calcolo del minimo antecedente comune (LCA), si farà uso di un vettore che corrisponde all'ordinamento dei vertici dato dal TDE.

Vediamo quindi come sia possibile calcolare il numero d'ordine di ogni arco nel TDE.

```
for i=1 to 2*(n-1) pardo
  preSum[i].value = 1;
  preSum[i].next = TDE[i];
perSum[edges[tailList[1]].rev] = null;
doPrefixSums(preSum);
```

Per ottenere la sequenza di vertici definita dal tour, è sufficiente memorizzare in un array il vertice di partenza di ogni arco.

```
for i=1 to 2*(n-1) pardo
  ord[i] = edges[preSum[i]].from;
```

Applicazioni della tecnica del Tour di Eulero

Vediamo ora come sfruttare il Tour di Eulero (TDE) per risolvere alcuni problemi che nel calcolo sequenziale vengono risolti usando le visite, che abbiamo gia visto non essere efficienti nel calcolo parallelo.

Il modello a cui facciamo riferimento è una P-RAM a lettura e scrittura esclusiva (EREW), con un numero N di processori, che noi ipotizziamo essere pari al numero degli archi. In realtà abbiamo già visto in precedenza che il numero degli archi è 2(n-1), dove n=N è il numero dei nodi dell'albero in esame. Questa discordanza tra l'assunzione fatta e il reale numero degli archi, può essere facilmente risolta facendo gestire ad ogni processore due archi anziché uno, risolvendo il piccolo inconveniente in tempo costante.

Sottolineiamo che tale modifica non altererebbe né la complessità asintotica dei nostri algoritmi, né implicherebbe modifiche sostanziali alla loro implementazione.

RADICARE UN GRAFO

Radicare un grafo G in un nodo $r \in V(G)$, significa stabilire una relazione gerarchica tra tutti i nodi in V(G) a partire dal vertice r, ottenendo così un albero.

Essendo il TDE un cammino chiuso che rappresenta una visita in profondità dell'albero T, l'idea è quella di partire dalla prima occorrenza del vertice r, per leggerlo come una visita in profondità dell'albero T'.

Riprendendo l'esempio del capitolo precedente radichiamo l'albero nel vertice 3. Per cominciare scriviamo il TDE in forma sequenziale elencandone gli archi a partire dal primo arco del vertice 3 ovvero (3, 1). Quindi assegniamo prima ad ogni arco il valore 1 e dopo ne calcoliamo le somme prefisse (Σ_{Pre}):

Tour di Eulero:	(3,1)	(1,4)	(4,1)	(1,2)	(2,1)	(1,3)	(3,5)	(5,3)	(3,6)	(6,7)	(7,6)	(6,3)
Assegnazioni:	1	1	1	1	1	1	1	1	1	1	1	1
Σ_{Pre}:	1	2	3	4	5	6	7	8	9	10	11	12

Possiamo ora affermare che:

$\forall$ arco (x, y) x è il padre di y sse $\Sigma_{Pre}(x, y) < \Sigma_{Pre}(y, x)$.

IMPLEMENTAZIONE

0. Il primo passo consiste nel calcolare il Tour di Eulero ottenendo così i vettori adjList ed edges. Ricordiamo che l'indice dell'arco (i+1)-esimo nel tour, è memorizzato nella posizione i-esima del vettore TDE.

1. Possiamo ora costruire in tempo costante la lista sulla quale calcolare le somme prefisse, memorizziamo questi dati nel vettore preSum.

```
for i=1 to 2*(n-1) pardo
  preSum[i].value = 1;
  preSum[i].next = TDE[i];
```

2. Eseguiamo ora le somme prefisse, facendo uso di un'opportuna funzione.

```
doPrefixSums(preSum);
```

3. Ricordando che dato un arco (x, y), x è il padre di y sse $\Sigma_{Pre}(x, y) < \Sigma_{Pre}(y, x)$, riempiamo il vettore dei puntatori al padre p, per ogni nodo.

```
for i=1 to 2*(n-1) pardo
  if preSum[i] < preSum[edges[i].rev] then
    p[edges[i].to] = p[edges[i].from];
```

Si noti che questa implementazione, se pur corretta dal punto di vista logico, in pratica non può funzionare perché il vettore preSum è una lista circolare, mentre le somme prefisse devono essere fatte su una lista il cui ultimo elemento deve avere null nel campo next.

1bis. Tra il passo **1** ed il **2** sarà quindi necessario capire quale degli archi deve essere l'ultimo della lista ed impostarlo a null.

Per trasformare il nostro tour in un cammino, abbiamo scelto come arco da cui iniziare, il primo di quelli uscenti dalla radice r. L'arco finale sarà quindi l'ultimo di quelli entranti in r, ovvero il reciproco dell'ultimo arco nella lista di adiacenza associata alla radice. Vediamone una possibile implementazione:

```
for i=1 to 2*(n-1) pardo
  if edges[edges[i].rev].next = adjList[r] then
    perSum[i] = null;
```

Questa soluzione potrebbe esser corretta se non venisse violato il modello EREW: è evidente che nella parte destra della condizione dell'if, tutti i processori leggono

contemporaneamente le stesse informazioni ovvero r e adjList[r]. Ciò ci costringe ad abbandonare questa ipotesi.

Si potrebbe pensare di creare delle copie in memoria delle informazioni con la tecnica **broadcast**. Tuttavia anche questa soluzione non è molto efficiente poiché ci costerebbe O(log n) e, pur non aumentando l'ordine di complessità asintotica, si appesantirebbe il tempo effettivo di esecuzione per realizzare un'operazione banale.

Anticipiamo che tale problema si riproporrà in termini analoghi anche per le successive applicazioni della tecnica del TDE. Appare quindi motivata l'idea di introdurre nell'algoritmo stesso che calcola il tour un passo **3bis**, che modifichi opportunamente la struttura. In questa fase memorizziamo in un vettore i puntatori agli archi finali di ogni lista di adiacenza, operando analogamente all'identificazione degli archi iniziali effettuata al passo **3**.

TDE.3bis.

```
for passo = 0 to 1 do
  for p = 1 to n-1 pardo
    /* indice dell'arco da confrontare in sortEdges */
    i = p + passo*(n-1);
    /* se e' l'ultimo arco */
    if (i == 2*(n-1)) then tailList[n] = i;
    else
      /* confrontare l'i-esimo arco con il successivo */
      e = sortEdges[i];
      succEdge = sortEdges[i+1];
      if (edges[e].from <> edges[succEdge].from) then
        tailList[edges[e].from] = e;
```

Si noti che al pari del passo TDE.**3** anche il TDE.**3bis** costa O(1).

Con queste nuove informazioni a disposizione ora riscrivere il passo **1bis** dell'algoritmo per radicare un albero in modo che venga eseguito da un solo processore.

```
perSum[edges[tailList[r]].rev] = null;
```

Questa sola riga costituisce per intero il passo **1bis**, ed al contrario della prima versione proposta, non viola il modello EREW e si mostra decisamente più efficiente

rispetto all'utilizzo del broadcast, in quanto entrambi i passi aggiunti ai due algoritmi, costano O(1).

QUATTRO APPLICAZIONI, UNA SOLUZIONE

Le quattro applicazioni che seguono sono nella loro implementazione del tutto simili, poiché sfruttano la medesima idea.

Gli algoritmi ai quali ci stiamo riferendo sono:

- Visita in Postorder;
- Visita in Preorder;
- Livello di un vertice;
- Numero dei discendenti di un nodo.

Quello che caratterizza queste quattro applicazioni è la necessità che hanno d'eseguire una visita in profondità con lo scopo di computare e/o ordinare i nodi dell'albero.

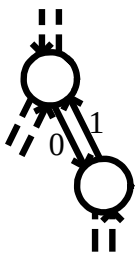
Ad esempio, l'unica caratteristica che distingue i due tipi di visita è che in un caso si numera un nodo dopo di aver visitato il sottoalbero corrispondente (visita in postorder), nel secondo caso si numera il nodo prima aver visitato il suo sottoalbero (visita in preorder).

Invece nel caso del Livello di un vertice e del Numero dei Discendenti di un nodo, si visita l'albero, sfruttando il senso di percorrenza degli archi.

