UNIVERSITÀ DEGLI STUDI DI TORINO

DIPARTIMENTO DI INFORMATICA

SCUOLA DI SCIENZE DELLA NATURA

Corso di Laurea in Informatica

LABORATORIO DI ALGORITMI E STRUTTURE DATI

RELAZIONE DI LABORATORIO

Riccardo Pala, Daniele Pira, Jacopo Tancredi

Anno accademico: 2015/2016

Contents

Dati

La seguente tabella mostra i risultati di alcune semplici analisi effettuate sui dati in input:

Tipo di dato	Valori distinti	Totale
String	822	20000000
Integer	4908200	20000000
Float	19997945	20000000

La prima colonna indica di quale tipo di dato si tratta, la seconda il numero di valori distinti assunti dagli elementi di quel tipo, mentre l'ultima colonna indica il numero totale di elementi.

Da una prima analisi dei dati possiamo quindi dedurre che tra le stringhe vi siano numerose ripetizioni. Anche fra gli interi ve ne è un buon numero. Mentre per i double esse sono trascurabili.

Ordinamento

La seguente tabella riporta i tempi di esecuzione (in secondi) dei diversi tipi di algoritmi di ordinamento implementati:

Algoritmo	**String**	**Integer**	**Double**
Quick (Tripartizione)	7.03	11.62	11.61
Quick (Libro di testo)	N/A	9.51	9.68
Merge	7.35	7.72	7.70
Heap	26.03	37.88	37.58
Insertion	N/A	N/A	N/A

Nel caso in cui un algoritmo non termini l'esecuzione entro 10 minuti viene riportato N/A.

2.1 Quicksort

Sono state utilizzate due implementazioni dell'algoritmo.

La prima effettua una tripartizione al posto della classica bipartizione. Inoltre utilizza la strategia di selezione del pivot chiamata *Median of three*, la quale consiste nel scegliere la mediana tra i tre elementi posti rispettivamente all'inizio, al centro ed alla fine della porzione di array su cui si effettuerà la partizione.

Osservando la tabella notiamo che il tempo di esecuzione della versione con tripartizione è significativamente ridotto nel caso in cui gli elementi da ordinare siano delle stringhe. L'ipotesi è che l'implementazione con tripartizione si comporta meglio nel caso di un grande numero di elementi ripetuti.

Per verificarla è stata creata una seconda versione che segue quella suggerita dal libro di testo del corso (con bipartizione e scelta "fissa" del pivot).

I risultati confermano la tesi.

2.2 Mergesort

I tempi di esecuzione del mergesort sono simili a quelli del quicksort.

Questo rispechia la teoria in quanto entrambi gli algoritmi hanno complessità Θ(n log n), dove n è la dimensione dell'input.

Confrontando le implementazioni, ci aspettiamo che il primo sia meno performante del secondo dato che lavora ricopiando reciprocamente i dati dall'array di partenza su uno di supporto (e viceversa). Sperimentalmente questo non succede per tre motivi: il mergesort, a differenza del quicksort, non soffre di sbilanciamento; il quicksort con tripartizione e m.o.t. risulta onerosa in termini di computazione; il mergesort riduce il "deficit" di copiatura delle porzioni intermedie avvalendosi di chiamate di sistema.

E' comunque sconsigliato usarlo per una grossa quantità di dati, in quanto non opera *in place*.

2.3 Heapsort

Anche in questo caso i risultati hanno un comportamento simile ai precedenti siccome l'heapsort ha una complessità nell'ordine di Θ(n log n).

Al contrario del quicksort gli elementi ripetuti non incidono sull'efficienza dell'algoritmo. Inoltre, al contrario del mergesort, esso non necessita di spazio aggiuntivo.

Questa garanzia sulle prestazione nel caso peggiore è, però, ottenuta al prezzo di cicli interni più complessi (vedi move up e move down), ed un numero di confronti più elevato. Motivo per cui impiega più tempo ad ordinare rispetto ai precedenti.

2.4 Insertion Sort

Come sappiamo la complessità asintotica dell'insertion sort è nell'ordine di $\Theta(n^2)$, dove n è la dimensione dell'input.

I risultati sperimentali concordano con tale affermazione, difatti l'esecuzione dell'algoritmo non termina entro i 10 minuti prestabiliti.

