Analisi degli Algoritmi

Alberto Tibaldi

16 ottobre 2010

Indice

Capitolo 1

Teoria dei Grafi

Molto spesso, in molti ambiti scientifici (elettronica, fisica, informatica...) è necessario introdurre modelli matematici in grado di rappresentare un certo numero di fenomenologie, di comportamenti di un dato soggetto: dato un modello dotato di determinate proprietà, se si riuscisse a ricondurre una fenomenologia ad esso, si potrebbero sfruttare le proprietà del modello per approfondire o semplificare lo studio della fenomenologia stesso.

Caso eclatante di modello matematico creato per questi motivi è quello di grafo: un grafo viene utilizzato per modellizzare strutture astratte (come ad esempio algoritmi, o strutture dati), o anche entità molto più concrete come un circuito elettronico, reti di telecomunicazioni, ed altro.

1.1 Definizioni Fondamentali

Al fine di comprendere meglio i grafi, introduciamo un certo numero di definizioni atte a chiarire le idee che verranno poi presentate.

1.1.1 Definizione di Grafo

Un grafo (graph) è una struttura astratta, definibile mediante una coppia di insiemi: gli elementi, detti "nodi" o "vertici", ed i lati, ossia i "segmenti" che "uniscono" i vari vertici. Si vuole evidenziare il fatto che la rappresentazione spesso utilizzata (il "pallogramma", ossia un insieme di pallini, o cerchi, collegati tra loro mediante segmenti rettilinei o curvi) può sì dare l'idea di cosa sia un grafo, ma di sicuro non si può dire che "un grafo sia un insieme di pallini e cerchi": la definizione più corretta è quella già accennata, e che ora sarà rivista.

Dato un insieme di elementi, detti "vertici" o "nodi", V, ed un altro insieme di elementi detti "lati", E, rappresentante coppie di vertici, un grafo G si definisce come:

$$G(V, E)$$

Si tratta ossia di una coppia ordinata di due insiemi (collezione di due oggetti nella quale si possano distinguere i due componenti, o membri, che la formano): l'insieme degli elementi del grafo, e l'insieme delle coppie realizzabili con gli elementi del grafo stesso. Un nodo dunque sarà un'entità includente, in qualche modo, informazioni di diverso tipo (a seconda di cosa dovrà modellizzare il nostro grafo); un lato, invece, sarà semplicemente un contenitore di due nodi, in modo da rappresentare il fatto che esiste una relazione tra due elementi del grafo.

Esistono alcuni casi particolari di grafo, che ora verranno presentati.

- Un cappio (self-loop) avviene quando un nodo è collegato a sè stesso da un lato; detto in un altro modo, un lato contiene due volte lo stesso nodo (ossia inizia e termina sullo stesso vertice, se vogliamo dirlo in modo più visibile);

- Un multigrafo è un grafo in cui può esistere più di un lato tra gli stessi due nodi; un grafo semplice è, dualmente, un grafo dove esista al più un lato comprendente due nodi;

- La cardinalità (ossia il numero di elementi appartenenti all'insieme) degli insiemi di vertici (V) e di lati (E) verrà indicata rispettivamente con $|V|$ e $|E|$;

- Dato un grafo semplice, esso si definisce "completo" se esistono tutti i possibili lati tra i vari nodi; altro modo di dire ciò, è il fatto che il numero dei lati è pari alle combinazioni semplici di ordine 2 di tutti i vertici:

  $$|E| = \frac{|V|\,(|V| - 1)}{2}$$

  Dato $|V| = n$, ossia dati n vertici, il grafo completo di ordine n si definisce mediante la notazione K_n: ad esempio K_3 sarà il grafo completo con 3 vertici, K_4 con 4, e così via.

- Un grafo è detto orientato o diretto (directed graph) se i sottoinsiemi di ordine 2, ossia i lati, sono ordinati. Come già detto prima parlando di

'coppie ordinate', ordinati significa "che si possono distinguere il primo dal secondo"; se dunque potessimo distinguere il primo nodo del lato dal secondo, potremmo dire che il grafo permette il collegamento dal primo al secondo nodo, ma non dal secondo al primo nodo.

- Si dice che un lato sia "incidente" nel nodo quando il nodo sia contenuto nella coppia "lato". Dualmente, si definisce "grado" (degree) del nodo il numero di lati su di esso incidenti. Esiste un'estensione del concetto appena presentato per quanto riguarda i grafi orientati: si definiscono, per ciascun nodo, due gradi:
  - Grado di uscita (out-degree): numero di lati incidenti in uscita, ossia diretti verso un altro nodo;
  - Grado di ingresso (in-degree): numero di lati incidenti in ingresso, ossia da un altro nodo diretti verso il nodo interessato;
- Due nodi di un grafo si dicono adiacenti (o contigui) se un lati li collega direttamente; per quanto riguarda i grafi orientati, non è detto che i nodi siano entrambi uno adiacente all'altro (a meno che non esistano entrambi i lati): dati due nodi di un lato, rispettivamente 1 e 2, la relazione di adiacenza, è valida tra 1 e 2, ma non tra 2 e 1.
- Un grafo si dice "pesato" (weighted) se per ciascun lato è associato un valore, detto "peso".
- Un percorso o cammino (path) di un grafo di lunghezza k è una sequenza di $k+1$ vertici tra di loro adiacenti. k si dice "lunghezza" del percorso ed è il numero di lati coinvolti in esso. Un percorso si dice "semplice" se un nodo non appare più di una volta nel percorso. In ambito di grafi orientati, un nodo si dice "raggiungibile" da un altro se esiste un cammino che dal primo possa giungere al secondo (cosa non garantita, per il fatto che la relazione di adiacenza non è assicurata tra nodi con lo stesso lato incidente su di entrambi);
- La "distanza" tra due nodi è la lunghezza del percorso minimo (ossia il percorso più breve) che si possa stabilire tra due nodi;
- Il "diametro" è il duale della precedente espressione: è la massima distanza tra due nodi del grafo, ossia la lunghezza del percorso più lungo effettuabile per raggiungere un nodo a partire da un altro;
- Si definisce "circuito" (circuit) o ciclo (cycle) un percorso con nodo iniziale coincidente con il nodo finale. Si importa la definizione di

"semplicità", nel senso che un ciclo si dice "semplice" se i vertici non compaiono più di una volta.

- Un grafo non orientato (e se orientato viene considerato il grafo non orientato uguale ad esso, ossia vengono "introdotti" anche i collegamenti inversi tra i nodi) è detto "connesso" se esiste un percorso in grado di collegare due nodi qualsiasi appartenenti al grafo;

- Un grafo nella sua forma non orientata (ottenuta come prima spiegato) si dice "foresta" se è aciclico, ossia se non esistono cicli all'interno di esso (duale della definizione di "ciclo").

- Un grafo aciclico, ossia una foresta, è detto "albero" se è una foresta connessa, ossia se, oltre ad essere aciclico, è connesso;

- Un albero è detto "radicato" (rooted tree) se si sceglie come "radice" uno qualsiasi dei suoi nodi. Per la radice si considera il "nodo di livello più basso", dove per "livello' si intende la distanza dalla radice stessa, per l'appunto (nodo di livello 0).

- Il grafo $G(V', E')$ è un "sottografo" di $G(V, E)$ se:

$$V' \subseteq V; \quad E' \subseteq E$$

  Se cioè i due insiemi costituenti il nuovo grafo (quello dei vertici e quello dei lati) sono entrambi sottoinsiemi di quelli costituenti il grafo di origine.

- $G(V', E')$ viene detto "albero ricoprente" (spanning tree) se è un sottografo aciclico e connesso dotato di tutti i nodi del grafo $G(V, E)$.

- Generalizzazione della precedente definizione è quella per cui si elimina la connessione tra le ipotesi, ottenendo così una "foresta ricoprente" (spanning forest), ossia un sottografo in cui sono presenti tutti i nodi ma non connesso.

