UNIVERSITA' DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Elaborato per l'esame di
"Fondamenti di Informatica II"
Prof. G.Soda

MPHF

di Menchetti Sauro

A.A. 1996/97

Funzioni hash perfette e minimali per grandi basi di dati

Introduzione

Presente in tutti i campi dell'informatica è il bisogno di accedere agli oggetti tramite il valore di una chiave, quindi la promessa di un accesso istantaneo fornita dalle funzioni hash è particolarmente entusiasmante. Purtroppo la maggior parte dei metodi hash spreca una certa quantità di spazio e di tempo: lo scopo di quest'articolo è proprio quello di migliorare le correnti tecniche di hashing, creando una funzione hash perfetta e minimale (MPHF) che eviti questi problemi. Il tutto fa riferimento a collezioni statiche di dati che in questi ultimi anni sono diventate sempre più comuni grazie all'enorme diffusione che hanno avuto i CD-ROM.

Partiamo con una collezione di oggetti ognuno dei quali ha associata un'unica chiave k, scelta da U, un universo finito di chiavi. La cardinalità di U è N. Mentre alcuni ricercatori assumono U come insieme degli interi $U = \{1...N\}$, noi adottiamo una meno restrittiva e più realistica assunzione, e scegliamo U come l'insieme delle stringhe di caratteri aventi una lunghezza massima finita. Il reale insieme di chiavi usato in un particolare database in un determinato momento è $S \subseteq U$, dove la cardinalità di S è molto minore di quella di U. Sia n la cardinalità di S. Tutti i record sono accessibili attraverso una tabella hash T avente $m \geq n$ locazioni; l'utilizzazione dello spazio in T è misurata dal *fattore di caricamento* $\alpha = n/m$. In tutti i casi è desiderabile non avere in T locazioni non utilizzate ed essere capaci di trovare velocemente l'esatta locazione di una data chiave k.

Il problema della ricerca è quello di individuare il record corrispondente ad una determinata chiave k o di verificare che tale record non esiste. Questo può essere fatto applicando una funzione hash alla chiave k ed esaminando la locazione in T con indirizzo $h(k)$. Se la funzione h è una funzione iniettiva su S, allora h è detta *funzione hash perfetta* (PHF), in quanto non c'è bisogno di spendere tempo per risolvere le collisioni. Inoltre se la tabella hash T è piena, cioè il fattore di caricamento $\alpha = 1$, la funzione h è detta *fun-*

zione hash minimale. Una funzione hash perfetta e minimale è ottima in quanto risolve i problemi di spazio e di tempo sprecato.

Caratteristiche di una MPHF

Ci concentreremo sulle funzioni hash perfette e minali (MPHF) invece che sulle tecniche di hashing riguardanti le collezioni dinamiche di dati. Il nostro obiettivo è di riuscire a trovare una MPHF che mappi ogni chiave k, scelta da un insieme di chiavi S di cardinalità n, in una locazione distinta della tabella hash T, contenente n slots; vogliamo cioè, associare ad ogni chiave un intero che rappresenta il suo indirizzo nella tabella hash.

Prova dell'esistenza

Possiamo chiederci se, dato un insieme di chiavi, esista sempre una MPHF che mappi ogni chiave in un intero. E' stato dimostrato che, dato un qualunque insieme di chiavi S, esiste sempre una funzione hash perfetta e minimale: il problema è come trovarla impiegando una ridotta quantità di spazio e di tempo. E' stato inoltre dimostrato che le PHF sono rare nell'insieme di tutte le funzioni: per mappare 31 parole in 41 indirizzi solo una funzione su 10 milioni è perfetta. Dobbiamo quindi cercare tali funzioni e definirle in una quantità di spazio e di tempo accettabili.

Spazio per immagazzinare una MPHF

Il limite inferiore per rappresentare un'arbitraria MPHF è di ≈ 1.4427 bits per chiave. E' più tipico specificare la taglia di una MPHF in termini di parole o words, ognuna di $\log_2 n$ bits. Così il limite inferiore è di $\Omega(n/\log_2 n)$ words per chiave mentre quello superiore è di $O(n)$ words per chiave. L'algoritmo che verrà presentato realizza una MPHF in meno di $O(n)$ words per chiave.

Classi di funzioni

Ci sono diverse strategie per trovare funzioni hash perfette. La più semplice è quella di selezionare una classe di funzioni che probabilmente includa un certo numero di funzioni hash perfette e di cercare tra di esse una MPHF assegnando valori diversi ai parametri che caratterizzano la classe. Esaminando due algoritmi su quest'argomento, possiamo trarne due importanti concetti: l'uso di tabelle di valori come parametri e l'uso di

tre passi, mapping, ordering e searching step in cui suddividere l'algoritmo di ricerca. Il **mapping** step trasforma le chiavi in un insieme di triple distinte, l'**ordering** step ordina le chiavi in una serie di livelli e il **searching** step assegna i valori hash alle chiavi un livello alla volta. Se il searching step trova un livello di chiavi che non è capace di accomodare, torna indietro ai livelli precedenti assegnando nuovi valori hash alle chiavi.

Concetti chiave dell'algoritmo

Ci sono tre fondamentali idee su cui si basa l'algoritmo:

- la casualità deve essere sfruttata ogni volta che è possibile; la probabilità di insuccesso nel caso medio viene così ridotta al minimo;
- osservando la distribuzione dei gradi dei vertici del *grafo delle dipendenze*, possiamo costruire l'ordering step in modo più efficiente;
- l'assegnazione dei valori hash alle chiavi può essere paragonata al tentativo di adattare in un disco già parzialmente pieno un certo insieme di dati, detto campione, dove è importante inserire i campioni più grandi quando il disco è solo parzialmente pieno.

Il Mapping step

Il primo passo dell'algoritmo consiste nel costruire tre tabelle di numeri casuali (*table* $_0$ *, table* $_1$ *, table* $_2$ *)*, ognuna delle quali contiene un numero casuale per ogni possibile carattere in ogni posizione *i* della chiave. Vengono definite tre funzioni ausiliarie nell'universo di chiavi *U*:

$$h_0 : U \rightarrow \{0,..., m - 1\}$$

$$h_1 : U \rightarrow \{0,..., r - 1\}$$

$$h_2 : U \rightarrow \{r,..., 2r - 1\}$$

nel seguente modo:

$$h_0(k) = \left(\sum_{i=1}^{y} table_{0i}(k_i) \right) \bmod n$$

$$h_1(k) = \left(\sum_{i=1}^{y} table_{1i}(k_i) \right) \bmod r$$

$$h_2(k) = \left(\sum_{i=1}^{y} table_{2i}(k_i) \right) \bmod r + r$$

dove la chiave è una stringa di caratteri $k = k_1 k_2 ... k_y$ ed r è un parametro (tipicamente $\leq n/2$) che determina quanto spazio occupa la MPHF (cioè $|h| = 2r$). Queste tre funzioni ausiliarie comprimono ogni chiave k in una tripla di interi $(h_0(k), h_1(k), h_2(k))$; è essenziale che queste n triple siano distinte, perché la MPHF finale deve essere in grado di distinguere una chiave dall'altra. La probabilità che n triple scelte a caso tra t triple siano distinte, è data da

$$p(n, t) \approx 1 - \frac{(\log_2 n)^2}{2c^2 n}$$

per un tipico valore di $r = cn/\log_2 n$, dove c è una costante; tale probabilità va rapidamente a 1 al crescere di n. La classe di funzioni cercata è

$$h(k) = (h_0(k) + g(h_1(k)) + g(h_2(k))) \bmod n$$

dove g è la funzione che viene assegnata durante il searching step.

