Università degli Studi di Firenze

Tabelle hash - Analisi risoluzione collisioni con concatenamento e indirizzamento aperto

Mattia Pardeo

22/09/2022

Contents

Chapter 1

Introduzione

1.1 Descrizione del problema

In questo esercizio vogliamo analizzare come si comportano le tabelle hash che utilizzano come risoluzione delle collisioni il concatenamento e l'indirizzamento aperto, in particolare vogliamo analizzarne il comportamento al crescere del fattore di caricamento $\alpha = \frac{n}{m}$ dove n è il numero totale di elementi presenti nella tabella e m il numero totale di celle.

1.2 Teoria tabelle hash

Le tabelle hash consistono in delle implementazioni efficienti dei dizionari, ovvero una lista di valori ognuno identificato da una chiave che può essere ad esempio un numero od una stringa ed implementa le operazioni di inserimento, ricerca e cancellazione. In genere l'universo delle chiavi è molto più grande rispetto all'insieme delle chiavi effettivamente utilizzate, non possiamo quindi permetterci ad esempio di usare un array standard (tabella ad indirizzamento diretto) in quanto avremmo bisogno di crearlo tanto grande quanto il numero di chiavi nell'universo delle chiavi. Quello che faremo invece sarà utilizzare un numero ridotto di slot in cui salvare gli elementi ed usare una funzione, chiamata **hash**, che permetta di mappare le chiavi che ci arrivano in ingresso in uno slot della tabella. Il problema di questa funzione è che non è iniettiva, pertanto ci saranno delle chiavi che avranno come risultato lo stesso valore hash, ovvero identificheranno lo stesso slot della tabella, in tal caso si viene a creare una **collisione** in quanto entrambe le chiavi vogliono salvare il proprio valore nello stesso slot.

1.2.1 Funzione hash

Come detto poco sopra, la funzione hash è una funzione che mappa le chiavi in una delle celle della tabella e non è una funzione iniettiva, in particolare non può mai esserlo in quanto la dimensione dell'universo delle chiavi è molto più grande del numero di celle di una tabella hash, quindi inevitabilmente alcune chiavi avranno lo stesso valore di ritorno della funzione, detto anche **valore hash**. Una cosa importante che richiediamo ad una funzione hash è che sia **uniforme e semplice**, ovvero ogni elemento ha la stessa probabilità di andare in ogni cella indipendentemente da dove vanno gli altri elementi, cosa in realtà nella realtà non possibile, questo perché non conosciamo la distribuzione di probabilità da cui sono estratte e le chiavi, potrebbero quindi non essere indipendenti. Esistono comunque dei buoni metodi come il **metodo delle divisioni** (utilizzato anche in questa analisi per il concatenamento e funzione ausiliaria nell'hash dell'indirizzamento aperto).

Metodo delle divisioni

Questo metodo consiste semplicemente in una operazione di modulo tra la chiave e il numero di celle: data k una chiave allora la funzione hash è definita come $h(k) = k \bmod m$. Particolare attenzione è da porre sulla scelta del valore di m, in particolare lo vogliamo primo e non troppo vicino ad una potenza di 2, quest'ultima perché se ad esempio prendiamo $m = 2^p$ avremo che il modulo prende in considerazione sono i p bit meno significativi della chiave, invece vogliamo che sia dipendente da tutti i bit della chiave e non solo da parte di essa. Ha un tempo di esecuzione di O(1).

1.2.2 Risoluzione con concatenamento

In questo tipo di risoluzione utilizziamo delle liste singolarmente o doppiamente concatenate situate all'interno di ogni cella della tabella. Durante quindi un **inserimento** abbiamo che l'elemento verrà posto in testa alla lista di una certa cella identificata dal valore hash della chiave risolvendo in modo semplice e veloce sia l'inserimento che la collisione con un tempo di esecuzione O(1). Per la **ricerca** possiamo dimostrare, nell'ipotesi di avere una funzione hash uniforme semplice, essere dipendente dal fattore di carico e che avremo un tempo atteso di $\Theta(1 + \alpha)$ sia per la ricerca con successo che senza successo. Per l'**eliminazione** avremo un tempo O(1) in caso in cui la lista sia doppiamente concatenata, questo perché passiamo come argomento non la chiave ma il nodo direttamente, e un tempo uguale alla ricerca in caso sia singolarmente collegata, in quanto dobbiamo andare a cercare il nodo

immediatamente precedente a quello da eliminare.

