POLITECNICO DI MILANO, DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA

Progetto di Ingegneria Informatica : Relazione Conclusiva

Nicolò Vendramin

8 luglio 2016

1 INTRODUZIONE

Con la presente relazione mi presto a concludere l'attività di progetto svolta, con il supporto e la supervisione del Professor Alessandro Barenghi, nell'ambito del corso da 5 CFU "Progetto di Ingegneria Informatica".

Il lavoro che è stato svolto si colloca nel contesto più ampio dell'attività di ricerca che ha prodotto e tuttora si occupa dello sviluppo e del mantenimento di *Papageno*[1]. Papageno è uno strumento che è in grado di generare in modo automatico analizzatori sintattici, sfruttando la proprietà di parsabilità locale dei linguaggi cosiddetti *operator precedence*, ossia a precedenza di operatori. Le proprietà di questi ultimi consentono a Papageno di realizzare dei parser che siano in grado di sfruttare il parallelismo messo a disposizione dalla presenza di processori multicore sulla maggioranza delle macchine di uso comune al giorno d'oggi e, quindi, di effettuare l'analisi sintattica in tempi notevolmente più brevi rispetto a quelli sequenziali generati dai *tool* di uso comune (e.g. GNU Bison).
In modo più specifico il compito di cui sono stato incaricato è stato quello di realizzare uno strumento, generatore di tabelle di hash perfetto, da impiegare all'interno del tool Papageno per produrre le strutture dati (e le relative interfacce di accesso) da utilizzare in fase di parsing per effettuare il matching tra la *right hand side (rhs)* in analisi e la corrispondente *left hand side (lhs)* della grammatica per cui il parser è stato generato.

[1] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. Parallel parsing of operator precedence grammars, Inf. Process. Lett., 113(7):245–249, 2013. [Pagina del progetto]

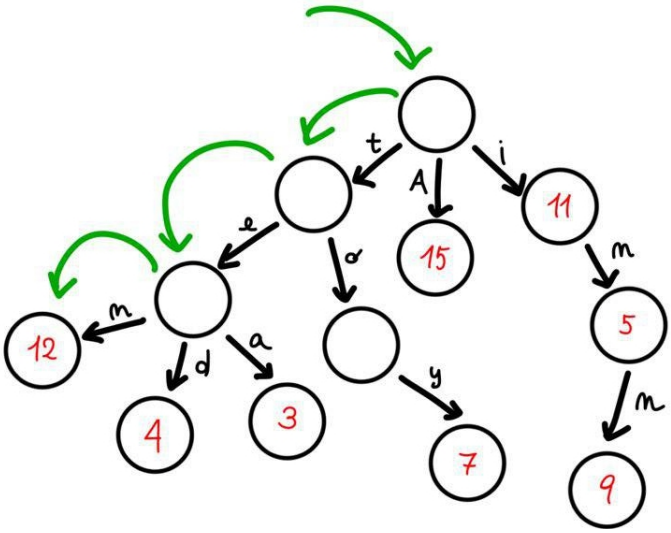

Figura 1.1: Trie: albero a prefissi

Precedentemente per svolgere questo compito veniva utilizzata una struttura dati denominata *Trie*: nota in italiano come albero a prefissi, una struttura di questo tipo è rappresentata da un albero le cui foglie sono associate ad un dato. Per arrivare ad un dato è necessario scandire passo passo la stringa, o l'insieme di interi, che ne descrivono il percorso all'interno dell'albero a prefissi. Ad esempio con riferimento alla Figura 1.1 si ha che per scoprire che il dato associato alla chiave *'ten'* è 12, è necessario accedere a 3 differenti nodi della struttura ad albero. Il numero di accessi in memoria (proporzionale al tempo di esecuzione e, quindi, indice della complessità temporale) è dunque pari alla lunghezza della chiave o, nel caso specifico di Papageno, alla lunghezza del lato destro della riduzione in analisi. La complessità temporale della struttura realizzata con l'albero a prefissi è $\Theta(m)$ dove m è la lunghezza della right hand side, e, considerato che tale lunghezza è finita e limitata superiormente per una data grammatica, tale complessità si può considerare costante.