I valori delle funzioni h_0, h_1, h_2 sono usati per costruire un grafo, detto *grafo delle dipendenze*. Metà dei vertici del grafo corrispondono ai valori di h_1 e sono etichettati 0,..., r - 1, l'altra metà corrispondono ai valori di h_2 e sono etichettati r,..., $2r$ - 1. C'è una linea nel grafo delle dipendenze per ogni chiave: ad ogni chiave k corrisponde una linea tra i vertici etichettati $h_1(k)$ e $h_2(k)$. Inoltre se il valore $h_0(k)$ è associato alla linea che unisce due vertici, tutte le informazioni necessarie all'algoritmo sono contenute nel grafo delle dipendenze. Se le triple generate non sono distinte, si procede alla costruzione di tre nuove tabelle di numeri casuali e questo fin quando le triple generate non sono distinte. Il grafo delle dipendenze è immagazzinato in due strutture dati, una per i vertici ed una per le chiavi, entrambe implementate in due array di record. Il tempo impiegato dal mapping step è $O(n)$.

L'Ordering step

L'ordering step esplora il grafo delle dipendenze in modo tale da dividere l'insieme delle chiavi in una sequenza di livelli. Si può dimostrare che in un grafo delle dipendenze costruito a caso, la maggior parte dei vertici ha basso grado (il grado di un vertice è il numero di chiavi che incidono in quel vertice). Questo fatto è sfruttato per ordinare le chiavi in sottolivelli. In realtà non sono le chiavi ad essere ordinate, bensì i vertici stessi ed in particolare sono ordinati i vertici di grado non nullo. Si parte da un vertice di grado massimo v_1 : ad ogni iterazione dell'ordering step, il vertice v_i è selezionato tra i vertici non selezionati adiacenti ad almeno uno dei vertici già ordinati v_1 ,..., v_{i-1}; tra questi, v_i è scelto di grado massimo. Dai vertici ordinati, è facile risalire ai sottolivelli in cui sono suddivise le chiavi. Se i vertici ordinati sono v_1 ,..., v_t , allora il livello di chiavi $K(v_i)$ corrispondente al vertice v_i , $1 \le i \le t$, è l'insieme delle chiavi incidenti in v_i e nei vertici precedenti nell'ordine, cioè

$$K(v_i) = \{ k_j \mid h_x(k_j) = v, h_y(k_j) = v, s < i \}$$

dove, se $0 \le v_i \le r - 1$ allora $x = 1$, $y = 2$, mentre se $r \le v_i \le 2r - 1$, allora $x = 2$, $y = 1$. $K(v_i)$ può essere vuoto, ma non può contenere più chiavi del grado di v_i , anzi spesso ne contiene di meno. L'algoritmo mantiene i vertici non selezionati che sono adiacenti ai vertici selezionati in un heap VHEAP che può essere implementato in vari modi. E' desiderabile avere livelli che sono più piccoli possibile e avere quelli più numerosi all'inizio dell'ordinamento: questo suggerisce che i vertici di gradi più alto devono essere esaminati per primi durante l'ordinamento. Inoltre deve essere semplice e veloce il metodo di scelta del vertice successivo e deve essere garantito il corretto funzionamento anche se il grafo non è connesso. Il tempo impiegato dall'ordering step è $O(n)$.

Il Searching step

Il searching step esamina i livelli prodotti dall'ordering step e assegna i valori hash alle chiavi, un livello alla volta. Assegnare valori hash alle chiavi significa assegnare i valori alla funzione g. E' preferibile adattare prima i livelli con più chiavi, poiché tutte le locazioni all'inizio sono vuote, poi di seguito tutti gli altri fino ai livelli contenenti una sola

chiave. La probabilità di adattare un campione di taglia *j* in una tabella con *m* slots con *f* slots già occupati è data da

$$\Pr(\text{fit}) = 1 - e^{-\mu}$$

dove

$$\mu = \left(\left(1 - \frac{f}{m} \right)^{j-1} (m - f) \right).$$

Se *f* è funzione di *m* tale che $f < (1 - \varepsilon)m$ per una certa costante $\varepsilon > 0$, allora $\mu \to \infty$ e $\Pr(\text{fit}) \to 1$. Di solito ci possono essere più valori di *g* che piazzano tutte le chiavi di $K(v_i)$ in degli slots non assegnati, comunque il valore di *g* viene scelto *a caso* tra tutti i possibili valori. Cercando un valore appropiato per *g*, il searching step usa una sequenza di accesso casuale per accedere agli slots 0,..., *n* - 1 della tabella hash. All'inizio vengono scelti 20 piccoli numeri primi che non dividono *n*, di questi uno è scelto a caso per essere s_1 ed usato come incremento per ottenere i rimanenti s_j, $j \geq 2$. Così la sequenza di accesso casuale è

$$0, q, 2q, 3q, \ldots, (n - 1)q$$

dove ognuno di questi numeri deve essere inteso *modulo n*. Il primo valore di *g* viene assegnato a caso senza esaminare alcun livello in quanto non assegna nessun valore hash alle chiavi, gli altri vengono assegnati in modo tale che il campione da inserire possa essere allocato nella tabella hash. Se il searching step trova un livello che non è capace di inserire nella tabella hash, allora viene usato un semplice schema di backtraking: nuovi valori di *g* vengono assegnati ai precedenti vertici che erano già stati definiti; si continua così finché tutti i livelli non vengono assegnati alle loro rispettive locazioni. Comunque è molto improbabile che accada ciò per un valore di *n* abbastanza elevato e per un'appropriata scelta di *r*. Il tempo impiegato dal searching step è $O(n)$. Il tempo complessivo impiegato dall'algoritmo è quindi $O(n)$.