Per capire meglio, entriamo più nel dettaglio dei quattro algoritmi.

VISITA IN POSTORDER

Nella visita in postorder si numera un nodo l'ultima volta che lo si visita, l'idea è quindi quella di numerare i figli prima dei padri.



Per questo motivo prendiamo in considerazione l'ultimo arco uscente da ciascun nodo v, ovvero l'arco che va dal figlio verso il padre: (v, p(v)).

Appare ovvio il perché si calcolano le somme prefisse (Σ_{Pre}), assegnando valore 0 agli archi discendenti e 1 agli archi ascendenti come in figura, ottenendo quanto segue:

Tour di Eulero:	(1,2)	(**2**,1)	(1,3)	(3,5)	(**5**,3)	(3,6)	(6,7)	(**7**,6)	(**6**,3)	(**3**,1)	(1,4)	(**4**,1)
Assegnazioni:	0	1	0	0	1	0	0	1	1	1	0	1
Σ_{Pre}:	0	1	1	1	2	2	2	3	4	5	5	6

Nella tabella sono evidenziati gli archi del tipo (v, p[v]) il cui valore sarà proprio il numero d'ordine nella visita in postorder relativo al vertice v.

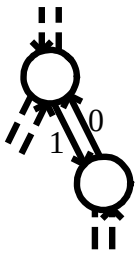
Più formalmente possiamo scrivere:

$\forall$ nodo $v \neq r$	$post[v] = \Sigma_{Pre}(v, p[v])$
se $v = r$	$post[v] = n$

Risulta evidente che la radice è l'ultimo vertice visitato in postorder, pertanto ha numero d'ordine pari ad n.

VISITA IN PREORDER

Nella visita in preorder si numera un vertice la prima volta che lo si visita e non quando lo si lascia, quindi l'arco a cui è associata l'informazione relativa al vertice v, sarà (p[v], v).



Volendo numerare i padri prima dei figli in questo caso, ad ogni arco discendente assegniamo il valore 1 e ad ogni arco ascendente il valore 0. Calcoliamo quindi le somme prefisse (Σ_{Pre}):

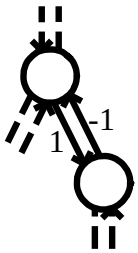
Tour di Eulero:	(1,**2**)	(2,1)	(1,**3**)	(3,**5**)	(5,3)	(3,**6**)	(6,**7**)	(7,6)	(6,3)	(3,1)	(1,**4**)	(4,1)
Assegnazioni:	1	0	1	1	0	1	1	0	0	0	1	0
Σ_{Pre}:	1	1	2	3	3	4	5	5	5	5	6	6

In realtà il primo nodo nella visita preorder è la radice, quindi sarà necessario aggiungere 1 ai valori ottenuti. Formalmente:

$\forall$ nodo $v = r$	$pre[v] = 1$
se $v \neq r$	$pre[v] = \Sigma_{Pre}(p[v], v) + 1$

LIVELLO DI UN VERTICE

Il nostro scopo in questo caso è quello di numerare un vertice secondo il proprio livello, quindi il senso di percorrenza di un arco (da padre a figlio o da figlio a padre), corrisponde a salire o a scendere di un livello l'albero.



Per questo motivo assegniamo, ad ogni arco ascendente il valore -1 e ad ogni arco discendente il valore +1.

Ora possiamo calcolare le somme prefisse (Σ_{Pre}):

Tour di Eulero:	(1,**2**)	(2,1)	(1,**3**)	(3,**5**)	(5,3)	(3,**6**)	(6,**7**)	(7,6)	(6,3)	(3,1)	(1,**4**)	(4,1)
Assegnazioni:	+1	-1	+1	+1	-1	+1	+1	-1	-1	-1	+1	-1
Σ_{Pre}:	1	0	1	2	1	2	3	2	1	0	1	0

Il livello del vertice v è la Σ_{Pre} dell'arco (p[v], v), ad eccezione della radice che ha livello 0.

$$\forall \text{ nodo } v \neq r \quad level[v] = \Sigma_{Pre}(p[v], v)$$
$$\text{se } v = r \quad level[v] = 0$$

NUMERO DI DISCENDENTI

L'applicazione consiste nel calcolare, per ciascun nodo v, il numero di nodi presenti nel sottoalbero radicato in v, compreso v.

L'idea è sfruttare la visita in postorder che numera i nodi dopo aver visitato il relativo sottoalbero. Per questo motivo assegniamo, ad ogni arco ascendente il valore 1 e ad ogni arco discendente il valore 0 come nella visita in postorder.

Calcoliamo quindi le somme prefisse (Σ_{Pre}):

Tour di Eulero:	(**1**,2)	(**2**,1)	(**1**,3)	(**3**,5)	(**5**,3)	(**3**,6)	(**6**,7)	(**7**,6)	(**6**,3)	(**3**,1)	(**1**,4)	(**4**,1)
Assegnazioni:	0	1	0	0	1	0	0	1	1	1	0	1
Σ_{Pre}:	0	1	1	1	2	2	2	3	4	5	5	6

I nodi presenti nel sottoalbero di v sono quelli compresi tra la prima e l'ultima apparizione di v nel TDE. Quindi il numero di discendenti del nodo v è $\Sigma_{Pre}(v, p[v])$-$\Sigma_{Pre}(p[v], v)$ ovvero la differenza tra i valori associati al primo arco entrante in v e all'ultimo arco da esso uscente.

Fa eccezione la radice che ha n discendenti.

$$\forall \text{ nodo } v \neq r \quad size[v] = \Sigma_{Pre}(v, p[v])-\Sigma_{Pre}(p[v], v)$$
$$\text{se } v = r \quad size[v] = n$$

IMPLEMENTAZIONE

Presentiamo di seguito un'implementazione comune per tutte le applicazioni viste sopra.

0. Il primo passo consiste nel calcolare il Tour di Eulero.

1. Possiamo ora costruire in tempo costante la lista preSum facendo uso di due costanti, A e D: sono relative al valore da assegnare rispettivamente agli archi ascendenti e discendenti poiché, tali valori variano a seconda dell'applicazione:

APPLICAZIONE	A	D
POSTORDER	1	0
PREORDER	0	1
LIVELLO DI UN NODO	-1	1
NUMERO DI DISCENDENTI	1	0

```
for i=1 to 2*(n-1) pardo
  if edges[i].to < edges[i].from then
    /* arco ascendente */
    preSum[i].value = A;
  else
    /* arco discendente */
    preSum[i].value = D;
  preSum[i].next = TDE[i];
```

Va messo a null il puntatore dell'ultimo arco.

```
perSum[edges[tailList[1]].rev] = null;
```

2. Eseguiamo ora le somme prefisse, facendo uso di un'opportuna funzione.

```
doPrefixSums(preSum);
```

3. Procediamo memorizzando i risultati in appositi vettori a seconda di quale applicazione si vuole implementare.

3.POSTORDER

```
post[1] = n;
for i=1 to 2*(n-1) pardo
  if edges[i].to < edges[i].from then
    post[edges[i].from] = preSum[i];
```

3.PREORDER

```
pre[1] = 1;
for i=1 to 2*(n-1) pardo
  if edges[i].to > edges[i].from then
    pre[edges[i].to] = preSum[i] + 1;
```

3.LIVELLO

```
level[1] = 0;
for i=1 to 2*(n-1) pardo
  if edges[i].to > edges[i].from then
    level[edges[i].to] = preSum[i];
```

3.DISCENDENTI

```
size[1] = n;
for i=1 to 2*(n-1) pardo
  if edges[i].to < edges[i].from then
    size[edges[i].from] = preSum[i] - preSum[edges[i].rev];
```

ADERENZA AL MODELLO EREW

L'unico passo dell'implementazione che potrebbe destare qualche preoccupazione, è il passo **3.**, dove i processori in parallelo sembrerebbero accedere ad una stessa locazione di memoria.

Tuttavia ciò non accade mai, in quanto il corpo di ciascun if verrà eseguito soltanto da un processore; per ogni vertice v, infatti, soltanto un arco verificherà la condizione dell'if: esiste un solo arco (v, p[v]) o (p[v], v).