Al fine di ridurre al minimo il numero di accessi in memoria, poichè particolarmente pesanti dal punto di vista del costo temporale, è sorta la necessità di andare a sostituire il meccanismo dell'albero a prefissi con una struttura differente, la quale garantisse di poter raggiungere il dato associato alla chiave (nel caso specifico la *lhs* associata ad una certa *rhs*) con un numero di accessi oltre che costante anche indipendente dalla dimensione della chiave ed ovviamente il più possibile limitato.
Per le proprietà che verranno approfondite nelle sezioni successive la scelta per lo svolgimento di questa funzionalità è ricaduta sulla struttura dati detta *Perfect hash table*.

2 TECNICHE DI HASHING

In questa sezione mi presto ad introdurre i concetti teorici e le nozioni fondamentali che sono state necessarie durante lo svolgimento del progetto, a partire da una generica analisi del principio dell'hashing per poi concentrare l'attenzione sul tema più specifico dell'hashing perfetto nel secondo paragrafo.

2.1 HASHING

Con il termine della lingua inglese *to hash* si intende l'atto di sminuzzare e compattare qualcosa. In ambito informatico e matematico, questo termine è stato utilizzato per indicare un particolare tipo di funzione in grado di prendere in input una quantità di dati arbitrariamente grande, per restituirne una versione compatta di dimensione massima fissata. Le applicazioni del concetto di *hashing* sono molteplici; tra le più note citiamo quella nell'ambito della tecnologia di firma digitale e quella, di specifico interesse per questa trattazione, relativa all'indirizzamento per le strutture dati denominate, per derivazione, tabelle di hash (in inglese *hashtable*).

Una tabella di hash è una struttura che viene utilizzata per memorizzare dati, e per la quale il meccanismo di inserimento e ricerca sono basati sull'indirizzamento compiuto a mezzo di un algoritmo di hash. I costituenti fondamentali di un algoritmo di questo tipo sono due:

- Funzione di hash,
- Politica di Gestione delle Collisioni;

La funzione di Hash da impiegare nel contesto di un algoritmo di hashing deve chiaramente avere la caratteristica di restituire sempre indici appartenenti alla tabella. Ossia in termini formali: data una funzione di hash h() applicabile a chiavi appartenenti a un insieme $\mathbf{K}$, per l'indirizzamento in una tabella $\mathbf{T_h}$ di dimensione $|\mathbf{T_h}|$=n, deve valere che:

$$h : \mathbf{K} \mapsto [0, n-1] \tag{2.1}$$

oppure equivalentemente:

$$\forall k \in \mathbf{K}, 0 \leq h(k) < n \tag{2.2}$$

Oltre a ciò una buona funzione di hashing dovrebbe soddisfare la proprietà di hashing uniforme semplice, ossia garantire che la probabilità che una chiave venga associata dalla funzione ad una certa cella è indipendente dalla particolare cella e dal risultato della funzione sulle altre chiavi del dominio.[2] In formule :

$$\begin{aligned} &\forall k_1, k_2 \in \mathbf{K}, \\ &\forall t, s \in [0, n-1] \\ &p(h(k_1) = t) = p(h(k_2) = s) \quad \wedge \quad p(h(k_1) = t | h(k_2) = s) = p(h(k_1) = t) \end{aligned} \tag{2.3}$$

[2] Di fatto, fatta eccezione per alcuni casi particolari, non è possibile soddisfare questa ipotesi nel senso vero e proprio del termine; ad esempio in molti contesti non c'è la possibilità di avere informazione sulla distribuzione statistica con cui le chiavi vengono estratte, e talvolta viene violato, per ragioni legate al contesto in cui è inserita la hashtable, il principio di interindipendenza delle singole estrazioni.

Una possibile e semplice implementazione di funzione di hash che rispetti le proprietà che sono state enunciate fino a questo momento ad esempio potrebbe leggere il parametro in input come un intero e ritornare il resto della divisione intera dell'input per la dimensione totale della tabella. Nell'algoritmo di hashing la funzione h ha quindi il compito di indicare in quale cella della tabella che rappresenta i dati in memoria debba essere inserito o ricercato il dato associato alla chiave (che può coincidere con il dato stesso). Appare evidente che nel caso, molto frequente, in cui la cardinalità dell'insieme **K** di chiavi sia maggiore della lunghezza n della tabella di hash, la funzione h risulterebbe essere non iniettiva e quindi si verificherebbe che

$$\exists k_1, k_2 \in \mathbf{K} : k_1 \neq k_2 \wedge h(k_1) = h(k_2) \tag{2.4}$$