1.2 Rappresentazioni dei grafi

Finora abbiamo presentato modi prevalentemente grafici per la rappresentazione dei grafi: come detto nell'introduzione, spesso si parla di grafi come di insiemi di pallini collegati tra di loro mediante segmenti rettilinei o curvi (per quanto la definizione formale sia un'altra). Una definizione del genere

non è però molto utile: dal momento che, come vedremo, i grafi sono dotati di proprietà utilissime per la risoluzione di problemi di vario genere, potrebbe essere utile disporre di rappresentazioni utili di grafi, in modo ad esempio di poterli importare in calcolatori elettronici, o di poter introdurre su di essi un maggior formalismo matematico rispetto a quello che per ora siamo in grado di utilizzare.

Considereremo, nella trattazione, sostanzialmente due rappresentazioni dei grafi: mediante lo studio dell' "adiacenza" tra i nodi, e dell' "incidenza" dei lati, evidenziando alcuni aspetti applicativi delle rappresentazioni

1.2.1 Matrici di Adiacenza

Una rappresentazione dei grafi consiste nel definire una matrice in grado di mostrare se due nodi siano uno adiacente all'altro: da qua il nome "matrice di adiacenza". La matrice di adiacenza è una matrice quadrata, dove il numero di righe e di colonne coincide con il numero di nodi presenti nel grafico. Essa si definisce così:

$$a_{i,j} \triangleq \begin{cases} 1, (V_i; V_j) \in E \\ 0, (V_i; V_j) \notin E \end{cases}$$

Ossia, in parole povere, si introduce un '1' se il nodo i-esimo è adiacente al nodo j-esimo, e 0 altrimenti. Per quanto riguarda i grafi orientati, è sufficiente ricordare che due nodi possono (e spesso sono) adiacenti solo in un verso (poichè i lati sono coppie ordinate, e quindi solo il "primo" nodo del lato va verso il "secondo"), quindi vi sarà un "1" sul valore (i, j) della matrice, ma non sul valore (j, i). Se per un grafo non orientato si può dunque dire che la matrice sia simmetrica, non si possono trovare caratteristiche simili per le matrici di adiacenza associate a grafi orientati.

Si può introdurre un'eventuale estensione per grafi pesati, dicendo che:

$$a_{i,j} \triangleq \begin{cases} W(V_i; V_j), (V_i; V_j) \in E \\ c, (V_i; V_j) \notin E \end{cases}$$

Si sceglie di introdurre una costante c anzichè 0 o altro poichè essa deve rappresentare il non collegamento compatibilmente con le dimensioni che si utilizzano come "pesi": se si trattasse di tempi di percorrenza, $c = \infty$: bisognerebbe idealmente percorrere "infinita" strada per arrivare verso quel nodo a partire da un altro; dualmente potrebbe avere altri significati fisici per i quali ha senso usare $c = 0$, o altro.

Come si può immaginare dal fatto che la matrice contenga esclusivamente riferimenti ai vertici, e sia quadrata, si può dire che lo spazio occupato da essa abbia l'ordine di grandezza del numero di elementi, e quindi sia $\Theta(V^2)$

Liste di adiacenza

Le relazioni di adiacenza tra nodi si possono anche rappresentare con una struttura dati astratta diversa dalla matrice appena presentata: le liste). Nella fattispecie, un'idea potrebbe essere quella di strutturare nel seguente modo una lista multipla: nella lista principale si introducono, in "catena", tutti i nodi del grafo; in ciascuna sottolista, collegata dunque ad un nodo del grafo, si introducono concatenati tra loro tutti i nodi adiacenti al nodo della lista principale. Eventualmente, per grafi pesati, si può introdurre nelle liste un peso.

1.3 Chiusura transitiva di un grafo

In teoria dei grafi, la chiusura transitiva ha un significato ben particolare (e notevoli utilità pratiche): prima di esporle, presentiamo una definizione. Si definisce chiusura transitiva (transitive closure) $C_T(V'; E')$ di un grafo $G(V; E)$ un grafo dalle seguenti proprietà:

- $V' \equiv V$
- $(V_i; V_j) \in E'$, se esiste un percorso da V_i a V_j.

Cosa significa ciò? La chiusura transitiva di un grafo deve avere gli stessi vertici del grafo stesso, e come nodi deve avere tutti quelli del grafo precedente, più nodi aggiuntivi che si introducono se tra due generici nodi V_i e V_j esiste un percorso (non importa da quanti lati esso sia formato).

Questo tipo di definizione è utilissimo perchè permette di risolvere un problema molto importante: una volta costruita la chiusura transitiva di un grafo, è possibile determinare "istantaneamente" se due nodi siano o meno collegati tra di loro nel grafo di partenza. In questo modo, l'operazione di "ricerca del collegamento" diventa a prezzo costante ($O(1)$): è sufficiente posizionarsi sul nodo del grafo, e verificare mediante un controllo se esista o meno collegamento con l'altro nodo interessato.

Esistono diversi algoritmi in grado di costruire la chiusura transitiva di un grafo; ci proponiamo di presentarne due dei più famosi.

1.3.1 Algoritmo 1

Un grafo, come abbiamo detto in precedenza, si può rappresentare mediante la sua matrice di adiacenza; data la matrice di adiacenza A associata ad un grafo $G(V; E)$, essa si può trattare di fatto (a meno che i grafi non siano

pesati, caso che non consideriamo) come una matrice booleana, ossia di variabili binarie (1 = TRUE, 0 = FALSE): ciò che si può fare è dunque utilizzare, anzichè le operazioni di somma e prodotto algebriche, nel calcolo matriciale, le operazioni di somma logica (OR) e di prodotto logico (AND).

Si può dimostrare che la chiusura transitiva di un grafo sia semplicemente il quadrato della matrice di adiacenza, trattata mediante le operazioni logiche:

$$A^2 = A \times A$$

In questo modo, ciascun elemento della matrice A^2 si costruirà come:

$$a^2_{i,j} = (A_{i,1} AND a_{1,j}) OR (a_{i,2} AND a_{2,j}) OR ...$$

Ciò è in grado di verificare se esista un percorso di lunghezza 2 tra il nodo V_i e V_j: in tal caso, $a^2_{i,j} = 1$. E se volessimo ottenere percorsi di lunghezza maggiore? Beh, semplice! E' sufficiente calcolare le potenze superiori di matrici quadrate! Il teorema è ancora valido, se viene applicato iterativamente: A^n è la matrice di chiusura transitiva in grado di presentare tutti i percorsi di lunghezza n.

Ciascuna matrice $A^{(i)}$ contiene tutti i percorsi di lunghezza i, e non quelli inferiori o superiori; essi dunque rappresentano solo una porzione della chiusura transitiva della matrice.

Al fine di determinare la matrice di adiacenza della chiusura transitiva del grafo, è sufficiente applicare l'operazione di somma logica sulle varie matrici, ottenendo:

$$C_T = \sum_{i=1}^{n} A^i = A + A^2 + A^3 + ... + A^n$$

Piccola nota: se vi sono n nodi, i cammini potranno essere al più lunghi $n - 1$ lati (o n se si ha un ciclo).

L'operazione di prodotto di matrici è estremamente pesante, computazionalmente parlando; è dunque richiesto, per ogni matrice, un costo computazionale pari a $\Theta(V^3)$; poichè è necessario calcolare ciò per n matrici, dove n potrebbe coincidere con V (numero di vertici, nel caso peggiore), il costo computazionale complessivo potrebbe ammontare a:

$$\Theta(n \cdot V^3) = \Theta(V^4)$$

1.3.2 Algoritmo di Warshall

Esiste una valida alternativa all'algoritmo ora introdotto, per determinare la chiusura transitiva di un grafo: l'algoritmo di Warshall. Anche esso viene definito partendo da una rappresentazione matriciale del grafo, tuttavia essa è fondamentalmente diversa da quella di adiacenza prima utilizzata.