1.2.3 Risoluzione con indirizzamento aperto

In questa risoluzione quello che facciamo è andare a salvare gli elementi direttamente nella tabella hash e non in una struttura esterna come nel caso precedente. Il fatto di non dover salvare dei puntatori in memoria ci permette di sfruttare la memoria risparmiata per aumentare di molto il numero delle celle della tabella, permettendo così un tempo medio di ricerca e inserimento minore e un numero minore di collisioni. L'avere più celle inoltre ci è anche utile in quanto, a differenza del caso precedente, la tabella può essere riempita e renderci impossibile quindi salvare ulteriori elementi. Le operazioni di inserimento e ricerca vengono svolti tramite una **esplorazione** della tabella, ad esempio alla ricerca di una cella vuota, la quale è identificata da un valore nullo. Per fare questa esplorazione però non partiamo ogni volta dall'inizio della tabella ma partiamo da una posizione dipendente dalla chiave, così da poter terminare in un tempo minore di Θ(n). Avremo bisogno quindi che la funzione hash ci vada a creare una sequenza di esplorazione in modo tale che si possano esplorare tutte le celle, in particolare abbiamo bisogno che sia una funzione hash **uniforme**, ovvero deve garantire che ogni chiave possa creare tutte le m! permutazioni della sequenza 0, ..., m-1 con la stessa probabilità. Per l'**inserimento** avremo quindi che andremo ad esplorare la tabella, tramite la sequenza generata, fintanto che non troviamo una cella vuota in cui salvare l'elemento oppure non abbiamo esplorato l'intera tabella. Facciamo una cosa simile per la **ricerca** in cui stavolta ci fermiamo se troviamo l'elemento cercato oppure se terminiamo la tabella o troviamo un elemento nullo. La **cancellazione** è più complessa rispetto al concatenamento, questo perché quando eliminiamo non possiamo semplicemente mettere il valore nullo nella cella ma dovremo metterci un carattere speciale, altrimenti, per come è fatta, rischiamo di far interrompere la ricerca prima del dovuto, inoltre dovremo modificare l'algoritmo di inserimento in modo che possa inserire anche quando trova il carattere speciale. La cancellazione però ci rimuove la dipendenza dell'inserimento e della ricerca dal fattore di carica. Possiamo verificare che per l'inserimento e la ricerca senza successo abbiamo un numero atteso di esplorazioni al più di $\frac{1}{1-\alpha}$, per la ricerca con successo invece si ha $\frac{1}{\alpha}ln(\frac{1}{1-\alpha})$.

Esplorazione lineare

Il tipo di esplorazione usata in questa analisi ha una funzione hash del tipo *h(k, i) = (h'(k) + i) mod m* dove i è un indice che va da 0 a m-1 e *h'(k) =*

$k \ mod \ m$ è una funzione hash **ausiliaria**. Il problema di questa esplorazione è il **clustering primario**, ovvero si vanno a creare una lunga sequenza di celle occupate che vanno inevitabilmente ad aumentare il tempo medio di inserimento e ricerca.

Chapter 2

Documentazione classi

Sono state usate varie librerie python per poter disegnare grafici e tabelle, matematiche e di misura del tempo. In particolare abbiamo **matplotlib** usata tramite l'interfaccia **pyplot** per i grafici, **plotly** tramite **graph_objects** per le tabelle, **random** per generare numeri casuali, **statistics** per eseguire la media di una lista e **timeit** tramite **default_timer** per misurare i tempi dei test.

2.1 Node e LinkedList

La classe Node è una classe che funge da nodo per una lista concatenata, come attributi avrà quindi il dato salvato e il puntatore al nodo successivo, come metodi possiede solo getter e settere degli attributi. LinkedList implementa una lista singolarmente concatenata al cui interno non utilizza direttamente la classe Node (infatti non va a creare nodi) ma nei metodi si presuppone che vengano inviati dei nodi come parametri, infatti troviamo al suo interno chiamate a metodi di tale classe. I metodi implementati sono aggiunta, ricerca ed eliminazione di un nodo, conteggio dimensione, verifica se vuota e stampa tutti gli elementi, l'unico attributo è un puntatore alla testa della lista.

