Esercizi di Strutture Dati Algoritmi e Complessità

Luca Becchetti

May 20, 2024

1 Esercizi del 20 Maggio 2024

Ridimensionamento di una tabella hash dinamica [1, Capitolo 11 e Sezione 16.4]

Si consideri una tabella hash dinamica realizzata con array e nella quale le collisioni sono gestite con liste di trabocco: quando il fattore di carico della tabella supera il valore $1/2$ si copia il contenuto della tabella in una nuova tabella avente dimensione *doppia* rispetto alla precedente. Poiché la tabella è realizzata con un array, occorre riassegnare ciascun elemento presente al momento del raddoppio a una posizione del nuovo array usando una nuova funzione hash,[1] dunque con un costo proporzionale al numero di elementi presenti nella tabella al momento del raddoppio.

Quesito. Si supponga che la dimensione iniziale della tabella sia pari a 1. Si consideri una successione di n inserimenti in una tabella hash dinamica gestita come descritto sopra, supponendo che queste siano le uniche operazioni (quindi non vi sono cancellazioni). Dimostrare che il costo atteso per inserimento è $\mathcal{O}(1)$, suppondendo che le funzioni hash usate si comportino in modo ideale, ossia soddisfino le ipotesi di uniformità e indipendenza.

Risposta. Si può procedere sulla falsariga dell'analisi dell'espansione di una tabella presentata nella Sezione 16.4.1 del libro di testo, sebbene con qualche accortezza, considerato che l'analisi presentata nel libro si riferisce a tabelle (non tabelle hash) nelle quali gli inserimenti avvengono nella prossima posizione disponibile a costo costante.

Sia C_i il costo dell'i-esimo inserimento e sia $C = \sum_{i=1}^{n} C_i$ il costo complessivo. Avremo allora:

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{i=1}^{n} C_i\right] = \sum_{i=1}^{n} \mathbb{E}[C_i],$$

mentre il costo atteso per inserimento sarà pari a $\mathbb{E}[C]/n$. Il problema si riduce quindi a calcolare il generico $\mathbb{E}[C_i]$. Premesso che per la politica di raddoppio la capacità della tabella hash è sempre una potenza di 2, abbiamo due casi possibili.

Abbiamo un raddoppio. In tal caso, in occasione dell'i-esimo inserimento il fattore di carico ha superato il valore $1/2$ (mentre non lo aveva superato in occasione dell'inserimento precedente). Poiché la dimensione della tabella è una potenza di 2, ciò significa che $i = 2^j + 1$ per qualche j. In tal caso, ogni chiave della vecchia tabella deve essere riassegnata a una delle liste di trabocco della nuova tabella usando la funzione hash associata alla tabella raddoppiata.[2] Se la funzione hash ha un comportamento ideale come abbiamo supposto, il costo atteso del reinserimento di ciascuna chiave nella tabella raddoppiata è $\mathcal{O}(1)$, in quanto il fattore di carico α della nuova

[1] La funzione hash cambia perché è cambiata la dimensione della tabella.

[2] Si noti che la funzione hash usata per riallocare le chiavi (rehashing) sarà in generale diversa da quella usata in precedenza, se non altro perché la capacità della tabella è raddoppiata.

tabella si mantiene minore di 1/2 durante il rehashing (si veda [1, Sezione 11.2]). Il costo complessivo atteso sarà quindi proporzionale al numero di elementi presenti nella tabella al momento dell'inserimento, quindi $\mathbb{E}[C_i] = \mathcal{O}(i) = \mathcal{O}(2^j + 1) = \mathcal{O}(2^j)$.
Il valore i non è una potenza di 2. Negli altri casi, il costo è quello dell'inserimento in una tabella con fattore di carico $\alpha \leqslant 1/2$. Se il comportamento della funzione hash è ideale come abbiamo supposto, il costo atteso di un inserimento è $\Theta(1 + \alpha) = \mathcal{O}(1)$ (si veda nuovamente [1, Sezione 11.2]).

A questo punto osserviamo che vi sono esattamente $\lfloor \log_2 n \rfloor$ inserimenti che danno luogo a rehashing, in particolare quelli corrispondente a $i = 2, 3, \ldots, 2^j + 1, \ldots 2^{\lfloor \log_2 n \rfloor}$. Il costo atteso di ciascuno degli altri $n - \lfloor \log_2 n \rfloor$ inserimenti è $\mathcal{O}(1)$ come abbiamo visto e il loro costo atteso complessivo è quindi $\mathcal{O}(n)$. Il costo complessivo atteso degli n inserimenti è quindi:

$$\mathbb{E}[C] = \mathcal{O}(n) + \mathcal{O}(\sum_{j=1}^{\lfloor \log_2 n \rfloor} 2^j).$$

D'altra parte abbiamo:

$$\sum_{j=1}^{\lfloor \log_2 n \rfloor} 2^j < \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j = 2^{\lfloor \log_2 n \rfloor + 1} - 1 < 2^{\log_2 n + 1} - 1 = 2n - 1 = \mathcal{O}(n).$$

Complessivamente abbiamo quindi $\mathbb{E}[C] = \mathcal{O}(n)$ e $\mathbb{E}[C]/n = \mathcal{O}(1)$.