Si definiscono le matrici P^k a partire dagli elementi $p_{i,j}^k$ come:

$$p_{i,j}^k \triangleq \begin{cases} 1(TRUE), (V_i; V_j) \in E \\ 0, (V_i; V_j) \not\in E \end{cases}$$

Questa definizione si può estendere al generico caso $(k+1)$-esimo, ottenendo una relazione in grado di fornire la seguente matrice:

$$p_{i,j}^{(k+1)} \triangleq \begin{cases} 1(TRUE), (V_i; V_j) \in E \\ 1, p_{i,k+1}^k = 1 AND p_{k+1,j}^k = 1 \\ 0 altrimenti \end{cases}$$

Queste matrici sono una sorta di estensione del concetto di matrice di adiacenza (considerate come al solito matrici booleane, ossia costituite da valori binari) per percorsi di lunghezza $k+1$ si può infatti dire ad esempio che $P^0 = A$, ossia la matrice P costruita con $k = 0$ coincida assolutamente con la matrice di adiacenza; man mano che k aumenta, si ottiene una matrice che tiene conto di percorsi sempre più lunghi; intuitivamente, quando la matrice terrà conto di tutti i percorsi, si avrà ottenuto il risultato desiderato, ossia la chiusura transitiva del grafo, $C_T(V'; E')$.

Poichè il numero massimo di volte in cui l'algoritmo è reiterabile è pari a $|V|$, ossia al numero dei vertici, e poichè al fine di realizzare ogni matrice di rango superiore servono $\Theta(|V|^2)$ operazioni, si può dire che l'algoritmo abbia costo computazionale pari a $\Theta(|V|^3)$. La costruzione degli elementi segue questo algoritmo di base:

$$p_{i,j}^{(k+1)} = (p_{i,j}^k) OR ((p_{i,k+1}^k) AND (p_{k+1,j}^k))$$

1.4 Visita di un grafo

Abbiamo detto che molti problemi teorici o pratici possono essere in qualche modo ricondotti ad un grafo; ciò che spesso può accadere è il fatto che, definito il grafo modellizzante il nostro problema, ci serva visitarlo tutto, ad esempio al fine di trovare un certo elemento, piuttosto che verificare la corretta modellizzazione del problema, piuttosto che verificare la connessione del grafo, o molte altre ragioni che ora non stiamo a dire. Per quanto riguarda

il problema della visita del grafo, esistono due soluzioni pratiche, che noi ora esporremo e cercheremo di comprendere.

1.4.1 Visita in profondità

Ogni nodo è collegato ad un certo numero di altri nodi del grafo; ciò che si può intuitivamente fare, è ripetere ricorsivamente la seguente operazione: entrare in un nodo a caso di quelli disponibili (solitamente, in un'implementazione effettuata mediante liste di adiacenza, si parte dalla testa della lista principale), visitare e marcare ogni sottonodo, e ripetere ricorsivamente questa operazione. Il ciclo è sostanzialmente senza fine, a meno che non si introduca una qualche condizione di controllo, quindi scegliamo di farlo sui nodi contrassegnati: una volta che non vi sono più nodi contrassegnati, e si è tornati al punto di partenza, ossia al nodo dal quale è incominciata la procedura, si ferma il processo di ricerca. Riassumendo le operazioni effettuate, dunque:

1. Si parte da un nodo a caso, lo visita e contrassegna;
2. Si va ad uno qualunque dei nodi ad esso adiacenti, lo si visita e contrassegna;
3. Si ripete il punto 2 fino al raggiungimento di una foglia; si effettua dunque il backtrack, e si riparte da un nodo non contrassegnato;
4. Una volta tornati al nodo di partenza, e una volta che tutti i nodi adiacenti sono contrassegnati, la visita in profondità può terminare.

1.4.2 Visita in ampiezza

Questo tipo di algoritmo sfrutta un'idea un po' diversa dalla precedente, che ora cercheremo di spiegare. Si usa una struttura a "coda" al fine di memorizzare temporaneamente i puntatori agli elementi del grafo che si intende memorizzare. Fatto ciò, essendo la struttura una FIFO (First In First Out), il primo elemento ad entrare sarà il primo ad essere rimosso dalla coda. Si introduce nella coda un nodo elemento (presumibilmente, la testa del grafo, anche se la scelta del primo elemento può essere del tutto aleatoria) e viene marcato; poichè è l'unico elemento presente nella coda, esso viene prelevato, e vengono introdotti nella coda i puntatori dei nodi adiacenti a quello appena rimosso. Si applica quindi ricorsivamente questa operazione, usando come condizione di termine del programma quella di "coda vuota". Riassumendo, l'algoritmo effettua le seguenti operazioni:

1. Introduzione di un nodo del grafo nella coda;

2. Prelievo del primo elemento della coda, quindi visita e sua marcatura;

3. Introduzione nella coda di tutti i nodi adiacenti a quello prelevato;

4. Ripetere dal punto 2, fino a quando la coda non è vuota.

1.5 Albero ricoprente minimo

La definizione di albero ricoprente minimo (o minimum spanning tree) ha senso a partire da un grafo non orientato pesato. Cos'è però questo MST ? Si tratta semplicemente dell'albero ricoprente i cui lati abbiano la somma minima dei pesi. Si noti che non è detto esista un unico minimum spanning tree: uno stesso grafo può averne diversi, poichè è assolutamente plausibile il fatto che esistano più percorsi con lo stesso peso complessivo.

Esiste un algoritmo in grado di determinare il minimum spanning tree, che verrà ora introdotto.

1.5.1 Algoritmo di Dijkstra/Prim

Questo algoritmo fa parte della classe degli algoritmi greedy: come la parola inglese può suggerire, è dunque un algoritmo "avido", dal momento che ricerca l'ottimo locale, ossia per ogni iterazione ricerca il minimo percorso effettuabile, sperando che questo sia il metodo ideale di ottenere anche l'ottimo globale. Si può dimostrare che, in questo caso, si ottenga sempre l'ottimo globale.

Descriviamo l'algoritmo: dato un vertice di partenza, esso viene inserito in una struttura ad albero (che sarà il nostro minimum spanning tree); per ogni iterazione si deve aggiungere all'albero un nuovo vertice, che viene scelto dal grafo con il seguente criterio: il vertice a distanza 1 dal nodo in cui si è posizionati, tale che abbia un collegamento con un nodo di costo minimo precedentemente scelto. Cerchiamo di rivedere questo algoritmo chiarendo alcuni aspetti importanti, che potrebbero essere sfuggiti: partendo dal nodo arbitrario, alla prima iterazione semplicemente bisognerà selezionare il nodo adiacente e più vicino a quello arbitrariamente scelto. A questo punto, si dispone di due nodi, dunque bisognerà scegliere il nodo adiacente ad uno di questi due tale per cui il peso sia minimo. Procedendo in questo modo, e scegliendo dunque ad ogni iterazione un nodo adiacente ad uno già introdotto nell'albero, tale per cui esso abbia il peso minimo tra tutti quelli possibili, si riesce ad ottenere il minimum spanning tree.

La complessità di questo algoritmo è dettata dal fatto che si può realizzare mediante due loop, uno interno all'altro, di lunghezza pari a $|V| - 1$; quindi,

si avrà in totale un algoritmo di complessità computazionale pari a $O(|V|^2)$. Esistono algoritmi più efficienti, utilizzanti strutture dati diverse (quali la coda prioritaria), ma verrà affrontato solo in seguito.

1.6 Albero dei cammini minimi

L'albero dei cammini minimi (shortest path tree), in qualche modo può ricordare il problema precedentemente affrontato come minimum spanning tree: dato un grafo pesato, eventualmente orientato, e dati due nodi di questo grafo, si vuole determinare il minimo percorso da effettuare al fine di collegarli, ossia il percorso tale per cui la somma dei pesi sia la minima.