A causa di questo fatto è generalmente possibile che al momento dell'inseririmento di un dato nella tabella, la cella, corrispondente all'indice ottenuto applicando la funzione di hash alla chiave relativa al dato stesso, sia già occupata. Tale fenomeno è definito *collision*, collisione, ed è proprio questo il secondo aspetto che un buon algoritmo di hashing deve tenere in considerazione.
Per gestire il fenomeno delle collissioni gli algoritmi di hashing possono impiegare politiche differenti. Le due più comunemente utilizzate sono

- Indirizzamento aperto
- Indirizzamento chiuso

Nel primo dei due casi la funzione di hash non fornisce più semplicemente l'indice della tabella associato alla chiave presa in input, ma fornisce anche l'intera *sequenza di ispezione* con cui esaminare la tabella fino a trovare l'elemento cercato o trovare una cella libera in cui inserire l'elemento, in base a che si stia effettuando rispettivamente un *lookup* o una *insert*. Un esempio banale è il caso in cui la funzione di hash oltre al parametro *key* riceva in ingresso anche un indice di iterazione, e produca come output, al variare del secondo dato in input, una sequenza di ispezione. A titolo esemplificativo si consideri il caso di una funzione che prenda in ingresso due parametri *key* e *iteration* e ritorni il resto della divisione intera tra il valore *key*, aumentato di una quantità pari al valore *iteration* moltiplicato per una costante, e la dimensione della tabella di hash. Nel caso di indirizzamento chiuso invece nella tabella di hash non vengono più registrati i dati stessi, ma il puntatore ad una lista concatenata che contiene tutti gli elementi associati dalla *hash function* a quella particolare cella.

Per quanto riguarda gli aspetti legati alla complessità in ricerca le tavole di hash presentano un $T_n = \Theta(1)$ nel caso ottimo, ma negli altri casi presentano un valore dipendente dal *load factor*, fattore di carico, $\alpha = \frac{m}{n}$, dove m è il numero di celle della tabella $\mathbf{T_h}$ già occupate al momento in cui si effettua il lookup. Il caso pessimo è quello della ricerca per mezzo di una chiave k in una tabella piena in cui non è presente l'elemento desiderato. In questo caso il lookup in una tabella gestita con indirizzamento aperto (risp. concatenamento) renderebbe necessario percorrere l'intera sequenza di ispezione (risp. la lista concatenata associata a h(k), che nel sottocaso pessimo include tutti gli elementi inseriti), quindi con complessità $\Theta(n)$, per poter affermare che l'elemento non appartiene all'insieme di dati memorizzati.

2.2 PERFECT HASHING

Con il termine *Perfect Hashing* viene designata una particolare tipologia di hashing in cui la funzione che compone l'algoritmo di hash è strutturata in modo tale da determinare l'assenza di collisioni, e che quindi, oltre a rendere non necessaria l'implementazione di una politica per la gestione di queste ultime, è caratterizzata da complessità temporali pari a $\Theta(1)$ per tutte le operazioni fondamentali. Ovviamente la maggiore efficienza garantita da tecniche di questo tipo è resa possibile dalla restrizione del campo di applicazione delle stesse. Per poter ottenere un algoritmo di hash perfetto è infatti necessario conoscere in fase di scrittura (o generazione automatica) della funzione, non solo l'insieme **K** di tutte le stringhe di chiave, ma anche il sottoinsieme **K'**$\subseteq$ **K** delle chiavi che devono poter essere inserite o ricercate nella tavola di hash utilizzata per memorizzare i dati. Volendo eliminare la possibilità che due chiavi differenti tra quelle del subset **K'** di **K** possano collidere, ossia produrre lo stesso hash, è necessario che valga la seguente proprietà:

$$\forall k_1, k_2 \in \mathbf{K'} \subseteq \mathbf{K} \quad k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2) \tag{2.5}$$

che corrisponde precisamente con la definizione di funzione iniettiva.
Il fatto che h sia una iniezione implica che la cardinalità del codominio della funzione è necessariamente almeno uguale alla cardinalità del suo dominio.
In formule questo significa che la dimensione di una tabella di hash da utilizzare in associazione ad una funzione di hash perfetto è almeno uguale al numero di chiavi che compongono l'insieme **K'**, ossia che saranno estratte, e quindi che $|\mathbf{T_n}| = n \geq |\mathbf{K'}|$. Sulla base di queste ultime affermazioni possiamo anche ricavare che la complessità spaziale di una tabella di hash perfetto appartiene in generale a $O(n)$. Nel caso particolare in cui la dimensione della *hashtable* sia precisamente uguale al numero di chiavi del set **K'**, e in cui quindi la funzione h risulta essere biunivoca, la struttura dati prende il nome di *MPHT*, acronimo per *minimal perfect hash table*. Una tabella di hash perfetto minima è, tra le tabelle di hash perfetto, quella a complessità spaziale minore.