Heap incredibili

Il professor Knowitall sostiene di aver concepito una nuova versione degli heap (la chiameremo K-Heap) che consente le seguenti operazioni su chiavi appartenenti a un universo perfettamente ordinato. 1) INSERT(x): inserisce la chiave x nella struttura dati; 2) EXTRACTMIN(): estrae e restituisce la chiave minima presente nell'heap modificato. Il Prof. Knowitall sostiene inoltre che le operazioni descritte sopra sono basate *esclusivamente* su confronti tra le chiavi e hanno costo $\mathcal{O}(1)$ nel caso peggiore.

Quesito. Dimostrare in modo rigoroso che il Prof. Knowitall ha preso un abbaglio.

Risposta. Se il Prof. Knowitall avesse ragione allora potremmo usare la sua struttura dati per ordinare un array in tempo lineare come segue:

Input: Array $\mathbf{A}[1 \ldots k]$ disordinato
Output: Array ordinato contenente le chiavi di $\mathbf{A}$

$KH = \emptyset$ `// Inizializza K-Heap vuoto`
for $i = 1$ *to* k **do**
  INSERT(KH, $\mathbf{A}[i]$)
end
$L = \emptyset$ `// Inizializza lista ordinata da restituire`
while $KH.heapsize > 0$ **do**
  x =EXTRACTMIN(KH)
  $L.add(elem)$
end
return L

Algorithm 1: Algoritmo K-Sort.

Sia il ciclo for che il ciclo while vengono eseguiti esattamente n volte ciascuno. Se il Prof. Knowitall avesse ragione, il costo di ciascuna iterazione del ciclo for (e del ciclo while) sarebbe $\mathcal{O}(1)$ nel caso peggiore, quindi il costo dell'algoritmo **K-sort** sarebbe $\mathcal{O}(n)$ nel caso peggiore. Avremmo quindi un algoritmo basato su confronti che ordina un array in tempo lineare nel caso peggiore. Ciò contraddice il noto limite $\Omega(n \log n)$ al costo dell'ordinamento basato su confronti (si veda [1, Sezione 8.1]). Il Prof. Knowitall si è evidentemente sbagliato.

Cammini minimi

Si consideri un grafo non orientato $G = (V, E)$, non necessariamente connesso, in cui tutti gli archi hanno lo stesso peso positivo $\alpha > 0$.

Quesito. 1) Scrivere lo pseudo-codice di un algoritmo efficiente che, dato in input il grafo G di cui sopra e due nodi u e v, calcola la lunghezza del cammino minimo che collega u e v. Se i nodi non sono connessi, la distanza è infinita. Non è consentito usare variabili globali. 2) Analizzare il costo computazionale dell'algoritmo proposto al punto 1). **Nota.** Se come subroutine si usano algoritmi noti visti a lezioni, non è necessario scriverne il codice ma soltanto dichiarare esplicitamente quale algoritmo si usa, identificare chiaramente il nome della relativa subroutine e gli argomenti di quest'ultima.

Risposta. Ovviamente sarebbe possibile usare l'algoritmo di Dijkstra o qualunque altro algoritmo per il calcolo di cammini minimi in grafi pesati. Tale scelta è tuttavia inutilmente costosa. Essendo infatti i pesi uguali, il problema corrisponde a quello di trovare il cammino che connette u e v con il minor numero possibile di archi. La soluzione più efficiente per tale caso è la visita in ampiezza, che ha costo $\mathcal{O}(n + m)$ se n e m sono rispettivamente il numero di nodi e archi di G (si veda [1, Sezione 20.2]).

Visita in profondità

Si consideri il grafo orientato in Figura 1. Si consideri la DFS su tale grafo, supponendo che essa esamini i vertici in ordine alfabetico e che ogni lista di adiacenza sia ordinata alfabeticamente.

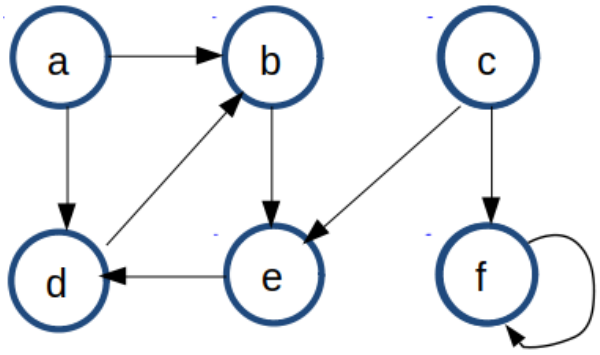

Figure 1: Grafo diretto.

Quesito. 1) Illustrare la struttura a parentesi della visita in profondità del grafo in figura; 2) indicare la classificazione di ciascun arco (si usi T per denotare gli archi d'albero B per gli archi all'indietro, F per gli archi in avanti e C per gli archi trasversali).

Risposta. La risposta a questo quesito nozionistico si trova in [1, Sezione 20.3]. Per quanto riguarda la struttura parentetica della visita abbiamo `(a (b (e (d d) e) b) a) (c (f f) c)`. Per la classificazione degli archi si faccia riferimento alla Figura 20.4 del libro di testo (e il relativo commento), che rappresenta lo stesso grafo e con lo stesso ordinamento alfabetico dei vertici.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduzione agli algoritmi e strutture dati 4/ed.* 2023.