Abbiamo detto che in qualche modo il problema può essere simile al precedente, ma non è del tutto vero in pratica: ciò che prima si intendeva fare era costruire l'albero di peso minimo, dunque si puntava a costruire l'albero la cui somma complessiva dei pesi era minima. Ora il nostro obiettivo è diverso: se prima si puntava in qualche modo all'ottimo globale, ora non siamo più interessati a ciò, poichè ci interessa esclusivamente lavorare "localmente", o meglio tra due nodi. Si noti che non è assolutamente detto che il percorso tra i due nodi stabilito con l'albero ricoprente minimo sia anche il cammino minimo tra due nodi: i due problemi per quanto possano sembrare simili hanno soluzioni diverse, e non sono per forza compatibili tra di loro.

1.6.1 Algoritmo di Dijkstra

Esiste un algoritmo in grado di determinare, a partire da un determinato vertice V_0, i cammini minimi verso tutti gli altri nodi, costruendo dunque un albero che abbia V_0 come radice e tutti gli altri nodi, collegati però solo con i lati che permettano il minimo cammino.

L'algoritmo funziona così: si inserisce come radice dell'albero il vertice di partenza, V_0. Si definisce dunque un concetto di distanza da V_0 agli altri nodi, nel seguente modo:

- Se il nodo i-esimo, V_i è adiacente a V_0, allora si introduce la distanza $d(V_i; V_o)$ come:

$$d(V_i; V_0) = W(V_0, V_i)$$

  Dove ovviamente $W(V_0; V_i)$ è il peso del lato compreso tra i due.

- Altrimenti, se il nodo i-esimo non è adiacente alla radice, si introduce peso $W(V_0; V_i) = \infty$ tra i due nodi (ossia si considerano tra di loro scollegati).

Effettuata questa prima fase, ad ogni passo x-esimo si aggiunge il nodo V_x per cui la distanza $d(V_0; V_x)$ sia minima, tra quelli a distanza unitaria dal punto di partenza V_0. A questo punto si ripete il ragionamento di prima, partendo da V_x: si crea un elenco di distanze tra V_x e tutti i nodi adiacenti, rendendo pari a ∞ tutte le altre; viene inoltre aggiornata la distanza complessiva percorsa a partire dal nodo V_0: per ciascun nodo i-esimo si aggiorna la lista delle distanze, considerando:

$$d(V_0; V_i) = min(d(V_0; V_i); d(V_0V_x) + W(V_xV_i))$$

L'algoritmo dunque viene ripetuto, fino a quando non si è trovato il percorso minimo tra ciascun nodo e quello di partenza, V_0.

La complessità di questo algoritmo è simile a quella dell'algoritmo di Dijkstra/Prim: $O(|V|^2)$. Anche in questo caso, mediante una coda prioritaria, è possibile migliorare le cose rendendo l'algoritmo più efficiente.

1.7 Definizioni fondamentali - parte seconda

Abbiamo precedentemente introdotto alcune definizioni legate ai grafi piuttosto basilari; per gli argomenti che abbiamo appena introdotto, è necessario introdurre alcune altre definizioni, a partire dalle quali potremo introdurre nuove definizioni di tipi di grafi, per poterne ricavare alcune proprietà importanti.

1.7.1 Grafi bipartiti

Un grafo bipartito è un grado i cui vertici possono essere suddivisi in due sottoinsiemi, tali per cui ogni vertice di un sottoinsieme sia adiacente a uno o più vertici appartenenti solo ed esclusivamente all'altro sottoinsieme.

La caratteristica fondamentale è dunque quella per cui elementi dello stesso sottoinsieme non siano incisi dallo stesso lato. Caso particolare dei grafi bipartiti sono i grafi bipartiti completi, ossia grafi suddivisibili in due sottoinsiemi con n e m nodi, tali per cui tra essi esistano tutti i collegamenti possibili. Essi vengono indicati mediante la notazione $K_{n,m}$.

1.7.2 Isomorfismo

Due grafi $G(V;E)$ e $G(V',E')$ sono isomorfi se esiste una corrispondenza biunivoca tra gli elementi di V e V', in modo che anche gli elementi di E e E' siano in corrispondenza biunivoca. Dire che due grafi sono isomorfi, dunque, è un modo fine per dire che essi sono completamente uguali tra di loro: se si riesce a stabilire questa corrispondenza biunivoca tra i due insiemi costituenti il grafo, si può dire che i due grafi siano isomorfi. Dati dunque due grafi, essi sono isomorfi se esiste un'applicazione biunivoca in grado di preservare tutti i vertici ed i collegamenti tra di essi.

Lo studio dell'isomorfismo tra due grafi non è assolutamente banale: non si può infatti per ora dire con certezza se la verifica dell'isomorfismo faccia parte dei problemi P o NP; unico caso particolare, studiabile a partire da una definizione non ancora introdotta, è quello di grafi planari isomorfi: in tal caso la determinazione dell'eventuale isomorfismo è un problema appartenente alla classe P.

1.7.3 Automorfismo

Per automorfismo si intende l'isomorfismo di un grafo con sè stesso. Si può dire che un automorfismo di un grafo sia una permutazione dei nodi, in grado di preservare però tutti gli archi ed i non-archi del grafo. Intuitivamente si può pensare che i grafi completi dispongano di tutti gli automorfismi possibili: permutando i vertici a propria scelta, si potrà sempre e comunque ritrovare un grafo isomorfo a quello di partenza, poichè in un grafo completo esistono collegamenti tra ogni vertice, quindi non si varierà nessun lato.

1.7.4 Omeomorfismo

Per omeomorfismo di due grafi si intende un "rilassamento" della condizione di isomorfismo, ossia una relazione meno pesante da ottenere tra due grafi. Nella fattispecie, due grafi si dicono omeomorfi se sono isomorfi a meno di nodi di grado 2.

In parole più semplici, per omeomorfismo si intende un isomorfismo tra due grafi, con in più la possibilità di introdurre, "tra due vertici" collegati da un singolo lato, uno o più nodi (a patto che essi siano di grado 2, ossia abbiano esclusivamente "un ingresso ed un'uscita").

1.8 Grafi planari

Come abbiamo già detto più e più volte, un grafo è un'entità astratta, una struttura realizzata al fine di modellizzare determinati problemi più o meno concreti. Talvolta di tutti i tipi di grafi esistenti può esserne considerato solo un particolare tipo: abbiamo finora parlato di grafi orientati, pesati, e vari altri tipi di strutture dotate di determinate proprietà più o meno utili al fine di modellizzare molti problemi.

Un tipo di grafo non ancora introdotto e assolutamente fondamentale per modellizzare un enorme numero di problemi di diversi tipi (strade, percorsi di diverso tipo, circuiti elettronici, idrici, e quant'altro), è il grafo planare.

Cosa significa dire che un grafo è "planare" ? La risposta a questa domanda è assolutamente non banale, e proveremo ad esprimerla pur anticipando il fatto che non sarà possibile comprenderla immediatamente appieno.

Un grafo si definisce planare se è possibile rappresentarlo su di un piano, o su di una varietà topologicamente equivalente ad una sfera, senza che due dei suoi lati si intersechino (si parla di varietà o più semplicemente superficie topologicamente equivalente ad una sfera riferendosi al fatto che la proiezione ottenuta mediante una luce posta su di una sfera dà luogo ad un piano (intuitivamente parlando)). Come già detto, un grafo si può disegnare su di un piano indicando con dei pallini i vertici, con segmenti i lati. Un grafo si dice planare se non è possibile disegnarlo su di questo piano senza intersezioni, ossia se non è possibile "vederlo" in due dimensioni senza incappare in intersezioni (in tre dimensioni sarebbe possibile poichè, introducendo la terza dimensione, sarebbe possibile evitare le intersezioni; esisteranno chiaramente dunque casi in cui non è possibile rappresentare in tre dimensioni, ed estendere ulteriormente sarebbe difficile in quanto la quarta dimensione non si può visualizzare direttamente).

1.8.1 Teorema di Kuratowski

Esistono alcuni teoremi e/o criteri in grado di determinare la planarità di un grafo. Un risultato fondamentale è quello di Kuratowski, che nel 1930 ha proposto il seguente teorema, del quale non verrà riportata la dimostrazione.