Risultati sperimentali

Confronto con il test degli autori

La prima prova da me eseguita sull'algoritmo ha lo scopo di fare un confronto con i dati già raccolti dagli autori dell'articolo: il massimo valore di *n* che ho potuto testare è 1024 con lo stesso valore del parametro *r* dell'articolo, cioè ***r = 0.3n***, il che significa 0.6 words per chiave. I tempi sono espressi in secondi ed arrotondati alla seconda cifra decimale. Il computer utilizzato è un 486 DX2 a 66 Mhz con 8Mb di memoria. I risultati ottenuti sono riportati nelle seguenti tabelle dove per ogni valore di *n* ho eseguito 10 prove ed ho riportato la media nell'ultima colonna:

***n* = 32**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	0.05	0	0	0	0.05	0.05	0	0	0.05	0.05	0.02
Ordering	0	0	0	0	0	0	0	0	0	0	0
Searching	0	0.05	0.11	0.05	0	0.05	0.05	0.05	0	0	0.04
Totale	0.05	0.05	0.11	0.05	0.05	0.11	0.05	0.05	0.05	0.05	0.06

Tabella 1: *n* = 32

***n* = 64**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	0.05	0.05	0	0	0	0	0	0	0	0	0.01
Ordering	0	0	0	0	0	0	0	0	0	0	0
Searching	0.16	0.16	0.05	0.05	0.11	0.05	0.27	0.16	0.11	0.11	0.12
Totale	0.21	0.21	0.05	0.05	0.11	0.05	0.27	0.16	0.11	0.11	0.13

Tabella 2: *n* = 64

***n* = 128**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
Ordering	0	0	0	0	0	0	0	0	0	0	0
Searching	0.16	0.33	0.05	0.05	0.05	0.22	0.05	0.44	0.11	0.05	0.15
Totale	0.21	0.38	0.10	0.10	0.10	0.27	0.10	0.49	0.16	0.10	0.20

Tabella 3: *n* = 128

***n* = 256**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	0	0	0.05	0.05	0.05	0.05	0	0.05	0.05	0.05	0.04
Ordering	0	0	0	0	0	0	0	0	0	0	0
Searching	0.33	0.27	0.27	0.77	0.27	0.27	0.27	0.05	0.27	0.49	0.33
Totale	0.33	0.27	0.32	0.82	0.32	0.32	0.27	0.10	0.32	0.54	0.37

Tabella 4: *n* = 256

***n* = 512**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	0.05	0.05	0.05	0	0.05	0.05	0	0.05	0.05	0.05	0.04
Ordering	0	0.05	0	0.05	0	0	0.05	0	0.05	0	0.02
Searching	1.10	1.04	1.04	0.99	1.04	1.04	0.99	1.10	0.93	0.93	1.02
Totale	1.15	1.14	1.09	1.04	1.09	1.09	1.04	1.15	1.03	0.98	1.08

Tabella 5: *n* = 512

n = 1024	1	2	3	4	5	6	7	8	9	10	Media
Mapping	0.05	0.05	0.05	0.11	0.05	0.05	0.05	0.05	0.05	0.11	0.06
Ordering	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
Searching	3.52	3.74	3.57	3.79	3.90	4.12	3.90	4.01	3.46	3.74	3.78
Totale	3.68	3.90	3.73	4.01	4.06	4.28	4.06	4.17	3.62	3.96	3.95

Tabella 6: *n* = 1024

Di seguito riporto la tabella riassuntiva di tutti i valori medi:

n	Bits/key	Mapping	Ordering	Searching	Totale
32	3.0	0.02	0	0.04	0.06
64	3.6	0.01	0	0.12	0.13
128	4.2	0.05	0	0.15	0.20
256	4.8	0.04	0	0.33	0.37
512	5.4	0.04	0.02	1.02	1.08
1024	6.0	0.06	0.11	3.78	3.95

Tabella 7: riassunto dei valori medi

Ad un primo esame si può subito notare che non tutti i valori riportati sono significativi: in quasi tutte le tabelle compaiono degli zeri insieme a dei valori che si ripetono sempre uguali, come per esempio lo 0.05. Questo è dovuto al fatto che il computer è troppo veloce per quel numero di chiavi e che quindi *servirebbe un numero maggiore di chiavi per poter misurare dei tempi significativi*. Il mapping step e l'ordering step non assumo mai dei valori significativi: anche quando il loro tempo non è nullo, il valore è sempre troppo piccolo e per lo più costante: si può pensare che tali valori siano indipendenti dal numero di chiavi, cioè che tale numero non influisce sui loro tempi. Comunque i tempi del searching step appaiono significativi e dipendenti dal numero di chiavi, così come i tempi totali. Con tali tempi si può tentare un confronto con i risultati ottenuti dagli autori. Si vede che i tempi che ho misurato per il searching step sono superiori a quelli degli autori e tendono a crescere in modo quadratico e non lineare: questo può essere dovuto al fatto che la sequenza di accesso casuale viene molto spesso scandita per una buona parte dei suoi valori, in quanto *ho verificato che per r = 0.3n non entra mai in gioco il backtracking*. Tre sono i miglioramenti che mi sono venuti in mente:

- invece di generare tutta la sequenza casuale di accesso ogni volta che un livello è stato accomodato, si generano solo i valori di accesso che sono effettivamente necessari;
- invece di passare al searching step i vertici ordinati da cui poi esso deve trarre i vari livelli di chiavi, si passano già le chiavi divise in livelli spostando questa operazione nell'ordering step;

- occorre una sequenza di accesso casuale più robusta, cioè con una casualità più elevata.

Osservando attentamente il searching step si può capire che è facile avvicinarsi ad un comportamento quadratico: all'interno del ciclo for, che scandisce tutti i vertici, troviamo la generazione della sequenza casuale, la sua scansione, un ciclo for che si ripete per ogni chiave del livello e il dover cercare le chiavi che appartengono a tale livello. Per stare lontani da un comportamento quadratico occorre scandire meno termini possibili della sequenza di accesso casuale e questo può essere fatto generando una più robusta sequenza di accesso. Nel caso in cui il searching step incontri un livello che non è capace di accomodare, entra in gioco il backtracking e i tempi assumono un comportamento molto vicino a quello quadratico: comunque per un appropriato valore di r è molto improbabile che ciò si verifichi. I tempi riportati dagli autori sembrano un po' troppo ottimistici, visto anche il computer usato.

Verso il limite inferiore di spazio: il Mapping step

Per riuscire a misurare dei tempi significativi per il mapping step, ho pensato di spingere l'algoritmo verso il suo limite inferiore di spazio: fissato il numero di chiavi $\boldsymbol{n = 1024}$, ho fatto variare r fino a quando ho ottenuto dei tempi abbastanza alti. Diminuire lo spazio significa diminuire il valore di r ($|h| = 2r$): diminuendo r, diminuisco il numero di vertici e quindi aumento la probabilità di collisione delle triple; il tempo impiegato dal mapping step cresce di conseguenza. I risultati ottenuti sono riportati nelle seguenti tabelle dove per ogni valore di r sono state eseguite 10 prove; si tiene anche conto del numero di collisioni avvenute:

$r = 512$	1	2	3	4	5	6	7	8	9	10	Media
Mapping	0.05	0.05	0.05	0	0.05	0.05	0.05	0.05	0.05	0.05	0.05
Collisioni	0	0	0	0	0	0	0	0	0	0	0