Le Perfect Hash Table possono essere distinte sulla base di come provvedono a garantire l'iniettività del processo di hashing. Le due principali tecniche impiegate sono le seguenti:

- Hashing diretto
- Hashing a due livelli.

Nel primo dei due casi la struttura è molto simile a quella di una tavola di hash ad indirizzamento aperto, con l'unica differenza che la funzione di hash non prevede alcun tipo di tecnica per la gestione delle collisioni. In questo caso, quindi, è proprio la funzione h in se che deve essere in grado di garantire che a chiavi diverse corrispondano output differenti.
Nel secondo caso invece l'hashing viene effettuato in due step: in un primo momento la funzione di hash stessa, o un altra funzione h'() definita ad hoc, viene fatta eseguire sulla chiave per poi usare il valore di ritorno come indirizzo di accesso ad una tabella intermedia $\mathbf{D_h}$, detta tabella di disambiguazione, in cui sono salvati dei valori costanti che vengono calcolati durante la generazione della funzione. Successivamente la funzione di hash sulla base del

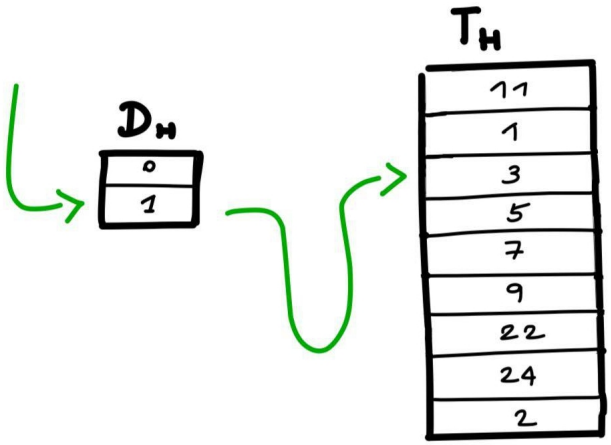

Figura 2.1: Hashing con tabella di disambiguazione

valore trovato nella tabella di disambiguazione e della chiave produce l'hash definitivo con cui accedere alla vera e propria tavola dove sono memorizzati i dati. Si noti che la dimensione della tabella di disambiguazione è generalmente molto più piccola di quella della tavola di hash, e che l'hashing che viene effettuato durante il primo step non è iniettivo. Infatti a garantire che il processo visto complessivamente sia una iniezione è il fatto che il valore i-esimo della tabella di disambiguazione è calcolato in modo tale che:

$$\begin{aligned} &\forall\, k_1, k_2 \in \mathbf{K'} \subseteq \mathbf{K} \\ &h'(k_1) = h'(k_2) = i \Rightarrow h(k_1, D_h[i]) \neq h(k_2, D_h[i]) \end{aligned} \tag{2.6}$$

Come anticipato precedentemente h ed h' possono anche essere la medesima funzione invocata con parametri diversi.Si faccia riferimento alla Figura 2.1 per avere una rappresentazione visuale semplificata di quanto descritto.

Il processo con cui una *PHT* (Perfect Hash Table) o una *MPHT* viene generata è solitamente basato sull'impiego di algoritmi in grande parte randomizzati. È infatti dimostrato che è possibile generare una funzione di hash perfetto (minima e non) in un tempo lineare al numero di chiavi che essa deve distinguere[3], sfruttando la strategia *by trial and error*. Il meccanismo di funzionamento fondamentale di questo tipo di algoritmi è quello di stabilire il prototipo di una funzione che sia dipendente da un certo numero di parametri (nel caso di in cui venga usata una tabella intermedia sono proprio i valori di quest'ultima) che poi sono fatti variare in modo casuale fino al momento in cui tutte le chiavi riescono ad essere correttamente collocate senza generare collisioni.

[3]Per approfondire si rimanda a ̀Hash, displace, and compress´ Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger. [Link alla pubblicazione.]