Il teorema di Kuratowski dice che un grafo è planare se e solo se non contiene alcun sottografo omeomorfo a K_5 o $K_{3,3}$. Per verificare la planarità di un grafo, quindi, è sufficiente studiare un grafo e cercare tra i suoi sottografi K_5 o $K_{3,3}$: se sono presenti, il grafo non è planare, se non sono presenti il grafo è planare. Effettuare verifiche del genere richiedono tuttavia algoritmi molto complessi da spiegare, per quanto essi siano efficienti.

Faccia di un grafo planare

Prima di introdurre uno dei risultati fondamentali della teoria dei grafi planari, introduciamo una definizione ulteriore: la faccia di un grafo planare. Per faccia di grafo planare si intende un insieme di punti connessi sul piano sul quale è disegnato il grafo planare. Detto in altre parole, una faccia è una regione limitata dai lati del grafo; viene considerata faccia anche quella "esterna" alle regioni limitate, ossia quella illimitata. Questa definizione verrà poi estesa in seguito.

1.8.2 Teorema di Eulero

Dato un grafo planare e connesso con $|V|$ nodi, $|E|$ lati e $|F|$ facce, si ha che:

$$|V| + |F| = |E| + 2$$

Questo teorema si può applicare per poliedri convessi o isomorfi a poliedri convessi: solo in questo caso essi sono proiettabili su di una superficie topologicamente equivalente ad una sfera, e quindi formare un grafo planare. Esistono tuttavia estensioni al teorema di Eulero, per esempio per grafi planari non connessi: si può infatti dire che, dato un grafo con k componenti non connesse (ossia k nodi isolati rispetto al resto del grafo), si ha:

$$|V| + |F| = |E| + k + 1$$

Prima di introdurre una generalizzazione definitiva del teorema di Eulero, vengono presentate due definizioni preliminari.

Grafo duale di un grafo planare

Si definisce, a partire da un grafo planare $G(V, E)$, il suo duale $G^*(E, V)$, come un grafo che ha:

- Per ogni faccia di G un vertice;
- Per ogni vertice di G una faccia;
- Per ogni lato di G un lato, che colleghi i due vertici corrispondenti alle due facce separate dal lato di G.

Cosa significa tutto ciò? Per ogni faccia si inserisce un nodo del grafo, e per ogni vertice si introduce una faccia. Realizzato ciò, si introducono i vertici tali da permettere questo tipo di realizzazione, ossia tutti i vertici in grado di collegare le facce ed i vertici ottenuti con il procedimento prima descritto.

Genus di un grafo

Se un grafo non planare non può essere disegnato su di una superficie topologicamente equivalente ad una sferica, possiamo immaginare intuitivamente che comunque esista almeno un'altra classe di superfici sulle quali si possa disegnare il grafo, senza dover utilizzare intersezioni.

Si può verificare visivamente facilmente che un grafo come K_5 o $K_{3,3}$ può essere rappresentato senza incroci su superfici di tipo toroidale, ossia superfici presentanti un "foro" centrale. Si dice, per questo motivo, che K_5 e $K_{3,3}$ abbiano "genus" pari a 1.

Cos'è dunque il "genus" di cui abbiamo parlato? Data una generica varietà topologica, si definisce il suo genus come il numero di "buchi" che comunque non la rendono disconnessa. Il genus di un grafo è equivalente al genus della minima varietà topologica necessaria al fine di rappresentare un grafo senza dover "incrociare" i lati. Possiamo dunque dire che un grafo planare, dal momento che è rappresentabile su di una varietà topologica equivalente ad una sfera, abbia genus 0.

Si può dunque estendere il teorema di Eulero per grafi "rappresentabili" su generiche varietà topologiche, dicendo che, dato il genus G della minima varietà sulla quale è possibile "disegnare" il grafo senza incroci, si ha:

$$|V| + |F| = |E| + (2 - 2G)$$

1.9 Grafi euleriani

Il matematico Eulero nella prima metà del 1700 propose un problema, a partire da una caratteristica della città di Koenigsberg, in Russia. La città in questione è costruita su di un fiume, e presenta diversi isolotti collegati al resto della città mediante un certo numero di ponti. Il problema di Eulero era molto semplice: è possibile determinare un percorso in grado di attraversare tutti i sette ponti colleganti le varie zone della città una sola volta, e ritrovarsi al punto di partenza?

Questo problema è risolubile, utilizzando la teoria dei grafi, teoria nata esattamente insieme a questo problema, grazie al matematico Eulero: modellizzando ciascun isolotto con un vertice, e ciascun ponte con un lato, si può semplicemente introdurre osservazioni sul grafo così costruito e cercare risoluzioni del sistema.

A partire da queste idee, si fornisce una definizione di ciclo euleriano: per ciclo euleriano si intende un cammino chiuso che contenga una sola volta

tutti i lati del grafo. A partire da qua, nasce la definizione di grafo euleriano, ossia grafo in cui sia contenuto almeno un ciclo euleriano.

Esiste un teorema, che permette di determinare il fatto che un grafo sia o meno euleriano.

Teorema: condizione necessaria e sufficiente affinchè un grafo connesso sia euleriano è il fatto che tutti i nodi siano di grado pari (ossia che il numero di lati incidenti sul nodo sia pari).

Si propone una dimostrazione del teorema in quanto essa può tornare utile per comprendere alcuni meccanismi sui grafi. Sostanzialmente, al fine di dimostrare il fatto che un grafo sia euleriano, si procede con due step:

1. Si decompone il grafo in una serie di cicli semplici, mediante un algoritmo apposito; questo verrà descritto in seguito.

2. Si crea il ciclo Euleriano percorrendo i cicli trovati nel punto precedente e utilizzando la seguente regola: ogni lato che viene percorso viene eliminato; ogni qual volta si incontri, giunti ad un nodo, un lato di un altro ciclo non ancora percorso, si inizia a percorrere questo; dualmente, se il lato è di un ciclo già percorso, si rimane sul ciclo corrente.

Per quanto riguarda la costruzione dei cicli semplici, si propongono due algoritmi da utilizzare uno alternativamente all'altro:

1. Un ciclo si può determinare partendo da un nodo arbitrario, ed effettuando una visita in profondità, considerando come condizione di fine visita il ritorno al nodo di partenza. Giunti al nodo di partenza dopo aver visitato un certo numero di nodi e passando per un certo numero di lati, si eliminano i lati visitati ed eventuali nodi isolati ottenuti nel grafo in seguito alla cancellazione dei nodi. Quelli che si otterranno saranno o sottografi semplici del grafo di partenza, o sottografi contenenti self-loop (che noi intendiamo eliminare).

2. Se vi sono ancora lati nel grafo in seguito all'applicazione dell'algoritmo 1, si vogliono determinare le componenti ancora connesse. Le operazioni svolte dall'algoritmo 1 infatti possono aver spezzato un singolo grafo in più sottografi, connessi. L'algoritmo 2 selezionerà un nodo di partenza e, mediante una visita (di ampiezza o profondità) determinerà l'ampiezza dei sottografi connessi. Si applicherà dunque di nuovo l'algoritmo 1 ai componenti connessi del grafo rilevati in questo modo. Se per caso ci fossero invece nodi con cappio (self-loop), essi prima o poi verranno processati dall'algoritmo 1, che entrerà nel nodo, mediante il cappio tornerà su sè stesso, annullerà il lato ed il nodo (ormai rimasto isolato, dopo l'eliminazione del suo unico lato, ossia il cappio stesso).

A seconda delle decisioni arbitrarie prese nei due passi, si potran ottenere, al termine dei due step, diversi cammini euleriani (se ne esistono).

1.9.1 Algoritmo di Fleury

Esiste un altro algoritmo per la determinazione di un ciclo euleriano in un grafo: l'algoritmo di Fleury. Esso si basa sulla definizione di "istmo", ossia di lato che, se eliminato, rende sconnesso il grafo.