2.2 Chained

Questa classe implementa la tabella hash che utilizza come metodo di risoluzione delle collisioni il concatenamento. Possiede come attributi il numero di collisioni totali, il numero di celle e una lista di celle inizialmente inizializzate con delle liste vuote. Abbiamo il metodo che ritorna il numero di collisioni, la funzione hash che usa il metodo delle divisioni, le funzioni di inserimento, ricerca e cancellazione che vanno a posizionarsi in una cella specifica tramite

il valore hash e va a richiamare i metodi definiti in LinkeList, come ultimo un metodo per stampare tutta la tabella.

2.3 OpenAddress

Questa classe implementa la tabella hash che utilizza come metodo di risoluzione delle collisioni l'indirizzamento aperto. Come attributi possiede il numero di esplorazioni, il numero di collisioni totali, il numero di celle, una lista di celle inizializzata coi valori nulli. Possiede come metodi il ritorno delle collisioni totali, la funzione hash per l'esplorazione lineare, l'inserimento, la ricerca, la cancellazione e la stampa di tutti i valori.

Chapter 3

Esperimenti

Gli esperimenti effettuati riguardano solo l'inserimento e la ricerca e non la cancellazione, in quanto nell'implementazione con concatenamento possiamo avere un tempo O(1) o ridursi allo stesso tempo della ricerca e nell'indirizzamento aperto per mantenere la dipendenza col fattore di caricamento. All'inizio del programma possiamo andare a cambiare i valori di num_test, per decidere quante iterazioni devono essere fatte dei vari test, e di num_cells per decidere il valore di m. Inoltre i risultati finali rispecchiano una media effettuata su tutti i dati raccolti da ogni iterazione. Gli esperimenti sono stati condotti su un pc dotato di processore a 6 core e 12 thread con frequenza base di 3.2GHz e 3.6GHz in boost, la memoria RAM a disposizione è di 16GB ed il sistema operativo è Windows 10. La versione di Python utilizzata è la 3.10.

3.1 Test inserimento

All'inizio del test e prima delle iterazioni vengono create 5 liste: due saranno liste di liste che conterranno i tempi misurati in ogni iterazione rispettivamente per la tecnica con concatenamento e indirizzamento aperto, due liste di liste che conterranno le collisioni di ogni iterazione e un'ultima lista contenente valori crescenti di 1 del fattore di caricamento che fungerà da asse x nel grafico che andremo a disegnare. Nelle varie iterazioni andremo a creare sempre delle nuove tabelle hash, così da ripartire sempre da zero, ed a generare una lista, di dimensione maggiore delle celle in modo da testare anche $\alpha > 1$, di interi casuali da andare ad inserire all'interno delle tabelle. Dopo aver generato i valori andremo ad iterare sulla lista di valori da inserire e ne inseriremo uno ad uno nelle tabelle e ad ogni inserimento andremo a misurare quanto tempo hanno impiegato a farlo, così da avere il tempo testato per ogni fattore di caricamento possibile. Dopo aver concluso le iterazioni

verrà fatta la media dei tempi e delle collisioni e verrà richiamata la funzione che andrà a disegnare e il grafico dei tempi di inserimento e collisioni insieme e le tabelle con tutti i valori di tempi in dettaglio per ogni valore del fattore di caricamento espressi tutti in notazione esponenziale.

3.2 Test ricerca con successo

Il test è come quello di inserimento, quello che cambia è che immediatamente dopo aver inserito l'elemento andiamo a fare la ricerca proprio di quest'ultimo elemento inserito e ne misuriamo il tempo.

3.3 Test ricerca senza successo

Uguale al test di ricerca, stavolta però creiamo una ulteriore lista di elementi che non sono all'interno della lista e lo facciamo in funzione dei valori da inserire.

Chapter 4

Risultati sperimentali

Gli esperimenti sono stati condotti facendo 1000 iterazioni per ogni test e usando 503 celle, le righe delle tabelle è di 513 per le collisioni, 603 per l'inserimento, 504 per la ricerca con e senza successo. I grafici presenti sono stati tracciati utilizzando i dati contenuti nelle tabelle presenti nella rispettiva cartella.

4.1 Inserimento

In questo esperimento ho effettuato dei test che andassero oltre il valore di 1 nel fattore di caricamento per testare anche il comportamento della risoluzione con concatenamento in tal caso, quindi abbiamo un numero di inserimenti dato dal numero di celle più 100. Possiamo vedere dal grafico in figura 4.1 come l'indirizzamento aperto è più performante fintanto che si rimane all'incirca al di sotto del 60% del fattore di caricamento, da quel momento in poi i tempi di inserimento iniziano a peggiorare. Vediamo anche come, a parte per alcuni casi sporadici, il caso con concatenamento rimanga costante anche quando si supera il valore 1 del fattore di caricamento, questo è dovuto al fatto che l'inserimento avviene in testa alla lista e non si deve andare a cercare una cella vuota.