Tabella 8: $r = 512$

$r = 256$	1	2	3	4	5	6	7	8	9	10	Media
Mapping	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
Collisioni	0	0	0	0	0	0	0	0	0	0	0

Tabella 9: $r = 256$

$r = 192$	1	2	3	4	5	6	7	8	9	10	Media
Mapping	0.05	0.05	0.05	0.05	0	0.05	0.05	0.05	0.05	0.05	0.05
Collisioni	0	0	0	0	0	0	0	0	0	0	0

Tabella 10: $r = 192$

$r = 160$	1	2	3	4	5	6	7	8	9	10	Media
Mapping	0.05	0.05	0.05	0.16	0.05	0.05	0.05	0.05	0.27	0.05	0.08
Collisioni	0	0	0	1	0	0	0	0	2	0	0

Tabella 11: *r* = 160

***r* = 128**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	0.88	0.16	5.60	0.16	0.93	0.82	0.38	0.77	3.90	0.05	1.37
Collisioni	7	1	49	1	8	7	3	6	34	0	12

Tabella 12: *r* = 128

***r* = 115**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	17.42	16.32	42.03	15.82	41.92	1.65	13.35	3.46	6.54	3.57	16.21
Collisioni	154	144	374	141	374	14	118	30	58	31	144

Tabella 13: *r* = 115

***r* = 110**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Mapping	65.22	53.79	54.73	264.23	84.89	34.07	100.27	140.51	11.16	13.79	82.27
Collisioni	580	478	488	2236	755	302	891	1189	111	138	717

Tabella 14: *r* = 110

Di seguito riporto la tabella riassuntiva di tutti i valori medi:

r	**bits/key**	**Mapping**	**Collisioni**
512	10	0.05	0
256	5	0.05	0
192	3.75	0.05	0
160	3.12	0.08	0.3
128	2.5	1.37	12
115	2.25	16.21	144
110	2.15	82.27	717

Tabella 15: riassunto dei valori medi

Si può subito osservare che fino ad $r = 192$, l'algoritmo non risente della riduzione del valore di r. Da tale valore in poi si iniziano ad avere delle collisioni ed il tempo impiegato dal mapping step cresce in modo esponenziale, così come le collisioni.

Verso il limite inferiore di spazio: l'Ordering step

La tabella riassuntiva dei valori medi è la seguente dove si è usato ***n = 1024***:

r	**bits/key**	**Ordering**	**Connessione**
512	10	0.16	No 26
256	5	0.08	1/5
192	3.75	0.06	Si
160	3.12	0.05	Si
128	2.5	0.04	Si
115	2.25	0.04	Si
110	2.15	0.03	Si

Tabella 16: riassunto dei valori medi

L'effetto prodotto dalla riduzione di r è opposto a quello precedente: i tempi diminuiscono al diminuire di r. Questo può essere spiegato se si tiene conto che con il diminuire di r, diminuiscono anche i vertici da ordinare (il numero di vertici è proprio $2r$).

Si osserva anche che per $r = 512$ il grafo non è connesso ed è diviso mediamente in 26 sottografi, per $r = 256$ solo un grafo su 5 non è connesso ed è diviso mediamente in 1 o 2 sottografi e infine per $r \geq 192$ il grafo è sempre connesso. Un grafo connesso aumenta la velocità dell'algoritmo, in quanto non si devono andare a cercare vertici non selezionati di grado maggiore di zero al di fuori di VHEAP, scandendo tutta la tabella dei vertici.

Verso il limite inferiore di spazio: il Searching step

Riportiamo i tempi misurati per ***n = 1024*** al variare di *r* nelle seguenti tabelle:

***r* = 512**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Searching	4.95	4.62	4.84	4.62	4.40	4.89	4.56	4.01	4.51	4.56	4.60
Backtracking	0	0	0	0	0	0	0	0	0	0	0

Tabella 17: *r* = 512

***r* = 380**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Searching	3.85	3.52	4.23	3.85	3.68	4.29	3.74	4.07	3.68	3.85	3.88
Backtracking	0	0	0	0	0	0	0	0	0	0	0

Tabella 18: *r* = 380

***r* = 320**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Searching	3.57	3.68	3.90	3.62	4.07	3.41	3.68	3.79	3.51	3.57	3.68
Backtracking	0	0	0	0	0	0	0	0	0	0	0

Tabella 19: *r* = 320

***r* = 288**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Searching	4.01	4.07	4.23	4.01	8.13	4.12	4.18	4.45	4.40	4.34	4.59
Backtracking	0	0	0	0	1	0	0	0	0	0	0.1

Tabella 20: *r* = 288

***r* = 256**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Searching	12.14	9.95	8.52	11.43	41.48	84.34	15.05	4.56	85.99	14.73	28.82
Backtracking	2	1	1	2	9	21	3	0	22	3	6

Tabella 21: *r* = 256

***r* = 248**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
Searching	12.25	28.30	18.85	19.78	15.82	16.61	58.02	297.5	8.57	34.73	51.05
Backtracking	2	6	4	4	3	4	15	69	1	8	12

Tabella 22: *r* = 248

La tabella riassuntiva dei valori medi è la seguente:

r	**bits/key**	**Searching**	**Backtracking**
512	10	4.60	0
380	7.42	3.88	0
320	6.25	3.68	0
288	5.63	4.59	0.1
256	5	28.82	6
248	4,84	51.05	12