L'algoritmo di Fleury funziona nel seguente modo: partendo da un nodo arbitrario, si visita il grafo in profondità scegliendo ogni volta un lato non percorso e cancellandolo; se possibile (a seconda dei lati già percorsi e quindi cancellati) si dovrebbe preferire di non passare su di un istmo. Ad un certo punto passare per l'istmo sarà obbligatorio, e quindi si dovrà cancellare anche esso, rendere il grafo sconnesso, e quindi terminare l'algoritmo. A questo punto, se il grafo non aveva vertici di grado dispari, quello ottenuto è stato un ciclo euleriano, altrimenti un percorso euleriano.

Per percorso (o cammino) euleriano si intende un cammino (ossia dove vertici iniziale e finale sono diversi), che contiene una sola volta tutti i lati del grafo. Per avere un grafo "semi-euleriano" si intende una condizione più "debole": si richiede che, esista un cammino (non obbligatoriamente chiuso, e quindi non obbligatoriamente un ciclo) in grado di toccare una ed una sola volta ciascuno dei lati del grafo. Poichè quella di "semi-eulerianità" è una condizione più debole rispetto all"'eulerianità", si può immaginare che le richieste siano più deboli (come già detto), e così effettivamente è: perchè un grafo sia semi-euleriano, è sufficiente che tutti i nodi siano di grado pari, eccetto due. Se "euleriano" è dunque un percorso o ciclo che tocchi tutti i lati almeno una volta, "semi-euleriano" è un ciclo o percorso che tocca tutti i lati almeno una volta. Ovviamente, eulerianità implica semi-eulerianità, ma l'implicazione inversa non è garantita.

1.10 Grafi hamiltoniani

Se il matematico Eulero propose come teorema lo studio di cicli o percorsi in grado di toccare tutti i lati, il fisico-matematico Hamilton propose un problema sostanzialmente duale: lo studio di grafi in cui sia possibile visitare tutti i vertici una sola volta.

Si definisce per l'appunto un cammino hamiltoniano un generico cammino che permetta (con $|V| - 1$ lati) di visitare una sola volta tutti i vertici del grafo. Ciclo hamiltoniano è l'equivalente appena pronunciato, però con la condizione aggiuntiva che il percorso inizi e termini sullo stesso nodo.

Un grafo hamiltoniano è un grafo che contenga almeno un ciclo hamiltoniano. Esistono anche per quanto riguarda l'hamiltonianità condizioni meno stringenti: chiedendo che nel grafo sia presente almeno un cammino (non obbligatoriamente chiuso) in grado di passare per ciascuno dei nodi, si ottiene la condizione di semi-hamiltonianità.

Per quanto, dalla presentazione, il problema della determinazione dell'hamiltonianità di un grafo potrebbe sembrare risolubile in maniera relativamente semplice, esattamente come la determinazione di cammini o cicli euleriani, non sono ancora state trovate condizioni necessarie e sufficienti, ossia condizioni minimali per la determinazione dei cicli hamiltoniani. Quelli che si hanno sono risultati in grado di fornire quantomeno la possibilità di capire se un grafo sia hamiltoniano, ma non in grado di fornire cicli hamiltoniani nel grafo.

1.11 Clique di un grafo

La clique (o "cricca") di un grafo indica il suo sottografo massimo completo, ossia il sottografo completo di ordine più elevato del grafo di partenza. La massima cardinalità della clique è detta "clique number".

1.12 Colorabiità di un grafo

Ci si pone il problema seguente: dato un grafo non orientato, qual è il numero minimo di colori utilizzabile per marcare ciascun nodo, a patto che due nodi tra di loro adiacenti non siano colorati dello stesso colore?

Questo problema è molto particolare e molto studiato, poichè di difficile soluzione. Si definisce il "numero cromatico" di un grafo, $\chi(G)$ il minimo numero di colori richiesti per la colorazione di un grafo. Si tratta di problemi NP-completi, e dunque estremamente complicati da studiare.

L'unico risultato interessante ottenuto sulla colorabilità dei grafi è il seguente: esistono diversi algoritmi approssimati di complessità polinomiale per la colorazione di un grafo, ma nessuno è valido in casi generali, e non si sa nemmeno se esistano; si è dimostrato tuttavia che, se esistesse un algoritmo di colorazione approssimato che utilizzi al massimo il numero di colori doppio del numero cromatico, si potrebbe ottenere un algoritmo in grado di ricavare il numero cromatico con complessità polinomiale, e dunque si potrebbe dimostrare il fatto che i problemi P hanno la stessa complessità dei problemi NP (P = NP).

Solo recentemente sono stati ottenuti risultati concreti in merito a questi problemi, ma con dimostrazioni estremamente complesse, e dunque poco affidabili (poichè difficili da verificare); si attende di scoprire risultati migliori, più semplici, al fine di risolvere definitivamente il problema della colorabiità del grafo (e magari anche altri problemi ad essa correlati, come quello appena citato).

1.13 Dominanza di un insieme di nodi

Per "insieme dominante" di nodi di un grafo (non orientato) si intende un certo insieme di nodi tali per cui ogni altro nodo è adiacente ad un nodo appartenente all'insieme. Estensione di questo concetto è quello di "insieme dominante minimale" (o non riducibile), ossia un insieme tale per cui:

- Ogni vertice non appartenente all'insieme è adiacente ad un vertice appartenente all'insieme dominante;
- Nessun sottoinsieme dell'insieme dominante è un insieme dominante

Per insieme dominante "minimo", infine si intende l'insieme dominante di cardinalità minima presente in un grafo (si definisce come $\alpha(G)$.

Il motivo per cui si definisce la "dominanza" di un insieme di nodi è il fatto che molti problemi di diverso tipo sono riconducibili alla determinazione dell'insieme dominante di un grafo. Un esempio potrebbe riguardare le telecomunicazioni: data una rete, schematizzata con un grafo, gli insiemi dominanti sono i punti ideali per il posizionamento delle stazioni di trasmissione.

Per determinare un insieme dominante, vi sono alcune osservazioni che è importante tener presente: un insieme dominante deve contenere, per ogni nodo, o il nodo stesso o uno dei nodi adiacenti ad esso. Si può "schematizzare" questa idea, in un algoritmo di questo genere:

1. Si costruisce un'espressione logica in stile "somma di prodotti", dove la somma di diverse variabili (ciascuna identificante un nodo) rappresenta l'adiacenza tra essi; il prodotto logico "unisce" tutte le adiacenze, costituendo a tutti gli effetti il grafo;
2. Costruita l'espressione logica del grafo, la si minimizza con le tecniche note (mappa di Karnough piuttosto che altre); ciascuna delle espressioni minimali ricavate rappresenta un insieme dominante minimale.

Dato n grafo senza nodi isolati, dato un insieme D dominante minimo di termini, anche il complementare $\bar{D}$ è un insieme dominante minimo; da ciò, si possono dedurre le seguenti espressioni:

- Ogni grafo senza nodi isolati presenta almeno due insiemi dominanti;
- Un insieme dominante minimo contiene $\lfloor \frac{|V|}{2} \rfloor$ nodi.

1.14 Indipendenza di un insieme di nodi

Il concetto di indipendenza di un insieme di nodi è il duale di quello di dominanza: un insieme di vertici di un grafo è un insieme indipendente se non comprende vertici adiacenti. Se almeno due nodi dell'insieme sono adiacenti tra loro, allora l'insieme non è indipendente. Si definisce, dualmente a prima, l'insieme indipendente massimale come il massimo insieme indipendente realizzabile, ossia di cardinalità massima rispetto al grafo sul quale si ricerca: aggiungendo ad esso un vertice qualunque, esso diventa obbligatoriamente indipendente.