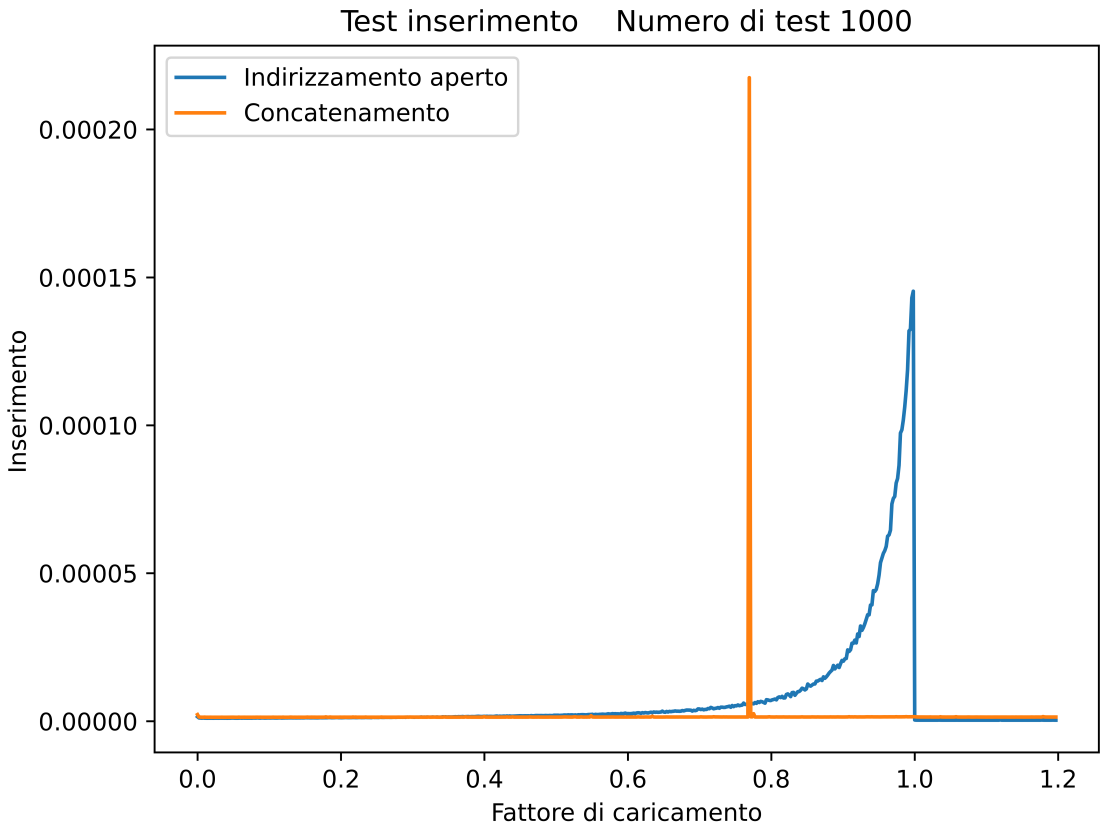

Figure 4.1: Confronto tra indirizzamento aperto e concatenamento nel test di inserimento. Sulla y abbiamo i tempi espressi in secondi.

Per quanto riguarda invece le collisioni vediamo, dalla figura 4.2, come in media sono maggiori per l'indirizzamento aperto rispetto che alla risoluzione con concatenamento.