Tabella 23: riassunto dei valori medi

Salta subito all'occhio la diminuzione del tempo al diminuire di *r* per i primi tre valori: questo può essere spiegato con il fatto che i vertici diminuiscono, mentre la scansione della sequenza di accesso rimane breve. Per $r = 288$ il tempo aumenta: la sequenza di accesso è molto scandita ed appare il primo backtracking. Da ora in poi i tempi aumentano velocemente: il backtracking avviene più volte; questo perché i vertici diminuiscono, i gradi aumentano e i livelli contengono più chiavi, quindi è più difficile adattarli nelle locazioni, soprattutto se non ci sono livelli che contengono poche chiavi. Inoltre ho testato l'algoritmo per $r = 240$: in 10 prove i tempi sono risultati sempre al di sopra dei 300 secondi. Per $r = 256$ si notano dei tempi piuttosto alti ed anomali, accanto a dei tempi accettabili: per ovviare a questo inconveniente, lo schema di backtracking dovrebbe far rinizializzare l'algoritmo dal mapping step, invece di tentare per molto tempo di adattare tali livelli. Il searching step è il più sensibile alle variazioni di *r*,il mapping step è meno sensibile del searching step ed infine l'ordering step assume un comportamento contrario rispetto ai precedenti due.

```
/*****************************************************************************************

        PRATICAL MINIMAL PERFECT HASH FUNCTIONS FOR LARGE DATABASES

*****************************************************************************************/

#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define MAXNUMKEY          1024          // massimo numero delle chiavi
#define MAXERRE  MAXNUMKEY / 2           // massimo valore di r: r <= n / 2
#define MAXLENKEY            20          // massima lunghezza di una chiave
#define ALFABETO             36          // numero dei caratteri (lettere e numeri) che compongono le parole
#define MAXNUMCAS         32767          // massimo numero casuale generato
#define TRUE                  1          // variabile booleana
#define FALSE                 0          // variabile booleana
#define MAXDEGREE            15          // massimo grado di un nodo stabilito a priori per l'inizializzazione
                                         // del vettore che testa la diversità delle triple nel searching_step
#define NUMPRIMI             20          // numero dei numeri primi usati dal searching_step

/*********************************    DEFINIZIONE DEI TIPI    *********************************/

typedef int   Long;                      // potrebbe essere sostituito con un long int

typedef int   Boolean;                   // Boolean: TRUE = vero = 1, FALSE = falso = 0

typedef Long  Ttable[ALFABETO][MAXLENKEY];       // tipo tabella dei numeri casuali

typedef char *Tsetkey[MAXNUMKEY];        // tipo  insieme delle chiavi mie

typedef struct vertici {                 // tipo tabella dei vertici
        Long  firstedge;                 // prima chiave che finisce nel vertice
        int   degree;                    // numero dei nodi incidenti
        Long  ordering;                  // vertici ordinati da ordering_step
        Long  g;                         // valore della g:{0,.,2 * ERRE - 1} -> {0,.,n - 1}
        } Tvertex[2 * MAXERRE];

typedef struct chiavi {                  // tipo tabella delle chiavi
        Long  h0, h1, h2;                // valori delle tre funzioni h0, h1, h2
        Long  nextedge1, nextedge2;      // prossima chiave dopo firstedge che finisce nel vertice
        } Tedge[MAXNUMKEY];

typedef struct heap {                    // tipo heap: serve all'ordering_step
        Long  vertici;                   // vertici che stanno nell'heap
        Boolean selected;                // serve all'ordering_step
        } Theap[2 * MAXERRE];

typedef struct tabella {                 // tipo tabella del searching_step
        Long  key;                       // numero della chiave
        Boolean assigned;                // dice se quella posizione è stata assegnata o no: TRUE o FALSE
        } Thash_table[MAXNUMKEY];
```

```
/*********************************     VARIABILI GLOBALI     *********************************/

Ttable   tableh0, tableh1, tableh2;          // sono le tre tabelle di numeri casuali per ogni possibile carattere
                                             // in ogni posizione i della chiave
Tsetkey  setkey;                             // setkey: insieme delle chiavi S
Tvertex  vertex;                             // tabella dei vertici
Tedge    edge;                               // tabella delle chiavi
Long     NUMKEY;                             // numero di chiavi usate
Long     ERRE;                               // numero di vertici usati

/*****************************     DEFINIZIONE DELLE FUNZIONI     *****************************/

void creasetkey(void);                       // crea l'insieme delle chiavi leggendolo da file

void costruiscitabella(Ttable tabella);      // costruisce una tabella di numeri casuali

Long h0(char *chiave);                       // è la funzione h0
Long h1(char *chiave);                       // è la funzione h1
Long h2(char *chiave);                       // è la funzione h2

void mapping_step(void);                     // è il mapping_step: costruisce Tvertex vertex e Tedge edge

void inizialize(Theap vheap);                // inizializza vheap e vertex[ ].ordering

Long cercamax(Theap vheap);                  // cerca il vertice di grado massimo tra quelli non selezionati in
                                             // vertex[ ].degree e non in vheap e restituisce la posizione

void insert(Long w, Theap vheap);            // inserisce un vertice in vheap

Long deletemax(Theap vheap);                 // cancella e restituisce il vertice di grado massimo da vheap

Boolean isinvheap(Theap vheap, Long w);      // cerca w in vheap: restituisce TRUE se c'è, FALSE se non c'è

Boolean evuoto(Theap vheap);                 // TRUE se vheap è vuoto, FALSE se vheap è pieno

Boolean isvertex(Theap vheap);               // esamina se c'è qualche vertice che non è stato selezionato e che
                                             // non è finito in vheap: può accadere se il grafo non è connesso

void ordering_step(void);                    // è l'ordering_step: ordina i vertici di grado > 0

void generaprimi(int vetprimi[ ]);           // genera 20 numeri primi che non dividono NUMKEY

void generasequenza(Long sequenza[ ]);       // genera la sequenza di accesso alla tabella finale hash_table

void searching_step(void);                   // è il searching_step

void savetableh0(void);                      // salva la tabella h0 nel file tableh0.dat
void savetableh1(void);                      // salva la tabella h1 nel file tableh1.dat
void savetableh2(void);                      // salva la tabella h2 nel file tableh2.dat

void savegvalues(void);                      // salva la funzione g:{0,.,2ERRE-1}->{0,.,NUMKEY-1} nel file
                                             // g.dat

void loadtableh0(void);                      // carica la tabella h0 in memoria
void loadtableh1(void);                      // carica la tabella h1 in memoria
void loadtableh2(void);                      // carica la tabella h2 in memoria

void loadgvalues(void);                      // carica i valori della g in memoria
```

```
/*****************************     INIZIO PROGRAMMA PRINCIPALE     *****************************/

void main(void)
{
	Long indirizzo;
	char c, key[MAXLENKEY];
	clock_t start, end, inizio, fine;

	clrscr();
	gotoxy(11, 4);
	printf("FUNZIONI HASH PERFETTE E MINIMALI PER GRANDI BASI DI DATI");
	gotoxy(1, 7);
	printf("Il programma genera una funzione hash perfetta e minimale partendo dall'insieme\ndelle chiavi e la salva sottoforma di tabelle nei seguenti files: tableh0.dat,\ntableh1.dat, tableh2.dat, g.dat.\nE'suddiviso in tre parti: mapping step, ordering step, searching step.\nLa prima crea le strutture dati su cui lavorare, la seconda ordina tali dati\nsecondo un particolare algoritmo e la terza assegna i valori hash alle chiavi.\nGioca un ruolo fondamentale la casualità…");
	printf("\nInserisci il numero di chiavi (massimo 1024): ");
	scanf("%d", &NUMKEY);
	printf("Inserisci il valore di r: "); scanf("%d", &ERRE);