LEFT E RIGHT DI UN NODO

Introduciamo ora un'applicazione i cui risultati saranno poi riutilizzati in applicazioni successive.

Chiamiamo left l'occorrenza più a sinistra di un vertice nel TDE, e il right quella più a destra. Più esattamente il left (right) è un vettore di booleani il cui i-esimo elemento vale 1 sse l'arco di indice i è il primo (ultimo) arco uscente dal suo vertice di partenza.

T. di Eulero:	(**1**,2)	(**2**,1)	(**1**,3)	(**3**,5)	(**5**,3)	(**3**,6)	(**6**,7)	(**7**,6)	(**6**,3)	(**3**,1)	(**1**,4)	(**4**,1)	(**1**,2)
Left:	1	1	0	1	1	0	1	1	0	0	0	1	0
Right:	0	1	0	0	1	0	0	1	1	1	0	1	1

Il vertice di partenza d'ogni arco è evidenziato in grassetto nella tabella, in quanto è il nodo cui si riferisce il valore del left e del right.

Si noti che gli ultimi due vettori, sono più lunghi degli altri, in quanto tengono conto del fatto che il tour torna sul vertice di partenza.

Al fine di procedere al calcolo di questi vettori, notiamo che l'apparizione di un vertice è la prima, se il vertice precedente era di livello minore ovvero se si sta scendendo dal padre al figlio.

Analogamente, se l'arco va dal figlio al padre, allora il livello del vertice successivo è minore. In questo caso l'apparizione di tale vertice sarà l'ultima.

Servono ora le informazioni relative al livello dei nodi di partenza di ciascun arco. Rivediamo le informazioni sul livello calcolate con l'applicazione precedente e confrontiamole con quelle che dobbiamo ricavare:

T. di Eulero:	(**1**,2)	(**2**,1)	(**1**,3)	(**3**,5)	(**5**,3)	(**3**,6)	(**6**,7)	(**7**,6)	(**6**,3)	(**3**,1)	(**1**,4)	(**4**,1)	(**1**,2)
Assegnaz.:	+1	-1	+1	+1	-1	+1	+1	-1	-1	-1	+1	-1	
Σ_{Pre}:	1	0	1	2	1	2	3	2	1	0	1	0	
Level:	**0**	**1**	**0**	**1**	**2**	**1**	**2**	**3**	**2**	**1**	**0**	**1**	**0**

In realtà nel calcolo del livello precedentemente proposto, il valore delle somme prefisse relative all'arco i, era il livello del vertice finale. Mentre in questo contesto si lavora sui vertici iniziali dei vari archi.

Per semplificare, quindi, si procederà ad una modifica di tale struttura: si slitterà verso destra di una posizione. In questo modo associamo l'informazione relativa al livello, non più al vertice finale di un arco ma, al vertice iniziale dell'arco successivo: tali vertici sono, infatti, coincidenti.

Sia l'arco i-esimo (u, v), possiamo affermare che:

left[i] = 1 sse level[i.prev] < level[i]

right[i] = 1 sse level[i.next] < level[i]

dove con prev e next identifichiamo il precedente ed il successivo arco nel TDE. Si noti che l'informazione prev non è memorizzata in nessuna struttura dati, sarà quindi necessario crearne una temporanea durante l'implementazione.

IMPLEMENTAZIONE

0. Iniziamo con il calcolare il Tour di Eulero.

1. Riprendiamo il codice visto per il calcolo del livello di un vertice e sfruttiamolo per calcolare il vettore level il quale contiene per ogni arco, il livello del nodo iniziale.

```
for i=1 to 2*(n-1) pardo
  if edges[i].to < edges[i].from then
    preSum[i].value = -1;
  else preSum[i].value = 1;
  preSum[i].next = TDE[i];
```

```
preSum[edges[tailList[1]].rev] = null;
```

```
doPrefixSums(preSum);
```

Il valore della somma prefissa relativa all'arco i-esimo deve diventare il livello dell'arco successivo nel tuor.

```
for i=1 to 2*(n-1) pardo
  level[TDE[i]] = preSum[i];
```

2. È ora necessario calcolare l'arco precedente d'ogni arco nel tour, manteniamo quindi queste informazioni in un vettore temporaneo prevTDE.

```
for i=1 to 2*(n-1) pardo
  prevTDE[TDE[i]] = i;
```

3. Procediamo al calcolo del valore del left e del right per ogni arco.

```
for i=1 to 2*(n-1) pardo
  if level[prevTDE[i]] < level[i] then
    left[i] = 1;
  else left[i] = 0;
  if level[TDE[i]] < level[i] then
    right[i] = 1;
  else right[i] = 0;
```

4. Notiamo che sarà utile poter accedere all'informazione di left e right tramite vertice. Manteniamo, quindi, tali informazioni nei vettori leftByVertex e rightByVertex.

```
for i=1 to 2*(n-1) pardo
  if left[i] then
```

```
    leftByVertex[edges[i].from] = i;
  if right[i] then
    rightByVertex[edges[i].from] = i;
```

VISITA INORDER

Prima di tutto è necessario ricordare che la visita inorder è definita unicamente su alberi binari. È inoltre opportuno notare che in un TDE, qualora un nodo non abbia fratello, non è possibile decidere se questo sia figlio sinistro o destro. Ci limiteremo quindi all'esame di alberi binari in cui ogni nodo non foglia, abbia entrambi i figli.

Si potrebbe non tener conto della restrizione appena fatta e, realizzare questa applicazione su un albero binario generico, se la struttura dati che ospita il TDE contenesse anche l'informazione se un nodo è figlio sinistro o destro.

Questa ipotesi d'implementazione è però lasciata al lettore.

A differenza delle due visite viste in precedenza, in questo caso sarà più complesso identificare quali siano gli archi del TDE da considerare nella numerazione. Per calcolare l'indice associato ad ogni nodo nella visita, sfrutteremo quindi il left e il right.

Osserviamo che i nodi per i quali l'apparizione sinistra e destra coincidono sono delle foglie.

Per tutti gli altri nodi, sappiamo che hanno esattamente due figli, quindi compariranno nel tour tre volte. Quello che interessa per la visita inorder è l'apparizione centrale; identificata dal non essere ne left ne right.

Assegneremo quindi 1 a quegli archi in corrispondenza dei quali left e right siano entrambi 1 (foglie) o entrambi 0 (apparizioni centrali).

Eseguiamo ora le somme prefisse:

T. di Eulero:	(**1**,2)	(**2**,1)	(**1**,3)	(**3**,4)	(**4**,3)	(**3**,5)	(**5**,6)	(**6**,5)	(**5**,7)	(**7**,5)	(**5**,3)	(**3**,1)	(**1**,2)
Left:	1	1	0	1	1	0	1	1	0	1	0	0	0
Right:	0	1	0	0	1	0	0	1	0	1	1	1	1
Assegnazioni	0	1	1	0	1	1	0	1	1	1	0	0	0
Σ_{Pre}:	0	1	2	2	3	4	4	5	6	7	7	7	7

Il risultato della somma prefissa, in corrispondenza di un arco (u, v), a cui è stato assegnato valore 1, equivale al numero d'ordine nella visita del vertice u.

IMPLEMENTAZIONE

0. Iniziamo con il calcolare il Tour di Eulero.

1. Calcoliamo left e right come visto nell'applicazione precedente.

2. Calcoliamo le somme prefisse.

```
for i=1 to 2*(n-1) pardo
  if left[i] = right[i] then
    preSum[i].value = 1;
  else preSum[i].value = 0;
  preSum[i].next = TDE[i];
```

```
perSum[edges[tailList[1]].rev] = null;
```

```
doPrefixSums(preSum);
```

3. Calcoliamo, sulla base delle somme prefisse, il vettore inorder.

```
for i=1 to 2*(n-1) pardo
  if left[i] = right[i] then
    inorder[edges[i].from] = preSum[i];
```

LCA (Lowest Common Ancestors)

INTRODUZIONE

Il minimo antecedente comune di due vertici u e v di un albero radicato T è un nodo w che è un antecedente sia di u sia di v, ed è il più lontano dalla radice. Sono molte le situazioni in cui si richiede di trovare il minimo antecedente comune di un'arbitraria coppia di vertici u e v di un albero radicato T=(V, E), tale problema è denotato con LCA (u, v), **lowest-common-ancestors** (LCA). Il nostro obiettivo è quello di trovare LCA(u, v) il più velocemente possibile, infatti, si cerca di rispondere ad ogni domanda in tempo sequenziale O(1). Sarà sviluppato un algoritmo per risolvere il problema LCA in un tempo O(logn).