Dizionari

La seguente tabella riporta i tempi di esecuzione (in secondi) di alcune sequenze di operazioni su diverse strutture dati:

Struttura Dati	**String**	**Integer**	**Double**
Binary Tree (Insert)	1.98	29.77	46.32
Binary Tree (Search)	0.29	1.64	2.54
Binary Tree (Delete)	0.09	0.64	2.01
Hash Table (Insert)	3.04	7.07	17.25
Hash Table (Search)	0.26	0.33	0.42
Hash Table (Delete)	0.21	0.58	0.63
Prealloc Hash Table (Insert)	1.63	5.66	5.55
Prealloc Hash Table (Search)	0.23	0.28	0.37
Prealloc Hash Table (Delete)	0.22	0.50	0.57

Per ogni struttura dati sono stati effettuati 20 milioni di inserimenti, 1 milione di accessi ed 1 milione di cancellazioni.

3.1 Binary Tree

Possiamo constatare dai dati che i tempi di inserimento aumentano con il diminuire delle ripetizioni.

Come sappiamo in un dizionario non si possono avere due elementi con la stessa chiave. Ogniqualvolta si tenta di inserire un elemento la cui chiave è già presente nella struttura dati, verrà semplicemente aggiornato il valore associato ad essa. Di conseguenza l'altezza dell'albero non cambia.

Possiamo quindi dedurre che l'altezza dell'albero è inversamente proporzionale al numero di ripetizioni delle chiavi. Più in generale sarà $O(k)$ dove k è il numero di chiavi distinte. Motivo per cui il tempo di esecuzione delle operazioni di insert e delete diminuisce all'aumentare delle ripetizioni.

3.2 Hash Table

Come precedentemente detto, per i binary tree, anche in questo caso, i tempi di inserimento aumentano con il diminuire delle ripetizioni.

La nostra implementazione per le hash table, quando un predeterminato fattore di carico raggiunge una certa soglia (nel nostro caso 5), prevede il ridimensionamento automatico; inoltre non sono ammessi elementi ripetuti. Davanti a dataset con un gran numero di elementi duplicati, è lecito prevedere che vi siano meno ridimensionamenti, di consuguenza ci aspettiamo un tempo di esecuzione inferiore rispetto ad un input eterogeneo.

Abbiamo formulato varie ipotesi sul motivo per cui i tempi di esecuzione di search e delete nelle hash table seguono lo stesso trand:

- funzioni di hash differenti per ogni tipo (e quindi con complessita' diverse);
- distribuzione non uniforme degli elementi (per alcuni tipi);
- perdita di località dei dati in memoria.

3.2.1 Dati raccolti

I dati che sono stati raccolti per analizzare le hash table sono i seguenti:

	String	**Integer**	**Double**
Dimensione finale tavola	200	3125000	15625000
Lunghezza media liste	4.2	1.5	1,7