	printf("Scegli tra:");
	printf("\n1) Costruzione della tabella hash e ricerca di un elemento");
	printf("\n2) Ricerca di un elemento nella tabella");
	gotoxy(1, 25);
	do
	{
		c = getch();
		if (c != '1' && c != '2')
			printf("\a");
	} while (c != '1' && c != '2');
	gotoxy(1, 18);
	switch (c)
	{
		case '1':
			randomize();
			creasetkey();
			printf("\nCostruzione della tabella hash in corso... ");
			inizio = clock();
			start = clock();
			mapping_step();
			end = clock();
			printf("\nMapping step per %d chiavi: %f secondi", NUMKEY, (end - start) / CLK_TCK);
			start = clock();
			ordering_step();
			end = clock();
			printf("\n'Ordering step per %d chiavi: %f secondi", NUMKEY, (end - start) / CLK_TCK);
			start = clock();
			searching_step();
			end = clock();
			printf("\nSearching step per %d chiavi: %f secondi", NUMKEY, (end - start) / CLK_TCK);
			fine = clock();
			printf("\nTempo totale per %d chiavi: %f secondi", NUMKEY, (fine - inizio) / CLK_TCK);
			savetableh0();
			savetableh1();
			savetableh2();
			savegvalues();
		case '2':
			loadtableh0();
			loadtableh1();
			loadtableh2();
			loadgvalues();
			do
```

```
                {
                        printf("\nInserisci la chiave da ricercare: ");
                        scanf("%s", key);
                        indirizzo = (h0(key) + vertex[h1(key)].g + vertex[h2(key)].g) % NUMKEY + 1;
                        printf("L'indirizzo della chiave è %d", indirizzo);
                        printf("\nVuoi ricercare un'altra chiave (s/n)? ");
                        do
                        {
                                c = getch();
                                if (c != 's' && c != 'n')
                                        printf("\a");
                        } while (c != 's' && c != 'n');
                        if (c == 's')
                                printf("s\n");
                        else
                                printf("n");
                } while (c == 's');
        }
}

/*****************************     FINE PROGRAMMA PRINCIPALE     ****************************/

/*********************     FUNZIONE CHE CARICA L'INSIEME DELLE CHIAVI     *********************/

void creasetkey(void)
{
        Long i;
        FILE *fp;
        char s[MAXLENKEY], file[13];

        printf("\nNome del file da cui caricare le chiavi: ");
        scanf("%s", file);
        if ((fp = fopen(file, "r")) == NULL)
        {
                fprintf(stderr, "\nNon posso aprire %s", file);
                exit(1);
        }
        else
        {
                for (i = 0; i < NUMKEY; ++i)
                {
                        fscanf(fp, "%s", s);
                        setkey[i] = (char *)malloc(strlen(s) + 1);
                        if (setkey[i] != NULL)
                                strcpy(setkey[i], s);
                        else
                                printf("\nImpossibile allocare la memoria");
                }
                fclose(fp);
        }
        if (ferror(fp))
        {
                fprintf(stderr, "\nErrore in lettura su %s", file);
                exit(2);
        }
}
```

```
/*************************      FUNZIONI USATE DAL MAPPING STEP      **************************/

void costruiscitabella(Ttable t)
{
        int i, j;

        for (i = 0; i < ALFABETO; ++i)
                for (j = 0; j < MAXLENKEY; ++j)
                        t[i][j] = random(MAXNUMCAS);
}

Long h0(char *k)
{
        long somma, m;
        int i;

        somma = 0;
        for (i = 0; k[i] != '\0'; ++i)
                somma += tableh0[(isalpha(k[i])) ? (islower(k[i]) ? k[i] - 'a' : k[i] - 'A') : k[i] - '0' + 26][i];
        m = somma % NUMKEY;
        return (int)m;
}

Long h1(char *k)
{
        long somma, m;
        int i;

        somma = 0;
        for (i = 0; k[i] != '\0'; ++i)
                somma += tableh1[(isalpha(k[i])) ? (islower(k[i]) ? k[i] - 'a' : k[i] - 'A') : k[i] - '0' + 26][i];
        m = somma % ERRE;
        return (int)m;
}

Long h2(char *k)
{
        long somma, m;
        int i;

        somma = 0;
        for (i = 0; k[i] != '\0'; ++i)
                somma += tableh2[(isalpha(k[i])) ? (islower(k[i]) ? k[i] - 'a' : k[i] - 'A') : k[i] - '0' + 26][i];
        m = somma % ERRE + ERRE;
        return (int)m;
}
```

```
/***********************************     MAPPING STEP     ***********************************/

void mapping_step(void)
{
        Long v, i;
        Boolean collision;

        do
        {
                /* 1 */                                 // costruzione delle 3 tabelle di numeri casuali
                costruiscitabella(tableh0);
                costruiscitabella(tableh1);
                costruiscitabella(tableh2);
                /* 2 */                                 // inizializzazione
                for (v = 0; v < 2 * ERRE; ++v)
                {
                        vertex[v].firstedge = -1;       // -1 perchè le chiavi vanno da 0 a NUMKEY - 1
                        vertex[v].degree = 0;
                }
                /* 3 */                                 // si costruiscono le tabelle vertex ed edge
                for (i = 0; i < NUMKEY; ++i)
                {
                        edge[i].h0 = h0(setkey[i]);
                        edge[i].h1 = h1(setkey[i]);
                        edge[i].h2 = h2(setkey[i]);
                        edge[i].nextedge1 = 0;
                        if (vertex[edge[i].h1].firstedge == -1)
                                vertex[edge[i].h1].firstedge = i;
                        else
                        {
                                Long j = vertex[edge[i].h1].firstedge;
                                while (edge[j].nextedge1 != 0)
                                        j = edge[j].nextedge1;
                                edge[j].nextedge1 = i;
                        }
                        ++vertex[edge[i].h1].degree;
                        edge[i].nextedge2 = 0;
                        if (vertex[edge[i].h2].firstedge == -1)
                                vertex[edge[i].h2].firstedge = i;
                        else
                        {
                                Long j = vertex[edge[i].h2].firstedge;
                                while (edge[j].nextedge2 != 0)
                                        j = edge[j].nextedge2;
                                edge[j].nextedge2 = i;
                        }
                        ++vertex[edge[i].h2].degree;
                }
                /* 4 */                                 // si esaminano tutte le chiavi con lo stesso valore h1
                collision = FALSE;                      // con grado > di 1 per testare la diversità delle triple
                for (v = 0; v < ERRE; ++v)
                        if (vertex[v].degree > 1)
                        {
                                Long j = vertex[v].firstedge;
                                Long vet[2][MAXDEGREE];
                                int x, x1;
                                for (x = 0; x < 2; ++x)
                                        for (x1 = 0; x1 < MAXDEGREE; ++x1)
                                                vet[x][x1] = -1;
                                x1 = 0;
                                do                      // si costruisce vet[ ][ ]
                                {
```

```
                                        vet[0][x1] = edge[j].h0;
                                        vet[1][x1] = edge[j].h2;
                                        ++x1;
                                        j = edge[j].nextedge1;
                                } while (j != 0);
                                for (x = 0; vet[0][x] != -1; ++x)
                                        for (x1 = x + 1; vet[0][x1] != -1; ++x1)
                                                if (vet[0][x] == vet[0][x1] && vet[1][x] == vet[1][x1])
                                                {
                                                        collision = TRUE;
                                                        v = ERRE;
                                                }
                        }
        } while (collision);
}