MINIMO ANTECEDENTE COMUNE

Ci sono due situazioni particolari in cui il problema LCA può essere risolto immediatamente.

Il primo caso è quello in cui l'albero T è un cammino semplice. Calcolando la distanza di ogni vertice dalla radice siamo in grado di rispondere ad ogni domanda LCA (u,v) in tempo costante attraverso la comparazione delle distanze di u e v dalla radice. Il minimo antecedente comune sarà il vertice più "vicino" alla radice fra u e v.

Il secondo caso è quando T è un albero binario completo. Dopo aver etichettato ogni vertice con la numerazione inorder è possibile rispondere ad ogni domanda in tempo costante. L'attraversamento inorder di un albero binario T con radice r consiste nell'attraversamento inorder del sottoalbero sinistro della radice di r seguito da r seguito nell'attraversamento inorder del sottoalbero destro di r.

Per ogni coppia di vertici x e y, LCA (x, y) può essere trovato con il seguente metodo.

Esprimiamo x e y come numeri binari e consideriamo il numero delle posizioni dei bit a partire da sinistra a destra. Sia i la posizione del primo bit da sinistra in cui x e y differiscono, dunque i primi i-1 bit più a sinistra $z_1 z_2 ... z_{i-1}$ di x e y sono uguali, e i bit i-esimi sono differenti. Allora, LCA(x, y) è uguale al numero la cui rappresentazione binaria è data da $z_1 z_2 ... z_{i-1}100.....0$.

In generale, però, il nostro approccio è basato sulla tecnica del Tour d'Eulero.

Determiniamo innanzi tutto il Tour di Eulero di T = (V, E). Successivamente sostituiamo ogni arco (u, v) col vertice v e definiamo l'array d'Eulero A che corrisponde all'ordinamento dei vertici dato dal Tour di Eulero, con la radice inserita in testa.

Se |V| = n, la lunghezza di A è m = 2n-1. Da A deriviamo l'array B considerando il livello di ogni elemento di A, quindi B è denotato da B= level(A).

ESEMPIO Consideriamo il seguente albero:

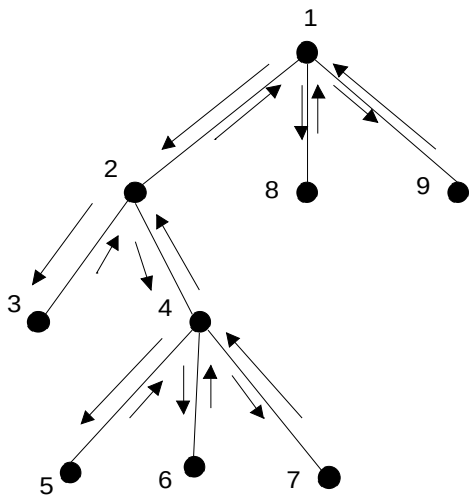


Fig. 1: esempio 1

Il corrispondente array di Eulero A è

A=(1,2,3,2,4,5,4,6,4,7,4,2,1,8,1,9,1)

L'array B, che contiene il livello d'ogni elemento di A, è:

B=(0,1,2,1,2,3,2,3,2,3,2,1,0,1,0,1,0)

Oltre ad A ci occorrono le seguenti informazioni. Per ogni vertice v sia level(v) il livello del vertice v e indichiamo con l(v) e r(v) l'occorrenza rispettivamente più a sinistra e quella più a destra di v in A. Dunque l(v) e r(v) possono essere determinati in tempo O(1) se l'array A e il livello di ogni elemento di A sono dati.

Assumiamo, dunque, che level(v), l(v) e r(v) siano grandezze note per ogni vertice v.

Per rispondere ad una domanda su LCA in un tempo sequenziale O(1), possiamo restringere la nostra attenzione all'array B = level(A) ottenuto da A dalla sostituzione

di ogni vertice v con level(v). Questo ragionamento è giustificato dal seguente lemma.

LEMMA 1:

Dato un albero radicato T=(V, E), siano A, level(v), l(v) e r(v), per ogni v ?V, come definiti precedentemente. Siano u e v due vertici arbitrari distinti di T. Allora valgono le seguenti affermazioni:

1. *u è un antecedente di v se e solo se l(u)< l(v)< r(u).*
2. *u e v non sono confrontabili, vale a dire u non è un discendente di v e v non è un discendente di u, se e solo se o r(u) < l(v) o r(v) < l(u).*
3. *se r(u) < l(v) allora LCA(u, v) è il vertice con il livello minimo nell'intervallo [r(u),l(v)].*

DIMOSTRAZIONE:

1. Supponiamo che u *sia un antecedente di* v. *Allora, u è visitato prima di v, e, inoltre, il sottoalbero radicato in v è completamente visitato prima dell'ultima comparsa di v. Allora, l(u)<l(v)<r(u). Viceversa supponiamo che l(u)<l(v)<r(u), e che u non sia un antecedente di v. Poiché l(u)<l(v) il sottoalbero radicato in u è visitato completamente prima della prima visita di v. Allora r(u)<l(v) il che contraddice le ipotesi. Dunque, u è un antecedente di v il che prova il punto 1.*

2. Se u *e* v *non sono confrontabili allora o r(u)<l(v) o r(v)<l(u), vale a dire che una volta abbandonato il nodo e il sottoalbero in esso radicato si passa ad un altro sottoalbero.*

3. Supponiamo che r(u)<l(v). Si ha dunque che tutti i vertici il cui livello compare nell'intervallo [r(u), l(v)] sono anche i vertici che appaiono sul cammino tra u e v, o i loro discendenti. Dunque il vertice di minimo livello deve essere LCA di u e v.

Per il lemma 1 trovare il lowest common ancestor di due vertici u e v equivale a ricercare il vertice di minimo livello nell'array [r(u),l(v)]. Il problema e' quello di calcolare il minimo di $\{b_K, b_{K+1}, \ldots, b_j\}$, dato un qualsiasi intervallo [k, j], con $1 \le k \le j \le n$ in modo efficiente.

DESCRIZIONE DELL'ALGORITMO

Descriviamo un algoritmo con tempo O(logn) che calcola il minimo elemento in un generico intervallo. Sia B l'array di input di lunghezza $n = 2^l$. Cerchiamo di calcolare B in modo tale che, dati una coppia di indici i e j, dove $1 \le i \le j \le n$, possiamo determinare l'elemento minimo nel sottoarray $\{b_i, b_{i+1}, \ldots\ldots, b_j\}$ in un tempo sequenziale O(1). Un modo intuitivo di procedere potrebbe essere quello di costruire

un albero binario completo sugli elementi di B così che ogni nodo interno v di T contenga alcune informazioni dell'array determinate attraverso la visita del sottalbero radicato in v.

Dati due indici i e j possiamo determinare il minimo antecedente comune v delle foglie di T conoscendo b_i e b_j in un tempo sequenziale O(1) poiché T è un albero binario completo. Se la visita del sottalbero radicato in v corrisponde esattamente al sottoarray $\{b_i, b_{i+1}, \ldots, b_j\}$ è sufficiente immagazzinare il minimo elemento nel nodo v. In ogni caso, il sottoarray associato a v è tipicamente della forma $B_v=\{b_r,\ldots,b_i,\ldots,b_j,\ldots,b_s\}$, dove $r \leq i \leq j \leq s$. In questo caso, alcune informazioni addizionali sono necessarie per rispondere alla domanda sull'elemento minimo.

Siano u e w rispettivamente i figli sinistri e destri di v. Indichiamo con B_u e B_w i sottoarray associati ad u e w che rappresentano una partizione di B_v, dove, $B_u=\{b_r,\ldots,b_i,\ldots,b_p\}$ e $B_w=\{b_{p+1},\ldots, b_j, \ldots, b_s\}$, con $i \leq p < j$. L'elemento che cerchiamo è il minimo dei seguenti due elementi: il minimo del suffisso $\{b_i,\ldots,b_p\}$ di B_u e il minimo del prefisso $\{b_{p+1},\ldots, b_j\}$ di B_w. Dunque, per ogni nodo v, è sufficiente immagazzinare il suffisso minimo e il prefisso minimo del sottoarray associato a v.