3 LA REALIZZAZIONE DEL GENERATORE

Come anticipato nella sezione introduttiva il lavoro di cui mi sono occupato per questo corso è stato quello di produrre un generatore automatico di tabelle di hash perfetto in grado di collocare correttamente e senza collisioni un insieme di valori naturali, le left hand side, utilizzando come chiavi le corrispondenti right hand side, rappresentate come un set di interi positivi. La specifica di partenza era quindi quella di elaborare un componente software in grado di prendere in input un certo insieme di coppie (rhs,lhs) con rhs $\in \mathbf{N^n}$ e lhs $\in \mathbf{N}$ e di produrre la tabella di hash in cui tutte le lhs fossero state collocate senza collisioni, rappresentata opportunamente in un file C in cui fosse presente anche la funzione di look-up per accedervi.

3.1 L'ALGORITMO DI HASH

L'implementazione che ho scelto per realizzare quanto richiesto prevede l'uso di una tabella di hash perfetto a due livelli, in cui la funzione di hash si basa sull'algoritmo non crittografico Fowler/Noll/Vo (FNV)[4] sia per l'hashing nella tabella intermedia che in quella finale.
L'algoritmo FNV di hashing non crittografico è pensato per l'utilizzo in ambito web; infatti esso garantisce una buona dispersione, e quindi un rate di collissioni molto basso, per chiavi tra loro molto simili come ad esempio URL, filenames, hostnames o testi, cosa che lo rende ampiamente utilizzato in applicazioni quali server DNS o generatori di hash-index per basi di dati. La scelta di questo algoritmo tra i tanti disponibili è stata circostanziata dal fatto che le right hand side di cui effettuare l'hashing sono costituite da insiemi di interi di piccole dimensioni e ciascuno dei valori di cui una singola chiave è costituita varia in un range molto limitato, cosa che rende le differenti rhs molto simili tra loro. Rispetto all'implementazione classica comunemente impiegata dell'algoritmo si è resa necessaria una personalizzazione per spostare il dominio di applicazione dalle stringhe agli interi. Di seguito si riporta lo pseudocodice che rappresenta il funzionamento dell'algoritmo di hash FNV:

```
Function fnvHash(key,iteration) {
    if (iteration == 0)
        hash = offsetbasis ;
    else
        hash = iteration ;
    foreach integer in key
        hash = ((hash * offsetbasis) ⊕ integer) & 0xffffffff;
    return hash;
}
```

[4] Si riportano due link di pagine in cui si trovano documentazione ed informazioni dettagliate sull'algoritmo di hashing non crittografico FNV. [Documentazione generale] e [IETF informational draft su FNV.]

Nell'implementazione dell'algoritmo di hashing a due livelli ho utilizzato l'algoritmo FNV per l'accesso ad entrambe le tabelle intermedia e finale; in particolare per l'hashing nella tabella di dismabiguazione il valore di iteration usato per invocare la funzione è 0, mentre, per la seconda fase di indirizzamento, il valore del parametro è scelto sulla base di quanto letto nella tabella intermedia.

3.2 L'ALGORITMO DI GENERAZIONE

Per quanto riguarda la generazione della tabella di hash perfetto ho utilizzato una procedura con il seguente funzionamento: inizialmente tutte le chiavi del set vengono hashate e collocate in buckets differenti sulla base del risultato ottenuto. Il numero di contenitori è pari alla dimensione della tabella di disambiguazione utilizzata. L'i-esimo bucket conterrà quindi, una volta terminata questa fase del processo, tutte le chiavi a cui corrisponde l'hash i.
A questo punto tutti i contenitori vengono ordinati sulla base del numero di chiavi che contengono, in ordine decrescente. Per ogni bucket, partendo dai contenitori più popolati fino ad arrivare a quelli con dimensione maggiore o uguale a due, tutte le chiavi vengono sottomesse alla funzione di hash con valori del parametro iteration crescenti fin quando non si trova il valore per cui vengono tutte collocate in slot liberi della tabella finale. Quest'ultimo viene salvato nella cella della tabella di disambiguazione corrispondente al bucket in analisi.
Una volta eseguito questo procedimento per tutti i contenitori con dimensione maggiore di uno vengono processati i rimanenti, semplicemente collocando la chiave che contengono nel primo slot libero della hashtable e segnando nella corrispondente casella della tabella di disambiguazione l'indice di tale slot cambiato di segno.
In questo modo al termine dell'esecuzione tutte le lhs si trovano in una cella della tabella finale e in ogni casella della tabella intermedia è inserito o un valore positivo corrispondente al parametro iteration da usare per il secondo step di hashing o un valore negativo che corrisponde all'opposto dell'indice della tabella finale a cui accedere.
Per quanto detto la funzione di lookUp può essere descritta dal breve codice riportato:

```
Function lookUp(key) {
    firstHash = disambiguationTable[hash(key,0)];
    if (firstHash <0 )
        return table[-firstHash];
    else
        return table[hash(key,firstHash)];
}
```