/**********************************      FINE MAPPING STEP      **********************************/

/************************      FUNZIONI USATE DALL'ORDERING STEP      ************************/

void inizialize(Theap vheap)
{
        Long v;

        for (v = 0; v < 2 * ERRE; ++v)
        {
                vertex[v].ordering = -1;
                vheap[v].vertici = -1;
                vheap[v].selected = FALSE;
        }
}

Long cercamax(Theap vheap)              // cerca il vertice di grado massimo tra quelli non selezionati
{
        Long p, v;                      // p è il vertice di grado massimo tra quelli non selezionati
        int  g;                         // grado del vertice di grado massimo

        p = 0;
        g = 0;
        for (v = 0; v < 2 * ERRE; ++v)
                if (g < vertex[v].degree && vheap[v].selected == FALSE)
                {
                        g = vertex[v].degree;
                        p = v;
                }
        return p;
}

void insert(Long w, Theap vheap)        // vheap[ ].vertici è tutto a -1
{
        Long v;

        for (v = 0; vheap[v].vertici != -1; ++v)
                ;
        vheap[v].vertici = w;
}
```

```
Long deletemax(Theap vheap)
{
	Long v, ver;
	Long p;				// è la posizione all'interno di vheap del vertice di grado massimo
	int  g;				// è il grado del vertice di grado massimo

	p = 0;				// si cerca il vertice di grado massimo in vheap
	g = 0;
	for (v = 0; vheap[v].vertici != -1; ++v)
		if (g < vertex[vheap[v].vertici].degree)
		{
			g = vertex[vheap[v].vertici].degree;
			p = v;
		}
	ver = vheap[p].vertici;
	for (v = p; vheap[v].vertici != -1; ++v)		// si cancella il vertice p
	vheap[v].vertici = vheap[v + 1].vertici;
	return ver;					// si restituisce il vertice di grado massimo
}

Boolean isinvheap(Theap vheap, Long w)
{
	Long v;

	for (v = 0; vheap[v].vertici != -1; ++v)
		if (vheap[v].vertici == w)
			return TRUE;
	return FALSE;
}

Boolean evuoto(Theap vheap)
{
	if (vheap[0].vertici == -1)
		return TRUE;
	return FALSE;
}

Boolean isvertex(Theap vheap)
{
	Long v;

	for (v = 0; v < 2 * ERRE; ++v)
	    if (vheap[v].selected == FALSE && vertex[v].degree > 0)
	        return TRUE;
	return FALSE;
}
```

```
/**********************************      ORDERING STEP      **********************************/

void ordering_step(void)
{
        Boolean  cevertice, vuoto;
        Theap vheap;
        Long  w;
        Long  i;            // indice che indica la posizione di ordinamento: si parte da 0 e riguarda vertex[i].ordering

        inizialize(vheap);
        i = 0;
        cevertice = TRUE;
        while(cevertice)
        {
                w = cercamax(vheap);                    // non in vheap ma tra tutti i vertici
                insert(w, vheap);
                vuoto = FALSE;
                while (!vuoto)                          // finchè vheap non è vuoto
                {
                        vertex[i].ordering = deletemax(vheap);
                        vheap[vertex[i].ordering].selected = TRUE;
                        w = vertex[vertex[i].ordering].firstedge;       // è la prima chiave che cade nel vertice
                        do
                        {
                                if (vertex[i].ordering >= 0 && vertex[i].ordering <= ERRE - 1)
                                {
                                        if (!vheap[edge[w].h2].selected && !isinvheap(vheap, edge[w].h2))
                                                insert(edge[w].h2, vheap);
                                        w = edge[w].nextedge1;
                                }
                                else
                                {
                                        if (!vheap[edge[w].h1].selected && !isinvheap(vheap, edge[w].h1))
                                                insert(edge[w].h1, vheap);
                                        w = edge[w].nextedge2;
                                }
                        } while (w != 0);
                        ++i;
                        vuoto = evuoto(vheap);
                }
                cevertice = isvertex(vheap);
        }
}

/**********************************      FINE ORDERING STEP      **********************************/
```

```
/*************************      FUNZIONI USATE DAL SEARCHING STEP      *************************/

void generaprimi(int vetprimi[ ])                  // NUMPRIMI numeri primi che non dividono NUMKEY
{
        Boolean isprime;
        int i, j, k;

        for (i = 3, k = 0; i < 500 && k < NUMPRIMI; i += 2)
        {
                isprime = TRUE;
                for (j = 3; j <= (int)sqrt(i); ++j)
                        if (i % j == 0)
                        {
                                isprime = FALSE;
                                j = (int)sqrt(i) + 1;
                        }
                if (isprime && NUMKEY % i != 0)
                        vetprimi[k++] = i;
        }
}

void generasequenza(Long sequenza[ ])
{
        Long i, indice;
        int q;
        int vetprimi[NUMPRIMI];

        generaprimi(vetprimi);
        q = vetprimi[random(NUMPRIMI)];
        indice = random(NUMKEY);
        for (i = 0; i < NUMKEY; ++i)
        {
                long m;
                m = ((long)q * ((long)i + (long)indice)) % NUMKEY;
                sequenza[i] = (int)m;
        }
}
```

```
/**********************************     SEARCHING STEP     ***********************************/

void searching_step(void)
{
        Long i;                         // scandisce l'ordine dei vertici
        Long j;                         // scandisce la sequenza casuale di accesso
        Long k;                         // scandisce le chiavi dei vari livelli
        Long hk;                        // valore hash della chiave k
        Long sequenza[MAXNUMKEY];       // sequenza casuale di accesso a hash_table
        Thash_table hash_table;
        Boolean collision, fail;

        do
        {
                fail = FALSE;
                /* 1 */                                         // inizializzazione
                for (i = 0; i < NUMKEY; ++i)
                        hash_table[i].assigned = FALSE;
                for (i = 0; i < 2 * ERRE; ++i)
                        vertex[i].g = -1;
                vertex[vertex[0].ordering].g = random(NUMKEY); // assegnazione di un valore casuale al primo
                                                                // vertice
                /* 2 */                                         // scandisce l'ordine dei vertici escluso il 1°
                for (i = 1; i < 2 * ERRE && vertex[i].ordering != -1; ++i)
                {
                /* 3 */   generasequenza(sequenza);          // genera la sequenza casuale di accesso
                        j = 0;
                        do
                        {
                                k = vertex[vertex[i].ordering].firstedge;       // prima chiave che cade nel vertice
                                collision = FALSE;
                                if (vertex[i].ordering >= 0 && vertex[i].ordering <= ERRE - 1)
                                        do
                                        {
                                                Long w;
                                                for (w = 0; w < i; ++w)
                                                        if (vertex[w].ordering >= ERRE && vertex[w].ordering
                                                        <= 2 * ERRE -1 && vertex[w].ordering == edge[k].h2)
                                                        {
                /* 4 */                                         hk = (edge[k].h0 + vertex[edge[k].h2].g +
                                                                sequenza[j]) % NUMKEY;
                /* 5 */                                         if (hash_table[hk].assigned)
                                                                        collision = TRUE;
                                                        }
                                                k = edge[k].nextedge1;
                                        } while (k != 0 && !collision);
                                else    // if vertex[i].ordering appartiene a [r,.,2r-1]
                                        do