Da ricordare che il prefisso minimo di B è l'insieme degli elementi dell'arrray $(c_1,c_2,\ldots,c_n)$ tali che $c_i=\min\{b_1,\ldots,b_i\}$, per $1 \leq i \leq n$. Analogamente possiamo definire il suffisso minimo di B. Chiaramente il prefisso minimo e il suffisso minimo di B possono essere calcolati in un tempo O(logn).

Il nostro algoritmo costruisce un albero binario completo le cui foglie sono gli elementi di B tali che ogni nodo interno v ha associato ad esso due array, P e S, rappresentanti, rispettivamente il prefisso minimo e il suffisso minimo definiti dalle foglie del sottoalbero radicato in v.

Input Array B di lunghezza $n = 2^l$, dove l è un intero positivo

Output Un albero binario completo con variabili ausiliarie P(h,j) e S(h,j), con $0=h=\log n$ e $1=j=n/2^h$, tali che P(h,j) e S(h,j) rappresentano, rispettivamente, il prefisso e il suffisso minimo del sottoarray definito dalle foglie del sottoalbero radicato in (h,j).

```
begin

for 1= j = n pardo
  Sia P(0,j) := B(j)
  Sia S(0,j) := B(j)
for h = 1 to log n do
  for 1= j = n/2h  pardo
```

```
      P(h, j) <- fusione[P(h-1, 2j-1), P(h-1, 2j)]
      S(h, j) <- fusione[S(h-1, 2j-1), S(h-1, 2j)]

end
```

L'algoritmo fusione e' il seguente:

Input Due array P(h-1,2j-1) e P(h-1, 2j). P_i indica il processore di indice i

Output L'array P(h,j)

```
begin

for i =1 to 2^(h-1) pardo
  P_i : v_i <- P(h-1,2j-1)_i
       P(h,j)_i <- v_i

for i =1 to 2^(h-1) pardo
  P_i : w_i <- P(h-1,2j-1)_2^(h-1)
       v_i <- P(h-1,2j)_i
       P(h,j)_(i+2^(h-1)) <- min(w_i,v_i)

end
```

Supponiamo di richiedere l'elemento di minimo valore all'interno di un intervallo [i,j]. Sia v= LCA(i,j) nell'albero binario costruito precedentemente. Poiché quest'albero binario è completo il nodo v può essere calcolato in un tempo sequenziale O(1). Siano u e w il figlio sinistro e il figlio destro di v. Il $\min\{a_i,a_{i+1},\ldots,a_j\}$ è il minimo dei due elementi: il suffisso minimo corrispondente ad i nell'array S di u e il prefisso minimo corrispondente a j nell'array P di w. Quindi, alla domanda sull'elemento minimo si risponde in un tempo costante.

LEMMA 2:
L'algoritmo 1 analizza l'array B dato in input in un tempo O(logn). Ogni domanda sulla ricerca dell'elemento minimo può essere soddisfatta in un tempo sequenziale O(1).

DIMOSTRAZIONE:
L'unico dettaglio che occorre chiarire è come viene eseguita l'operazione di fusione degli array P e S in tempo O(1). Sappiamo che la lunghezza di ogni array P(h,j) è pari a 2^h, 1= h = logn. La fusione di P(h-1, 2j-1) e P(h-1, 2j) in P(h, j) consiste nel copiare P(h-1, 2j-1) nella

prima metà di P(h,j) e rimpiazzare ogni elemento a *di P(h-1, 2j) con il minimo tra* a *e l'ultimo elemento di P(h-1,2j-1). Questa operazione di fusione può essere fatta in un tempo parallelo costante. Analogamente può essere effettuata la fusione di S(h-1, 2j-1) e S(h-1, 2j) in S(h, j). Dunque ogni livello dell'albero richiede un tempo parallelo O(1). Il tempo totale di esecuzione è O(logn).*

L'operazione di fusione richiede lettura concorrente, perché l'ultimo elemento di P(h-1,2j-1) deve essere confrontato con ciascuno degli elementi in P(h-1, 2j), questo ragionamento è valido anche per l'array S(h,j). Quindi, l'algoritmo 1 richiede un modello PRAM CREW.

Per determinare l'elemento minimo di un array di lunghezza n abbiamo un tempo pari a O(logn), dunque anche al problema del lowest common ancestor e' possibile applicare la stessa limitazione.

Ear Decomposition

INTRODUZIONE

I metodi di attraversamento di un grafo sono molto utili nello studio del grafo stesso perché, di fatto, ne inducono una decomposizione in un insieme di componenti semplici. La visita in profondità ed in ampiezza si sono rivelate utili per gestire varie problematiche relative allo studio dei grafi. Fin ora però non è nota una loro implementazione efficiente per il calcolo parallelo e per tal motivo, in questo ambito, si sono cercate delle strategie diverse e dei metodi di attraversamento alternativi che siano più efficientemente implementabili.Tra le varie tecniche introdotte parliamo di **ear decomposition**.

EAR DECOMPOSITION

Una **ear decomposition** è essenzialmente una partizione ordinata dell'insieme degli archi di un grafo in cammini semplici. Ricordiamo che, dato un grafo $G = (V, E)$, si definisce cammino semplice un cammino in cui dati due vertici i e j se $i \neq j$ allora $v_i \neq v_j$, se inoltre le due estremità del cammino coincidono si ha un ciclo. Più formalmente possiamo così definire un ear decomposition:

DEFINIZIONE 1: ***EAR DECOMPOSITION***

Sia $G = (V, E)$ *un grafo non orientato con* $|V| = n$ *e* $|E| = m$ *e sia* P_0 *un arbitrario ciclo semplice di* G. *Una ear decomposition di* G *inizia con* P_0 *ed è una partizione ordinata dell'insieme di archi* $E = P_0 \cup P_1 \cup \ldots \cup P_k$ *tale che per ogni* i *con* $1 \leq i \leq k$ P_i *è un cammino semplice i cui vertici iniziali e finali appartengono a* $P_0 \cup \ldots \cup P_{i-1}$ *e nessun altro suo vertice interno vi appartiene. Ogni cammino semplice* P_i *è detto ear. Se inoltre per ogni* $i > 0$ P_i *non è un ciclo la decomposizione è detta* ear decomposition aperta.

Consideriamo ad esempio il grafo in figura 1, una possibile ear decomposition è quella mostrata dove ogni arco che appartiene a P_i è etichettato con i. Notiamo che non è una decomposizione aperta poiché P_2 è un ciclo.

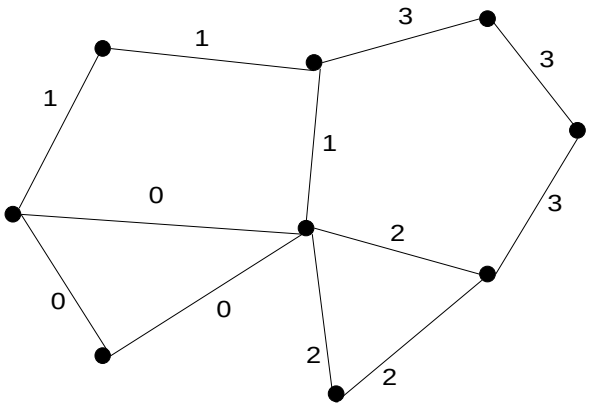


Fig.1: esempio di grafo con relativa ear decomposition non aperta

In generale una ear decomposition non è unica. Possiamo caratterizzare i grafi che hanno una ear decomposition attraverso il teorema 1. Prima di enunciarlo ricordiamo alcuni concetti relativi alla teoria dei grafi.

DEFINIZIONE 2: ***GRAFO CONNESSO***

In un grafo non orientato G due vertici v_0 e v_1 sono connessi se esiste un cammino in G che li collega ovvero che va da v_0 a v_1. Un grafo non orientato è connesso se, per ogni coppia di vertici distinti v_i e v_j, esiste un cammino da v_i a v_j.