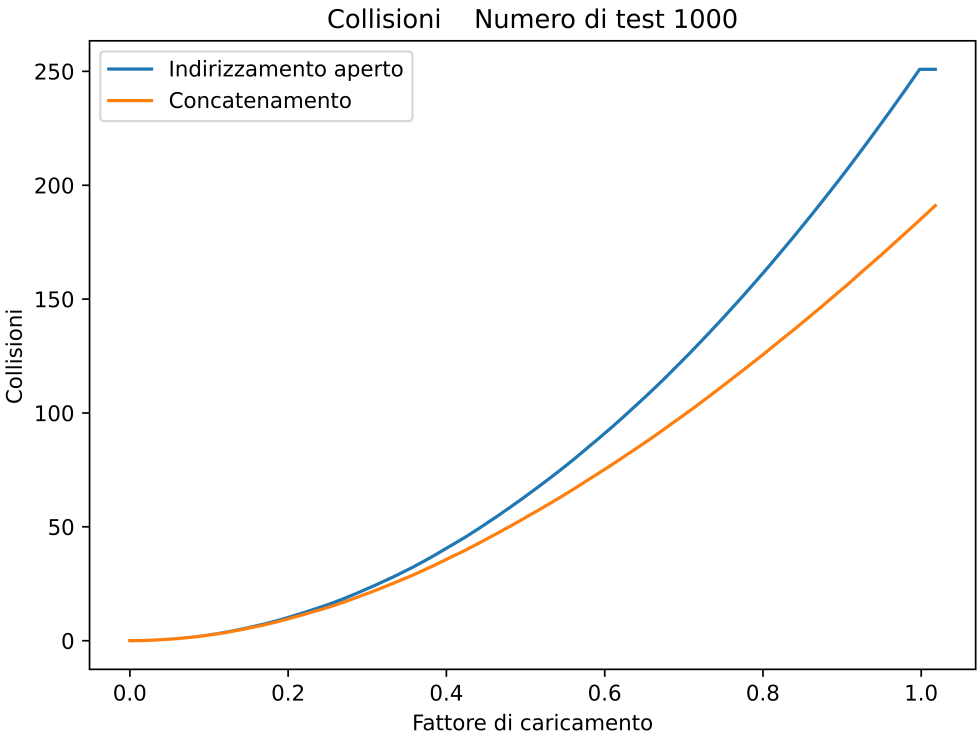

Figure 4.2: Confronto collisioni tra indirizzamento aperto e concatenamento

Se per l'indirizzamento aperto osserviamo quali sono i valori di esplorazioni attese rispetto a quelle effettive, figura 4.3, vediamo che non esattamente rimangono sempre minori, cosa che ci aspetteremmo invece in media, cosa che succede anche ai tempi essendo questi e le esplorazioni collegate.