Capitolo 2

Algoritmi di ricerca

2.1 Alberi binari di ricerca

Gli alberi binari di ricerca (BST: Binary Search Tree) rappresentano, assieme alle hash table, una delle strutture dati più diffuse e utilizzate per effettuare ricerche. Si tratta di alberi binari, e dunque di un sottoinsieme dei grafi in cui non esistono cicli, e ogni nodo ha al più un padre e due figli, in cui ad ogni nodo corrisponde un elemento dell'insieme di dati che si intende trattare. Ciascun nodo, nella fattispecie, contiene le seguenti informazioni:

- Chiave, ossia elemento utilizzato per la ricerca nel grafi;
- Dati associati, ossia le informazioni utili contenute in ciascun elemento;
- Puntatori al figlio sinistro (LEFT), al figlio destro (RIGHT), eventualmente al genitore (PARENT).

Ogni figlio è un altro elemento che potrebbe essere ricercato dagli algoritmi sulla struttura che studieremo. Poichè l'albero binario è una struttura dati già ordinata al momento della sua definizione, al momento dell'inserimento nella struttura, è necessario utilizzare una convenzione che sia comune a più utenti possibili. Si sceglie, nella fattispecie, che le chiavi di ogni figlio sinistro (LEFT) siano minori della chiave del nodo stesso; dualmente dunque le chiavi di ogni figlio destro (RIGHT) saranno maggiori della chiave del nodo in questione.

Per come è stato definito, un bst ha alcune proprietà fondamentali:

- Effettuando una visita di tipo IN-ORDER, ossia visitando prima il figlio sinistro, poi il padre, poi il figlio destro (ricorsivamente), si ottengono tutte le chiavi in ordine crescente; la cosa è abbastanza intuitiva, poichè

ciascun padre, per come è stato precedentemente definito il grafo, è sicuramente maggiore del figlio sinistro e sicuramente minore del figlio destro.

- Tutte le operazioni effettuabili in un BST sono $O(p)$, dove p è la profondità del grafo; se però l'albero è "bilanciato", ossia se tutte le foglie si trovano allo stesso livello, p (o $p-1$, a seconda di come si definisce la profondità del grafo, se considerando la radice livello 1 o 0), le operazioni sono $O(\log_2(n))$; tutto ciò, supponendo che tutte le chiavi siano distinte.

Descriviamo dunque le principali operazioni effettuabili su di un albero binario di ricerca, spiegando come si possono implementare.

- Ricerca (search): si dispone di una certa chiave k da ricercare (che potrebbe essere un numero, una stringa, o qualcos'altro), e si vuole ricercare nell'albero l'elemento associato a questa chiave. Supponendo che nell'albero sia presente un elemento associato a questa chiave, si effettuano i seguenti passi;

  1. Si confronta la chiave k con quella della radice, k_r; se coincidono, la ricerca è terminata; se $k < k_r$, si applica recursivamente la procedura al sottoalbero di sinistra, poichè esso contiene tutti gli elementi minori della radice; dualmente, se $k > k_r$, il procedimento va ripetuto al sottoalbero destro;
  2. La procedura termina se si trova k, o se il sottoalbero che si intende analizzare è vuoto.

  Al massimo, sono possibili $p+1$ confronti.

- Inserimento (insert): si applica il procedimento di ricerca, leggermente modificato: se prima ci interessava il cercare un elemento, ora ci interessa la ricerca di una posizione vuota, ossia di un sottografo vuoto, tale per cui si possa inserire un elemento rispettando le caratteristiche da noi prima definite sul binary search tree; quando si determina dunque un sottografo vuoto tale per cui la posizione dettata dalla chiave da inserire k, si effettua l'inserimento del nuovo nodo come figlio sinistro o destro (a seconda della topologia dell'albero); il massimo numero di confronti effettuabili coincide con quello della ricerca, e quindi è pari a $p+1$.

- Eliminazione (delete): prima di tutto, mediante la ricerca si trova l'elemento da cancellare dall'albero; a questo punto, si deve studiare una

serie di casistiche, e si dovrà procedere in modo diverso a seconda di quella presente:

- Se il nodo da cancellare non ha figli, basta cancellarlo, modificando il puntatore nel nodo padre;
- Se il nodo ha un solo figlio, è sufficiente modificare il puntatore del padre del nodo da cancellare in modo da farlo puntare verso suo figlio, facendo quindi di fatto "risalire" il figlio di una posizione;
- Se il nodo da cancellare ha due figli, il caso si complica leggermente: bisogna sostanzialmente capire quale dei due figli è idoneo ad essere "alzato", come nel caso precedente. Nella fattispecie, non esiste un solo modo di procedere, poichè vi sono due possibilità (entrambe valide): o si sostituisce il nodo da cancellare con il "nodo più piccolo del sottoalbero destro", o si sostituisce con il "nodo più grande del sottoalbero sinistro", dove "piccolo" e "grande" si riferiscono ovviamente alle chiavi, introdotte ordinatamente. La cancellazione di un elemento è $O(p)$.

Come abbiamo accennato precedentemente, una condizione ottimale per l'albero è il suo bilanciamento, poichè le funzioni studiate hanno una complessità computazionale pari a $O(\log_2(n))$. Le funzioni studiate, se applicate, possono però "sbilanciare" un albero (inserzione e cancellazione non hanno, per come le abbiamo definite, condizioni in grado di preservare il bilanciamento dell'albero); esistono tuttavia alcune considerazioni, ed alcune tecniche (che non tratteremo), in grado di mantenere bilanciati gli alberi.

2.2 Hash table

Quando si ha un numero di elementi da gestire, n, molto elevato, le tabelle di hash sono i mezzi più potenti di cui si possa disporre. Di fatto, una tabella di hash permette di effettuare una ricerca in un tempo medio circa costante.

Il concetto alla base delle hash table è estremamente diverso da quello usato nelle tecniche di ricerca finora studiate: fino ad ora, infatti, in qualche modo la ricerca avveniva mediante l'uso di confronti tra chiavi di vario genere. In altre parole, si disponeva di una chiave di ricerca, ed essa veniva continuamente confrontata con le chiavi utilizzate in strutture dati più o meno ottimizzate (dai vettori alle liste ai BST), fino a ricercare la condizione di "matching" tra le due chiavi (ossia fino a trovare la chiave uguale a quella da ricercare). Ora la chiave non viene più utilizzata per una ricerca mediante

confronti, ma viene "codificata" in qualche modo (mediante funzioni particolari) in modo da ottenere, un uscita dalla codifica, l'indice di un vettore di strutture, contenente i dati che si intende ricercare. La definizione più basilare che si potrebbe dunque fornire per una hash table è "una struttura dati in grado di associare una chiave di ricerca ad un valore numerico" (l'indice del vettore in questione, utilizzato per "ospitare" la base dati).

Finora abbiamo esclusivamente parlato dei pregi di questo tipo di strutture, ma è evidente che, se non vengono usate esclusivamente esse in qualsiasi tipo di ricerca ed ordinamento, devono obbligatoriamente avere problemi di qualche genere. Così effettivamente è: come abbiamo detto, alla base delle hash table vi sono funzioni particolari, dette "hash function": esse sono le funzioni che "traducono" una chiave di qualche tipo (numerica, alfabetica, alfanumerica...) in un valore numerico, rappresentante l'indice di un vettore di strutture. Ogni hash function può generare un solo valore, ma non è detto che ad un valore corrisponda un solo elemento, una sola chiave di ricerca: le hash function non sono funzioni biiettive (biunivoche): ad ogni chiave potrebbero essere associati più elementi. Ciascun caso in cui si generi lo stesso indirizzo a partire da due chiavi diverse, viene definito "collisione"; non esistono funzioni di hash in grado di non generare collisioni, e se esistessero avrebbero problemi sotto altri punti di vista: più si cerca di ridurre le collisioni, più la codifica utilizzata dalla funzione di hash è complicata, e rischia di coinvolgere spazio in memoria per la struttura dati. All'enorme velocità che queste strutture dati garantiscono per le funzioni di inserimento e ricerca, dunque, si contrappongono le grosse quantità di memoria necessarie per l'uso, e soprattutto le collisioni tra chiavi diverse.