DEFINIZIONE 3: ***PUNTO DI ARTICOLAZIONE***

Un punto di articolazione è un vertice v di G tale che l'eliminazione di v, insieme con tutti i lati incidenti su v, produce un grafo G' che ha almeno due componenti connesse (una componente connessa di un grafo non orientato è un sottografo massimo connesso).

DEFINIZIONE 4: PONTE

Un ponte è un arco di G la cui rimozione disconnette il grafo. Un ponte è un arco che unisce due punti di articolazione

DEFINIZIONE 5: ***GRAFO BICONNESSO***

Un grafo biconnesso è un grafo connesso che non ha punti di articolazione, questo implica che esistono almeno due cammini disgiunti che collegano due vertici distinti del grafo.

TEOREMA 1**:** *Un grafo non orientato $G=(V,E)$ ha una ear decomposition se e solo se non ha ponti. Il grafo G ha una ear decomposition aperta se e solo se è biconnesso.*

Consideriamo ad esempio il grafo in figura 2

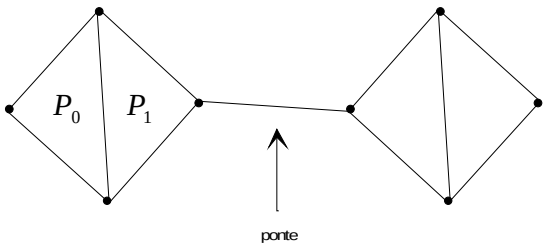


Fig.2: esempio di grafo che non ammette ear decomposition

Dopo aver calcolato P_0 e P_1 non sappiamo più andare avanti nella decomposizione, infatti, il grafo presenta un ponte quindi non ha una ear decomposition. Mostriamo, in figura 3, un grafo che non è biconnesso che non ha ponti ed ha una ear decomposition non aperta.

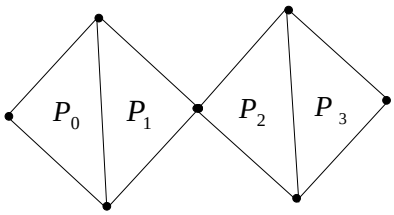


Fig.3: esempio di grafo che ammette una ear decomposition non aperta

DESCRIZIONE DELL'ALGORITMO

Calcoliamo la ear decomposition di un grafo che la ammetta.

Supponiamo che il grafo G abbia un ear decomposition aperta $E = P_0 \cup P_1 \cup \ldots \cup P_k$, in tal caso sappiamo che P_0 è un ciclo semplice, P_1 è un cammino semplice che ha due estremi distinti in P_0, P_2 è un cammino semplice che ha due estremi distinti in $P_0 \cup P_1$ e così via. E' chiaro che questo tipo di decomposizione porta ad un insieme di cicli, se rimuoviamo un arco da ogni ear otteniamo uno spanning tree per G. D'altronde se consideriamo uno spanning tree per G e prendiamo tutti gli archi di G non nello spanning tree e li aggiungiamo allo spanning tree otteniamo un sistema

completo di cicli fondamentali[1] di G. Da questa considerazione possiamo derivare un metodo per calcolare una ear decomposition in un grafo che la ammetta.

Inizialmente si calcola uno spanning tree T per il grafo G. Una prima idea è quella di associare un ear P_e ad ogni arco e di G non in T. Sia cioè C_e il ciclo fondamentale indotto da e potremmo porre $P_e = C_e$ ma questo non è corretto perché qualche arco in C_e potrebbe essere in un qualche altro ciclo fondamentale quindi avremmo delle sovrapposizioni di archi in ear diverse e questo non è permesso. Potremmo, per ogni arco non dell'albero, considerare una ear a meno però di sovrapposizioni. Per risolvere questo problema possiamo disgregare il ciclo C_e in cammini così che ogni cammino appartenga ad ear diverse, per far questo etichettiamo gli archi del grafo con opportune etichette la cui lettura fornirà la decomposizione cercata. Da queste considerazioni deriviamo che le ear trovate non sono più degli archi non dell'albero.

L'algoritmo per trovare un ear decomposition in un grafo che la ammetta lavora in questo modo:

1. Trovare uno spanning tree[2] T di G.
2. Radicare T in un arbitrario vertice r e calcolare $level(v)$ e $p(v)$ per ogni vertice $v \neq r$ dove $level(v)$ e $p(v)$ sono rispettivamente livello e padre di v[3].
3. Per ogni arco $e = (u,v)$ non in T calcolare $lca(e) = lca(u,v)$ (lowest common ancestor) e porre $level(e) = level(lca(e))$. Etichettare e con $label(e) = (level(e), s(e))$ dove $s(e)$ indica l'indice dell'arco e in una fissata enumerazione degli archi di G con $1 \leq s(e) \leq m$.
4.
   4.1 Per ogni arco e non in T calcolare il corrispondente ciclo indotto, sia esso C_e.
   4.2 Per ogni arco g in T considerare tutti i cicli indotti che lo contengono.
   4.3 Etichettare ogni arco g in T con
   $label(g) = \min\{label(e) | e \text{ è un arco non in T e } C_e \text{contiene g}\}$

[1] Vedere Appendice A: cicli fondamentali pag. 38
[2] Vedere tesina su Spanning tree
[3] Per quanto riguarda la definizione ed il calcolo di level(v) e p(v) vedere Capitolo 1 pag. 10

5. Per ogni arco e non in T calcolare l'insieme $P_e = \{e\} \cup \{g \in T \mid label(g) = label(e)\}$. Ordinare[4] gli insiemi P_e in base a $label(e)$.

Prima di dimostrare la correttezza dell'algoritmo proveremo una proprietà che garantisce la correttezza dell'etichettatura degli archi dell'albero.

LEMMA 1:

Sia T un albero ricoprente per il grafo G(V,E) radicato. Per ogni arco $g \in E$ sia label(g) la funzione definita nell'algoritmo. Dato un arco e non dell'albero sia $P_e = \{e\} \cup \{g \in T \mid label(g) = label(e)\}$, P_e è un cammino semplice o un ciclo. Ogni arco nell'albero appartiene ad uno solo di questi cammini.

DIMOSTRAZIONE:

Sia $e = (u,v)$ un arco non in T. Consideriamo il cammino Q in T che collega v a $lca(u,v)$ e sia $v \neq lca(u,v)$. Sia $h = (x, p(x))$ il primo arco in Q tale che $label(h) \neq label(e)$ (vedere figura 4). La diversità tra $label(h)$ e $label(e)$ implica l'esistenza di un arco non in T, sia esso $l = (z,x)$, che ha etichetta più piccola di quella di e. Quindi nessuno tra gli archi in Q compresi tra x e $lca(u,v)$ appartiene a P_e. Possiamo fare le stesse considerazioni se prendiamo il cammino che collega u a $lca(u,v)$. Da ciò segue che P_e è un cammino semplice. Poiché le etichette degli archi non in T sono distinte ogni arco in T appartiene ad esattamente un cammino semplice.

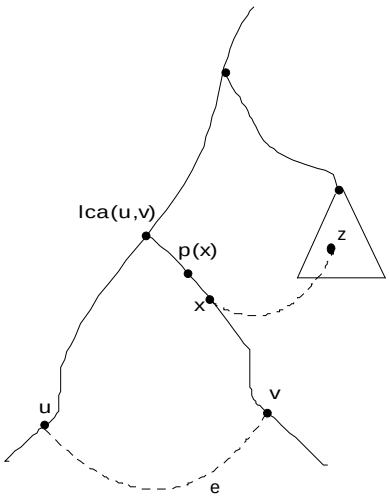


Fig.4: illustrazione della dimostrazione del lemma 1

[4] Per l'ordinamento vedere la tesina sugli Ordinamenti

***TEOREMA 2:** CORRETEZZA DELL'ALGORITMO*

Sia $G=(V,E)$ un grafo senza ponti. L'algoritmo calcola correttamente una ear decomposition di G.

DIMOSTRAZIONE:

Come si può vedere nel lemma 1 ogni P_e è un cammino semplice, si dimostra ora che per ogni cammino i suoi punti terminali appartengono ad una qualche ear $P_{e'}$ tale che label(e')<label(e) e che nessun vertice interno appartiene a tale ear.