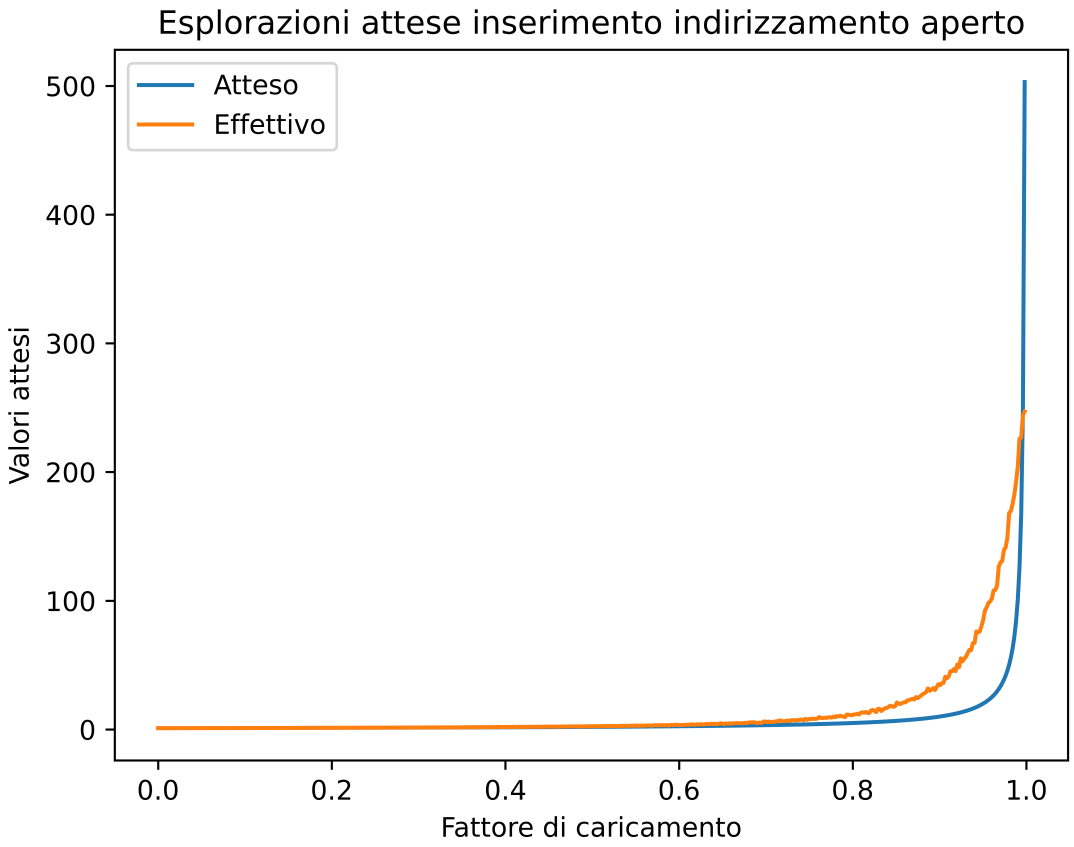

Figure 4.3: Confronto tra esplorazioni attese ed effettive per l'indirizzamento aperto.

4.2 Ricerca senza successo

Osservando il grafico, figura 4.4, vediamo come l'indirizzamento aperto stavolta non performa bene quasi fin da subito, da circa il 50% di riempimento i tempi si attestano al di sopra del concatenamento.

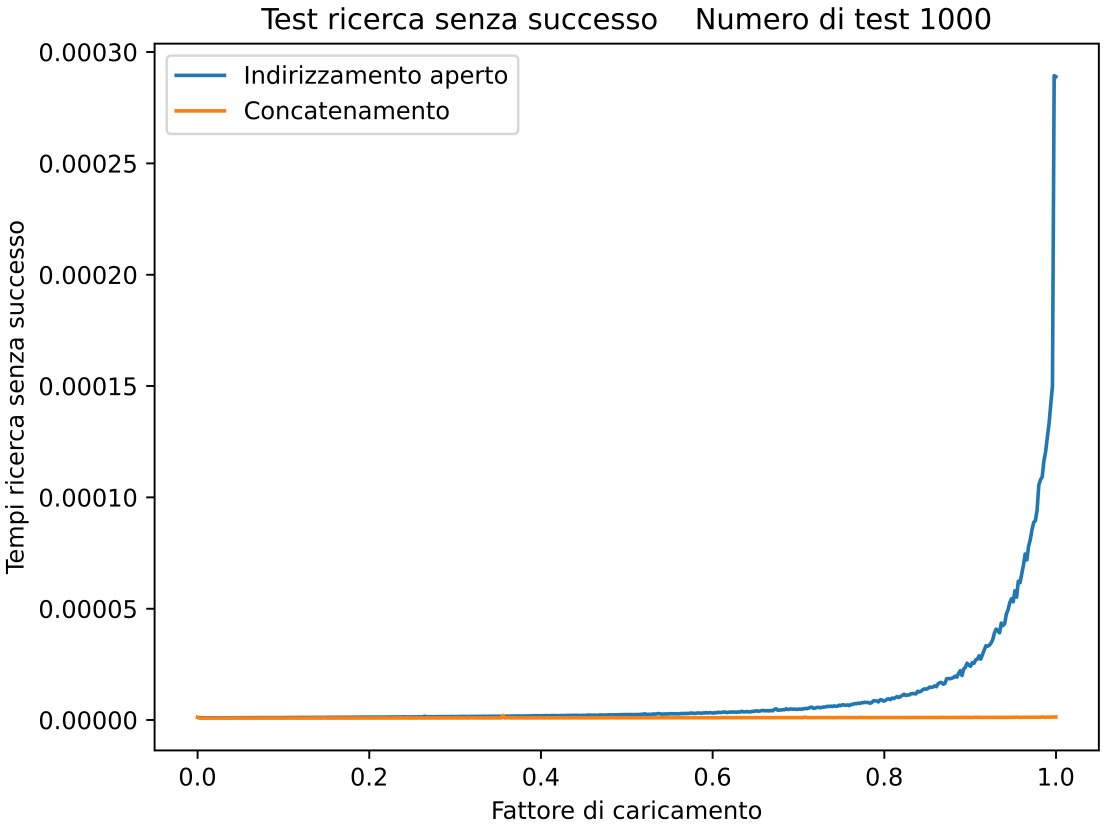

Figure 4.4: Confronto tra indirizzamento aperto e concatenamento nel test di ricerca senza successo. Sulla y abbiamo i tempi espressi in secondi.

Facendo riferimento di nuovo alla figura 4.3 vediamo che anche in questo caso l'andamento è molto simile all'andamento dei valori attesi.