Poichè la creazione di funzioni di hash efficienti è estremamente complicata e lo scopo di questa trattazione è un'introduzione alle strutture dati, si decide di discutere alcuni metodi di trattamento degli eventi di collisione, supponendo di avere funzioni di hash che ci costringano a rinunciare a priori all'univocità degli indici ricavati a partire dalle chiavi. Data una hash table con m indici, ossia con m posizioni (in altre parole, un vettore di strutture con m elementi) e k è una chiave di ricerca, esistono sostanzialmente due tecniche per il trattamento delle collisioni, che ora cercheremo di illustrare.

2.2.1 Metodo 1: concatenazione

La concatenazione (chaining) è una tecnica di risoluzione delle collisioni basato sull'uso di strutture di tipo dinamico, nella fattispecie sull'uso di liste.

Si dispone di un vettore di strutture, e supponiamo che nella struttura ci sia un puntatore per una lista di elementi dello stesso tipo di struttura; questa lista serve, come suggerisce il titolo, a "concatenare" eventuali collissioni

tra elementi la cui chiave ha prodotto lo stesso indice del vettore. Date n collisioni avvenute nel processo di hashing, si può ovviamente supporre che ogni lista sia mediamente lunga $\frac{n}{m}$ (date n collisioni, le mediamo su m elementi della hash table, data distribuzione uniforme di probabilità di collisione al variare di n e m). Si definisce, per questo tipo di tecnica, un coefficiente di riempimento α, definibile come:

$$\alpha = \frac{n}{m}$$

Nel caso *alpha* risultasse essere troppo grande, potrebbe esser necessario ristrutturare la tabella, in modo da cercar di ridurre le collisioni: dato questo coefficiente, le operazioni di ricerca, inserimento e cancellamento sono $O(\alpha)$; quindi mantenerlo basso significa rendere prossima a costante la ricerca di un elemento anche in caso di collisioni abbastanza frequenti (cosa che comunque non dovrebbe capitare, con una hash function progettata ad hoc).

2.2.2 Metodo 2: indirizzamento aperto

L'indirizzamento aperto (open addressing), è una tecnica basata sul fatto che ogni elemento del data base va inserito nella tabella di hash, a prescindere dal fatto che esso collida o meno con un altro elemento; perchè ciò dunque sia verificato, il numero di collisioni n deve sicuramente essere minore del numero di indici m (altrimenti il vettore sarebbe troppo piccolo per permettere di introdurre così tanti valori: sarebbero più i valori tra di loro collidenti che quelli introducibili, per questioni di spazio, nel vettore sul quale si basa la hash table); altro modo di esprimere questa condizione, è dire che $\alpha < 1$ (ricavabile banalmente dalla definizione).

Abbiamo detto che tutti gli elementi devono essere la tabella, ma come si gestiscono dunque le collisioni? Esistono diversi tipi di metodi, detti "scansioni": si tratta di tecniche in grado di assegnare ad un elemento collidente con un altro un elemento della tabella differente da quello che gli spetterebbe (poichè già occupato da un altro elemento). Presentiamo ora un certo numero di scansioni, in grado di risolvere o quantomeno gestire il problema delle collisioni.

Scansione lineare

La scansione lineare (linear probing) sostanzialmente si basa sull'esaminare l' "elemento successivo" della tabella rispetto a quello dove si desidererebbe introdurre il collidente. Vediamo come funziona questa tecnica, presentando la formula caratteristica in grado di realizzarla con un formalismo matematico:

$$h_r(k_i) = [h(k_i) + q \cdot r]$$

La funzione di hash $h(k)$ è in grado di restituire, a partire da una chiave k, un valore numerico associato ad un certo indice della hash table; poichè si verifica una collisione, è necessario posizionare in un'altra posizione questo elemento: si sceglie di fare ciò con elementi distanti r (dopo parleremo di q, per ora consideriamo $q = 1$) dall'elemento principale, dove r è il più piccolo numero naturale ($r \in \mathbb{N}$) che, sommato all'indice della tabella ricavato dalla funzione $h(k_i)$, indichi uno spazio vuoto. r dunque è un numero naturale che può valere 1, 2, 3... . q è una variabile aggiuntiva, che permette di "distanziare" in modo costante dal punto di collisione la posizione in cui sarà inserito l'elemento collidente. Questa tecnica è abbastanza efficiente, per tabelle non molto piene: se una tabella è riempita per circa $\frac{2}{3}$ delle sue dimensioni totali, mediamente si avrà un numero di tentativi molto piccolo (circa pari a 2); il caso pessimo sarebbe lineare ($O(n)$), ma disponendo di una buona funzione di hash, è praticamente impossibile anche solo avvicinarsi ad esso.

Un fenomeno non molto piacevole che spesso si verifica utilizzando una scansione di tipo lineare è l' "agglomerazione primaria": a forza di inserire elementi tra di loro adiacenti, si forma una sequenza di posizioni (contigue) piene; in questo modo, nel caso un elemento introdotto nella hash table dovesse essere inserito in un blocco che "gli spetta", ma già occupato da un elemento lì inviato per risolvere una precedente collisione, si verificherebbe un'ulteriore collisione. Un'idea che potrebbe venire è quella di porre $q \neq 1$: in questo modo cambierebbe la struttura della tabella, ma non la sostanza, poichè si formerebbero comunque agglomerati primari, solo non contigui. Un rimedio a questo tipo di problema, sarebbe utilizzare, al posto di q, un $q(k)$, ossia un distanziamento variabile a seconda delle caratteristiche della chiave. In questo modo, scegliendo una funzione idonea per il distanziamento, si potrebbe quantomeno attenuare gli effetti di questo problema, ma non eliminare del tutto di sicuro.

Scansione quadratica

L'idea alla base della scansione quadratica può essere formalizzata nella seguente espressione matematica:

$$h_r(k_i) = \left[h(k_i) + r^2\right] \%m$$

Cosa si fa dunque? Si considera r come sempre appartenente all'insieme dei naturali, ma invece che considerare r adiacente o comunque distanziabile

in modo lineare, lo si considera "quadraticamente": anzichè usare l'indice successivo (o comunque proporzionale al successivo per fattore q) si considera una variazione quadratica degli spazi utilizzabili, come quindi r^2. Tutto ciò viene normalizzato, per le posizioni disponibili per la tabella (m), mediante l'operazione % (resto, "mod").

Questo tipo di scansione è ulteriormente migliorabile, inserendo una costante di peso variabile con k_i, $q(k_i)$, e quindi rendendo la formula di questa forma:

$$h_r(k_i) = \left[h(k_i) + q(k_i) \cdot r^2\right] \%m$$

Note conclusive

Per quanto riguarda le hash table, vi è una "grande verità": per tabelle con occupazioni inferiori al 60% o 70%, il fatto che si usi una scansione piuttosto che un'altra non cambia molto. Il caso peggiore, comunque, è quello della scansione lineare (che però è anche la più semplice da gestire). Le tecniche finora descritte sono molto utili, per operazioni di ricerca (search) o di inserimento (insert); la cancellazione andrebbe gestita in modo diverso, ma ciò di solito non si effettua, per basi dati che sfruttano come idea la hash table. Una soluzione utilizzabile è quella di gestire meglio la struttura dati, introducendo un campo aggiuntivo, in grado di eliminare disambiguità: in fase di ricerca, un elemento libero potrebbe creare disambiguità con la chiave ricavabile mediante le scansioni finora ideate, e ciò peggiorerebbe notevolmente le prestazioni della struttura dati. Se nella struttura si inizialissasse un campo in grado di determinare se il blocco associato ad un certo indice sia "libero", "occupato", o "cancellato", permetterebbe di eliminare le disambiguità: in fase di ricerca, "cancellato = occupato", in fase di inserzione "cancellato = libero": la ricerca non studierebbe così un blocco libero e così eviterebbe problemi di diverso genere.