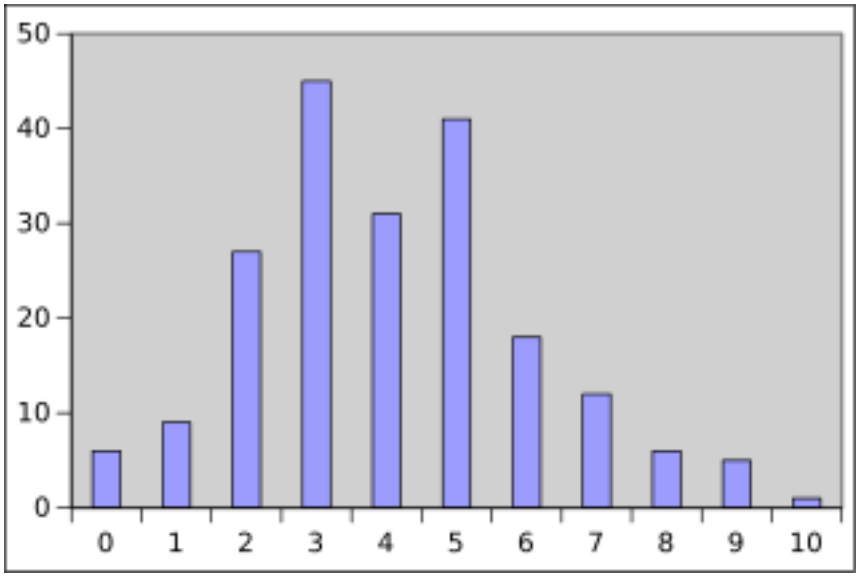

Figure 3.1: string data distribution

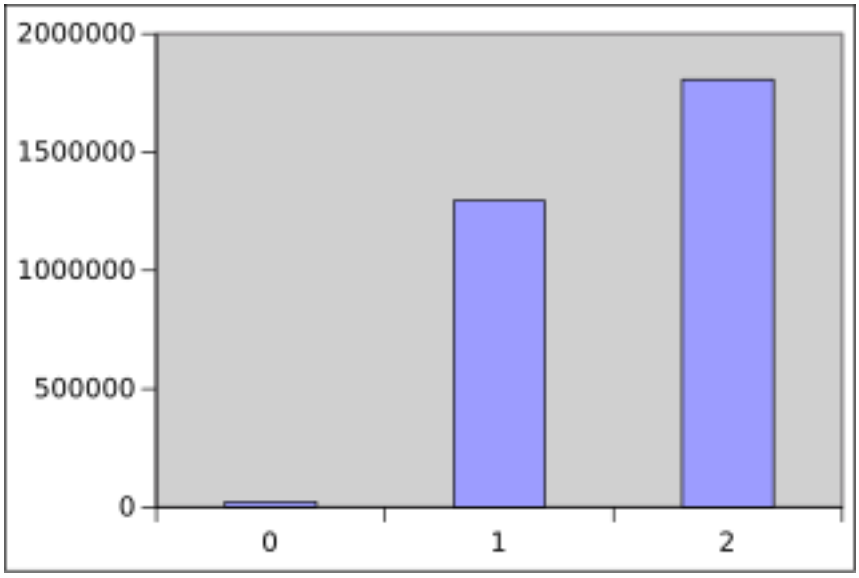

Figure 3.2: integer data distribution

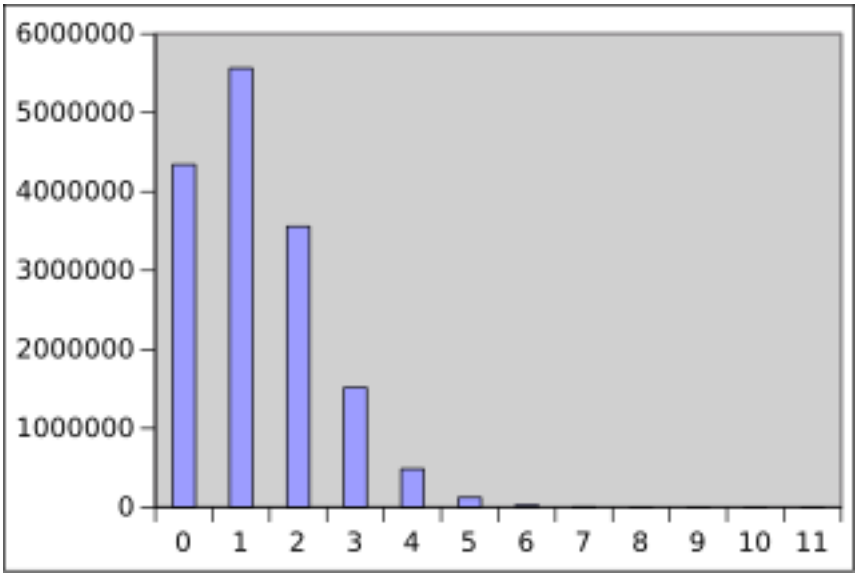

Figure 3.3: double data distribution

Grafi

4.1 Cammini Minimi

Nella seguente tabella vengono riportati i dati relativi all'applicazione dell'algoritmo di Dijkstra

Partenza	**Arrivo**	**Km**
Torino	Catania	1207.680
Torino	Borsoi	N/A
Torino	Sassari	639.883
Roma	Venezia	416.184
Perfugas	Bulzi	20.864

Nel caso in cui non esista un cammino minimo dalla località di partenza alla località di arrivo viene riportato N/A

4.2 Componenti Fortemente Connesse

Per calcolare il numero di componenti fortemente connesse abbiamo implementato l'algoritmo di Tarjan, avente complessità nell'ordine di $O(V + E)$.

Le componenti risultanti sono riportate nella seguente tabella

	SCC1	**SCC2**	**SCC3**
Dimensione	18636	2	2