4.3 Ricerca con successo

Osservando il grafico in figura 4.5 notiamo come anche in questo caso, come per l'inserimento, i tempi dell'indirizzamento aperto rimangono minori uguali a quelli del concatenamento fintanto che il fattore di caricamento si trova attorno al 50/60%, a quel punto l'indirizzamento aperto inizia a peggiorare.

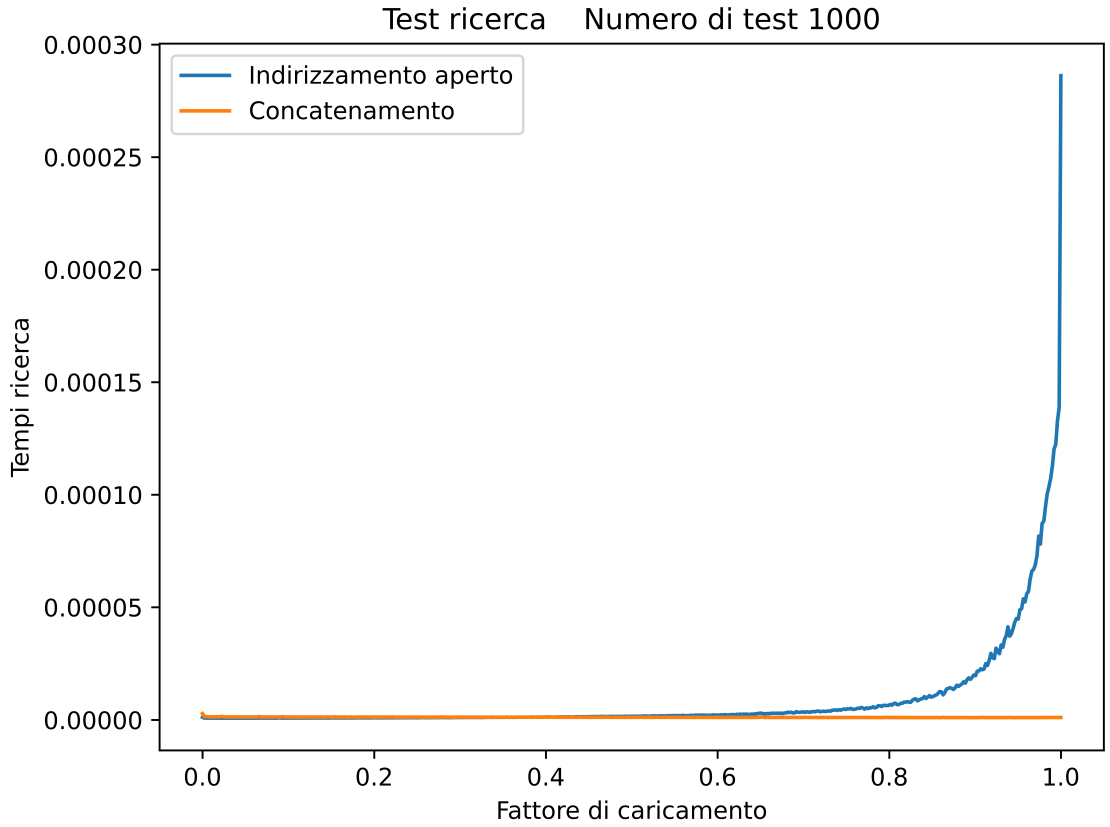

Figure 4.5: Confronto tra indirizzamento aperto e concatenamento nel test di ricerca con successo. Sulla y abbiamo i tempi espressi in secondi.

Chapter 5

Conclusioni

Dai dati osservati possiamo notare come effettivamente l'indirizzamento aperto abbia delle prestazioni migliori in termini di ricerca e inserimento fintanto che il fattore di caricamento rimane sotto una certa soglia, nel nostro caso vediamo che è intorno al 60%, dalla quale poi i tempi iniziano a peggiorare e conviene usare il concatenamento, il quale in generale rimane sempre costante qualunque sia il fattore di caricamento. Da qui capiamo che se il fattore di caricamento tende molto velocemente ad 1 può convenire usare il concatenamento, oppure adottare dei controlli tali che aumentino la dimensione della tabella nel caso dell'indirizzamento aperto così da mantenere il fattore di caricamento al di sotto di una certa soglia. Inoltre come sappiamo dalla teoria in caso si abbia bisogno di frequenti cancellazioni non è conveniente adottare l'indirizzamento aperto in quanto perde la dipendenza dal fattore di caricamento.