                                        {
                                                Long w;
                                                for (w = 0; w < i; ++w)
                                                        if (vertex[w].ordering >= 0 && vertex[w].ordering <=
                                                        ERRE -1 && vertex[w].ordering == edge[k].h1)
                                                        {
                                                                hk = (edge[k].h0 + vertex[edge[k].h1].g + se-
                                                                quenza[j]) % NUMKEY;
                                                                if (hash_table[hk].assigned)
                                                                        collision = TRUE;
                                                        }
                                                k = edge[k].nextedge2;
                                        } while (k != 0 && !collision);
                /* 6 */             if (!collision)
```

```
                                {
                                        k = vertex[vertex[i].ordering].firstedge; // prima chiave che cade nel vertice
                                        if (vertex[i].ordering >= 0 && vertex[i].ordering <= ERRE - 1)
                                                do
                                                {
                                                        Long w;
                                                        for (w = 0; w < i; ++w)
                                                                if (vertex[w].ordering >= ERRE && ver-
                                                                tex[w].ordering <= 2 * ERRE - 1 && ver-
                                                                tex[w].ordering == edge[k].h2)
                                                                {
                                                                        hk = (edge[k].h0 + vertex[edge[k].h2].g
                                                                        + sequenza[j]) % NUMKEY;
                                                                        hash_table[hk].assigned = TRUE;
                                                                        hash_table[hk].key = k;
                                                                }
                                                        k = edge[k].nextedge1;
                                                } while (k != 0);
                                        else    // if vertex[i].ordering appartiene a [r,.,2r-1]
                                                do
                                                {
                                                        Long w;
                                                        for (w = 0; w < i; ++w)
                                                                if (vertex[w].ordering >= 0 && ver-
                                                                tex[w].ordering <= ERRE -1 && ver-
                                                                tex[w].ordering == edge[k].h1)
                                                                {
                                                                        hk = (edge[k].h0 + vertex[edge[k].h1].g
                                                                        + sequenza[j]) % NUMKEY;
                                                                        hash_table[hk].assigned = TRUE;
                                                                        hash_table[hk].key = k;
                                                                }
                                                        k = edge[k].nextedge2;
                                                } while (k != 0);
                                        vertex[vertex[i].ordering].g = sequenza[j];
                                }
                                else
                                {
                                        ++j;
                        /* 7 */                 if (j > NUMKEY - 1)             // BACKTRAKING
                                        {
                                                fail = TRUE;
                                                collision = FALSE;      // serve ad uscire dal ciclo
                                                i = 2 * ERRE;           // serve ad uscire dal ciclo
                                        }
                                }
                        } while (collision);
                }
        } while (fail);
}

/*********************************     FINE SEARCHING STEP     *********************************/
```

```
/************************      FUNZIONI CHE SALVANO LE TRE TABELLE      ************************/

void savetableh0(void)
{
	int i, j;
	FILE *fp;

	if ((fp = fopen("tableh0.dat", "w")) == NULL)
	{
		fprintf(stderr, "\nNon posso scrivere su tableh0.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < ALFABETO; ++i)
		{
			for(j = 0; j < MAXLENKEY; ++j)
				fprintf(fp, "%3d ", tableh0[i][j]);
			fprintf(fp, "\n");
		}
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in scrittura su tableh0.dat");
		exit(2);
	}
}

void savetableh1(void)
{
	int i, j;
	FILE *fp;

	if ((fp = fopen("tableh1.dat", "w")) == NULL)
	{
		fprintf(stderr, "\nNon posso scrivere su tableh1.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < ALFABETO; ++i)
		{
			for(j = 0; j < MAXLENKEY; ++j)
				fprintf(fp, "%3d ", tableh1[i][j]);
			fprintf(fp, "\n");
		}
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in scrittura su tableh1.dat");
		exit(2);
	}
}
```

```
void savetableh2(void)
{
	int i, j;
	FILE *fp;

	if ((fp = fopen("tableh2.dat", "w")) == NULL)
	{
		fprintf(stderr, "\nNon posso scrivere su tableh2.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < ALFABETO; ++i)
		{
			for(j = 0; j < MAXLENKEY; ++j)
				fprintf(fp, "%3d ", tableh2[i][j]);
			fprintf(fp, "\n");
		}
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in scrittura su tableh2.dat");
		exit(2);
	}
}

/**************************	FUNZIONE CHE SALVA I VALORI DI G	**************************/

void savegvalues(void)
{
	Long i;
	FILE *fp;

	if ((fp = fopen("g.dat", "w")) == NULL)
	{
		fprintf(stderr, "\nNon posso scrivere su g.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < 2 * ERRE; ++i)
			fprintf(fp, "%3d\n", vertex[i].g);
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in scrittura su g.dat");
		exit(2);
	}
}
```

```
/**********************      FUNZIONI CHE CARICANO LE TRE TABELLE      **********************/

void loadtableh0(void)
{
	int i, j;
	FILE *fp;

	if ((fp = fopen("tableh0.dat", "r")) == NULL)
	{
		fprintf(stderr, "\nNon posso leggere tableh0.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < ALFABETO; ++i)
			for(j = 0; j < MAXLENKEY; ++j)
				fscanf(fp, "%d", &tableh0[i][j]);
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in lettura su tableh0.dat");
		exit(2);
	}
}

void loadtableh1(void)
{
	int i, j;
	FILE *fp;

	if ((fp = fopen("tableh1.dat", "r")) == NULL)
	{
		fprintf(stderr, "\nNon posso leggere tableh1.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < ALFABETO; ++i)
			for(j = 0; j < MAXLENKEY; ++j)
				fscanf(fp, "%d", &tableh1[i][j]);
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in lettura su tableh1.dat");
		exit(2);
	}
}
```

```
void loadtableh2(void)
{
	int i, j;
	FILE *fp;

	if ((fp = fopen("tableh2.dat", "r")) == NULL)
	{
		fprintf(stderr, "\nNon posso leggere tableh2.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < ALFABETO; ++i)
			for(j = 0; j < MAXLENKEY; ++j)
				fscanf(fp, "%d", &tableh2[i][j]);
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in lettura su tableh2.dat");
		exit(2);
	}
}

/*************************	FUNZIONE CHE CARICA I VALORI DI G	*************************/

void loadgvalues(void)
{
	Long i;
	FILE *fp;

	if ((fp = fopen("g.dat", "r")) == NULL)
	{
		fprintf(stderr, "\nNon posso leggere g.dat");
		exit(1);
	}
	else
	{
		for (i = 0; i < 2 * ERRE; ++i)
			fscanf(fp, "%d", &vertex[i].g);
		fclose(fp);
	}
	if (ferror(fp))
	{
		fprintf(stderr, "\nErrore in lettura su g.dat");
		exit(2);
	}
}
```