Sia e_0 l'arco non in T che ha etichetta minima, quindi $label(e_0)=(0,s(e_0))$ dove $s(e_0)$ è il valore minimo tra tutti gli archi e tali che lca(e) è la radice. Questo arco deve esistere perché altrimenti il grafo G avrebbe avuto un ponte incidente sulla radice e questo non è possibile. La ear P_{e_0}, che corrisponde a e_0, è il ciclo base C_{e_0}.

Sia e un arbitrario arco non in T tale che label(e)>label(e_0). La dimostrazione del lemma 1 indica che o P_e è un ciclo base C_e oppure ciascuno dei punti terminali del cammino appartiene ad un qualche arco g tale che label(g)<label(e). Nell'ultimo caso g ha una etichetta più piccola e quindi il lemma è provato. Se invece $P_e=C_e$ sia v=lca(e). Se v è la radice allora il lemma è provato perché C_e interseca la ear C_{e_0} alla radice. Altrimenti consideriamo l'arco (v,p(v)) che deve appartenere ad un qualche C_f per un qualche f arco non in T, se tal arco non esiste allora (v, p(v)) è un ponte il che non è possibile. Il livello di lca(f) deve essere più piccolo del livello di v quindi v appartiene ad un ear la cui label è più piccola della label di e. Quindi il lemma è provato.

Il tempo previsto per l'algoritmo, considerando una PRAM CRCW, può essere così calcolato:

1. Ricerca dello spanning tree : si può ad esempio usare l'algoritmo di Sollin[5] per il minimo spanning tree con m processori e tempo $O(\log n)$ implementato su una PRAM CRCW
2. Numerazione degli archi : somme prefisse $O(\log n)$
3. Calcolo del livello per ogni vertice: $O(\log n)$
4. Calcolo del lowest common ancestor : $O(\log n)$
5. Etichettatura degli archi non dell'albero : $O(1)$

[5] Per l'algoritmo di Sollin si fa riferimento al testo : Jaja J., An introduction to parallel algorithms, Addison Wesley Publishing company, pag. 223 e seg., per la trattazione sullo spanning tree vedere la tesina Spanning tree

6. Etichettatura degli archi dell'albero : ricerca del minimo $O(\log n)$
7. Fase di ordinamento e produzione della decomposizione : $O(\log n)$

Possiamo dire che il tempo dell'algoritmo presentato è $O(\log n)$ con costo $O(m \log n)$

ESEMPI O

Calcoliamo la ear decomposition del grafo G in figura 5.

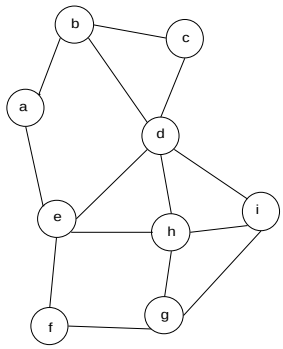


Fig.5: grafo che ammette una ear decomposition aperta

Calcoliamo uno spanning tree per il grafo che mostriamo in figura 6. Nella figura segnaliamo con il tratteggio anche gli archi non nell'albero ricoprente. Numeriamo inoltre gli archi, usando ad esempio le somme prefisse[6], in modo progressivo. Le etichette numeriche poste sugli archi indicano la posizione dell'arco rispetto alla numerazione fissata. Indichiamo in figura, con la numerazione verticale, anche i livelli dei vertici.

[6] Vedere la tesina sulle Somme prefisse

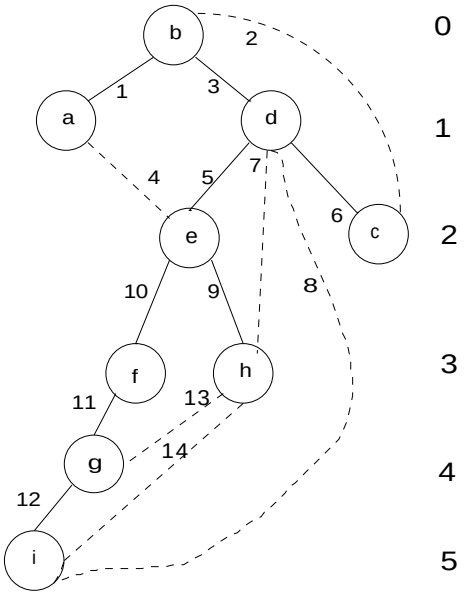


Fig.6: albero ricoprente del grafo G .Si indica inoltre il livello dei vertici

Caratterizziamo gli archi non dell'albero con $label(e) = (level(e), s(e))$.

Consideriamo l'arco 2, incidente sui vertici b e c. Calcoliamo $lca(2) = lca(b,c) = b$ b si trova a livello 0 dunque etichettiamo l'arco 2 con la coppia 0,2 cioè $label(2) = (level(2),2) = (0,2)$. Ripetiamo questo ragionamento per tutti gli archi non dell'albero, otteniamo la seguente etichettatura:

ARCHI e NON DELL'ALBERO	LABEL(e)
2	(0,2)
4	(0,4)
7	(1,7)
8	(1,8)
13	(2,13)
14	(2,14)

Si etichettano ora gli archi dell'albero, passo 4. Per ogni arco dell'albero consideriamo tutti i cicli fondamentali indotti da archi non dell'albero che lo contengono e lo etichettiamo con la label più piccola dell'arco non dell'albero corrispondente ad uno dei cicli fondamentali. Vediamo il procedimento per ogni arco nell'albero (sottolineiamo l'arco non dell'albero che genera il ciclo fondamentale[7]):

[7] Ogni ciclo fondamentale è dato come sequenza di archi

ARCHI g DELL'ALBERO	CICLI FONDAMENTALI CONTENTI g	LABEL(g)
1	1 3 4 5	(0,4)
3	1 3 4 5 2 6 3	(0,2)
5	5 4 3 1 5 7 9 8 12 11 10 5	(0,4)
6	2 3 6	(0,2)
9	13 11 10 9 7 5 9 9 10 11 12 14	(1,7)
10	8 12 11 10 5 13 9 10 11 14 12 11 10 9	(1,8)
11	13 11 10 9 8 12 11 10 5 14 12 11 10 9	(1,8)
12	14 12 11 10 9 8 12 11 10 5	(1,8)

Consideriamo ad esempio l'arco 3, esso è contenuto in due cicli fondamentali 1 3 4 5 e 2 6 3 indotti rispettivamente dagli archi 4 e 2 le cui label sono (0,4) e (0,2) quindi l'arco 3 prende come label la coppia (0,2). La doppia etichettatura serve per garantire la possibilità di ordinare ogni coppia in modo tale che qualora, come in questo caso, il livello sia uguale, possiamo definire la coppia più piccola sempre controllando l'arco. Ogni arco ha un'etichetta, ogni etichetta rappresenta un ciclo o un cammino semplice cioè una ear, dunque ordiniamo le etichette e associamo ad ogni etichetta il nome di una ear, otteniamo la seguente enumerazione:

(0,2)	P_0
(0,4)	P_1
(1,7)	P_2
(1,8)	P_3
(2,13)	P_4
(2,14)	P_5

L'ultimo passo dell'algoritmo consiste nel definire gli archi in ogni ear in base alla regola $P_e = \{e\} \cup \{g \in T \mid label(g) = label(e)\}$, otteniamo la seguente decomposizione:

$P_0 = \{2,3,6\}$
$P_1 = \{4,5,1\}$
$P_2 = \{7,9\}$
$P_3 = \{8,10,11,12\}$
$P_4 = \{13\}$
$P_5 = \{14\}$

In figura 7 presentiamo una visualizzazione grafica della decomposizione trovata:

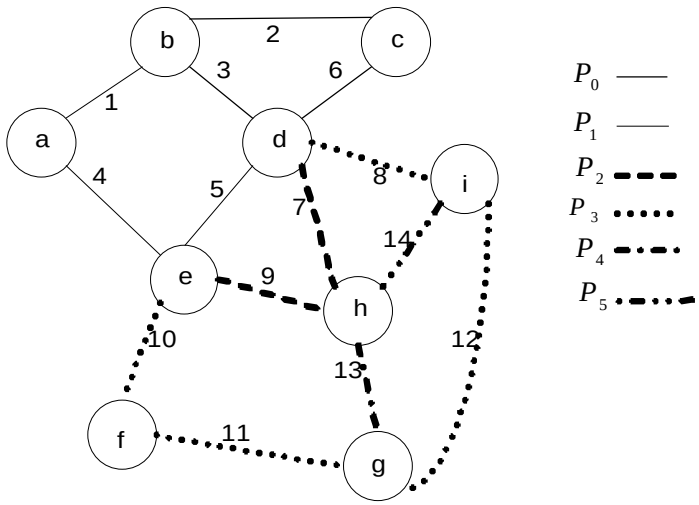


Fig.7: ear decomposition del grafo

APPENDICE A: CICLI FONDAMENTALI

Iniziamo con il richiamare alcune definizioni di base[8]:

DEFINIZIONE 1A: COMBINBAZIONE LINEARE

Sia V uno spazio vettoriale su F e $v_1,\ldots,v_n \in V$. Un elemento nella forma $a_1v_1+\ldots+a_nv_n$, dove tutti gli $a_i \in F$, si chiama combinazione lineare su F di $v_1,\ldots,v_n$.

DEFINIZIONE 2A: VETTORI LINEARMENTE DIPENDENTI

Se V è uno spazio vettoriale su F e $v_1,\ldots,v_n$ appartengono a V, si dice che $v_1,\ldots,v_n$ sono linearmente dipendenti su F se esistono elementi $a_1,\ldots,a_2 \in F$, non tutti nulli, tali che $a_1v_1+\ldots+a_nv_n=0$. Se i vettori non sono linearmente dipendenti si dicono linearmente indipendenti.

DEFINIZIONE 3A: BASE DI UNO SPAZIO VETTORIALE

Un sottoinsieme S di uno spazio vettoriale V si chiama base di V se gli elementi di S sono linearmente indipendenti ed inoltre S genera V cioè V = L(S) dove L(S) è la chiusura lineare di S (insieme delle combinazioni lineari di sottoinsiemi finiti di S).

Dato un campo F ed uno spazio lineare $V \subseteq F^m$ l'insieme di vettori $\left\{\underline{a}^1,\ldots,\underline{a}^n\right\} \subseteq V$ è linearmente indipendente se $\forall j\ 1 \le j \le n$ $\underline{a}^j$ non dipende linearmente da $\left\{\underline{a}^1,\ldots,\underline{a}^n\right\} - \left\{\underline{a}^j\right\}$. Sia $\underline{A} = \left(\left(a_i^j\right)\right)_{i\in I,\, j\in J}$ una matrice con $a_i^j \in F$ definiamo $\mathrm{V}_F(A)$ la chiusura lineare dell'insieme dei vettori colonna. Sia B un sottoinsieme di J, insieme degli indici di colonna, tale che $\underline{B} = \left\{\underline{a}^h : h \in B\right\}$ sia una base dello spazio $\mathrm{V}_F(A)$, il rango di $\underline{A}$ coincide con il massimo numero di vettori colonna (riga) linearmente indipendenti ed è pari alla cardinalità di B. Possiamo definire altri due spazi vettoriali $\mathrm{Y}_F(\underline{A})$ e $\mathrm{Z}_F(\underline{A})$ rispettivamente la chiusura lineare dell'insieme dei vettori riga e lo spazio dei nulli ovvero l'insieme delle soluzioni del sistema di equazioni omogenee $\underline{Az} = \underline{0}$. Data la base B per lo spazio $\mathrm{V}_F(A)$ è possibile trovare una base anche per gli spazi $\mathrm{Y}_F(\underline{A})$ e $\mathrm{Z}_F(\underline{A})$.

Dato un grafo $G(V,E)$, che supponiamo connesso, consideriamo la sua matrice di incidenza $\underline{A}$. Possiamo interpretare gli elementi della matrice di incidenza come elementi di un qualsiasi campo F. Questa matrice ha una base la cui dimensione è

[8] Per le definizioni algebriche si rimanda al testo: I.N. Herstein, Algebra, Editori riuniti, ottobre 1995

anche il suo rango. La base della matrice è formata da vettori colonna (o riga) linearmente indipendenti, scegliere vettori colonna è equivalente a scegliere archi del grafo. Dato il grafo si vuol calcolare la base dello spazio vettoriale definito dalle colonne della matrice, ma, nel caso del grafo, una base è costituita dagli archi che formano un albero di copertura per il grafo stesso, infatti le colonne corrispondenti sono linearmente indipendenti. Consideriamo infatti la situazione in figura 1a e supponiamo che quello in figura sia l'albero ricoprente di un certo grafo G.

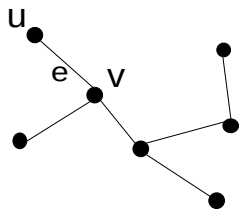


Fig. 1a: albero ricoprente di G

Il vettore colonna $\underline{a}^e$, nella matrice di incidenza per il grafo G, presenta un 1 in corrispondenza dei vertici u e v ed in particolare nessun vettore corrispondente ad un arco nell'albero presenta 1 in corrispondenza del vertice u ciò implica che sia impossibile esprimere $\underline{a}^e$ come combinazione lineare dei vettori relativi agli altri archi dell'albero. Questo è vero anche per le altre colonne relative all'albero stesso. Un ciclo invece è un insieme dipendente minimale, infatti tolto un arco da un ciclo otteniamo un cammino come quello mostrato in figura 1a i cui archi formano un insieme indipendente. Un ciclo è un insieme dipendente perché siamo in grado di esprimere un arco dell'insieme in funzione di altri archi dell'insieme stesso. Vediamo ad esempio il ciclo semplice in figura 2a:

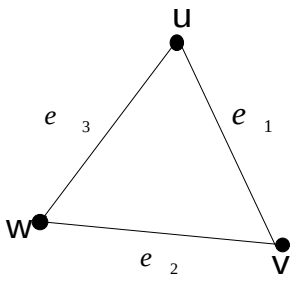


Fig. 2a: ciclo semplice

Poiché e_1 ed e_2 hanno in comune v si ha che $\underline{a}^{e_1} \oplus \underline{a}^{e_2} = \underline{a}^{e_3}$ ma allora $\underline{a}^{e_1} \oplus \underline{a}^{e_2} \oplus \underline{a}^{e_3} = 0$ quindi effettivamente le colonne formano un insieme linearmente dipendente. Date queste premesse è possibile dimostrare che se $\underline{B} = \left\{\underline{a}^h : h \in B\right\}$ è una base di $\underline{A}$ allora B è un albero di copertura per il grafo G ed inoltre una base per $Z_F(\underline{A})$ è formata da un insieme di vettori riga il cui supporto[9] sono dei cicli. Dato il grafo $G(V,E)$ ed una base B della sua matrice di incidenza $\underline{A}$, se e è un elemento di E che non appartiene a B allora l'insieme $B \cup \{e\}$ contiene esattamente un ciclo che indichiamo con C_e, la famiglia $\{C_e : e \in E - B\}$ prende il nome di sistema di cicli fondamentali associati a B.

Dato un grafo e calcolato un suo albero ricoprente possiamo partizionare gli archi del grafo in archi dell'albero e archi non dell'albero. Se aggiungiamo un solo arco all'albero si forma un ciclo infatti se n sono i vertici del grafo n-1 sono gli archi dell'albero, aggiungendo un arco si hanno n archi e quindi necessariamente si è creato un ciclo. Ogni arco non dell'albero se aggiunto all'albero genera un ciclo detto appunto ciclo fondamentale.

[9] Chiamiamo supporto di un vettore v di F^m l'insieme degli indici j$1 \le j \le m$ cui corrispondono componenti di v diverse da zero.

Riferimenti bibliografici

[1] : Tesina Spanning Tree

[2] : Tesina Somme prefisse e Salto del puntatore

[3] : Tesina Ordinamenti

[4] : J. Jaja, An introduction to parallel algorithms, Addison Wesley Publishing Company

[5] : I.N. Herstein, Algebra, Editori riuniti, ottobre 1995