Al fine di rendere più chiara la procedura descritta sopra in modo discorsivo si allega uno stralcio di pseudocodice che rappresenta l'algoritmo di generazione.
Alcuni dettagli sono omessi e altri descritti con linguaggio naturale così da non appesantire inutilmente la lettura.

```
Function perfectHashGenerator(keySet) {
    foreach key in keySet
        append(buckets[hash(key,0)],key);
    reverseSort(buckets);
    foreach bucket in buckets
        if (size(bucket) == 1)
            break;
        while (!ok);
            iteration ++ ;
            ok = true;
            foreach key in bucket
                if (table[hash(key,iteration)] is not empty)
                    ok = false and break;
                else
                    table[hash(key,iteration) = data(key);
            if (!ok)
                restores table status;
            else
                disambiguationTable[hash(key,0)] = iteration;
    foreach remaining bucket in buckets
        for int i in size(table)
            if (table[i] is empty)
                table[i] = data(bucket[0]);
                disambiguationTable[hash(bucket[0],0] = -i;
    return (table,disambiguationTable);
}
```

Il generatore è stato realizzato per mezzo di un programma Python che implementa l'algoritmo sopra descritto. Il software è costituito da un componente principale che espone un'interfaccia con il quale è possibile invocare la generazione su un dizionario che associa ad ogni chiave il corretto valore. Un secondo modulo Python da me implementato è utilizzato dal generatore per la produzione del codice C per la gestione e l'accesso della tabella generata. Questi due moduli principali sono quelli che, integrati in Papageno, forniscono le funzionalità obbiettivo del progetto.
Nella fase di implementazione è stata posta una particolare attenzione anche ai dettagli legati all'uso dello spazio in memoria, e per questo nella generazione dei sorgenti in C il tool effettua dei controlli per stabilire la *signedness* e il tipo più adeguato per rappresentare le strutture dati, al fine di garantire la rappresentazione meno costosa possibile. A titolo di esempio si consideri il caso di una tabella di disambiguazione con soli valori positivi: utilizzare un tipo *unsigned* potrebbe consentire di rappresentare con la metà dei *byte* i dati della tabella.

Per poter avere pieno controllo sull'occupazione in memoria delle strutture generate si è scelto di utilizzare, invece dei tipi short int e long int del C standard, che garantiscono so-

lo di essere rappresentati con rispettivamente non più o almeno i byte usati per un int, i tipi a dimensione fissata offerti dalla libreria **stdint.h**.
Al fine di ulteriore ottimizzazione inoltre le funzioni C prodotte sono definite usando le *keywords static* e *inline*, che chiedono al preprocessore di incorporare il *body* della funzione direttamente dove essa viene invocata nei file che includono il .h, invece che effettuare una chiamata come avviene normalmente per le procedure definite in file esterni.

3.3 Testing e Perfomances

Al fianco della progettazione ed implementazione dei moduli principali, la mia attività ha compreso anche la scrittura di alcuni *tool* accessori necessari per il testing del software prodotto. In particolare ho provveduto a realizzare un modulo in grado di produrre in modo automatico e randomico un dizionario di lunghezza arbitraria di coppie ((insieme di interi),intero), in modo da poter simulare degli input per il generatore, e uno script Python che si occupasse della selezione automatica dei casi di test e della scrittura di un insieme di file C da utilizzare per verificare il corretto funzionamento del codice sorgente prodotto dal generatore. Tutto il codice prodotto è stato sottoposto ad un intensa attività di testing e di misurazione delle performance risultando corretto e solido rispetto alla variazione dei dati in input. Durante lo sviluppo ho attuato una metodologia di progetto di tipo *test driven* avendo prima di tutto definito le componenti che avrebbero costituito il banco di prova per il software scritto. L'esecuzione dei test è stata condotta anche per input significativamente più grandi rispetto a quelli per cui il *tool* è chiamato ad operare nel contesto specifico di Papageno, e la risposta è stata sempre positiva, senza errori nè nella fase di hashing nè di produzione del sorgente C.

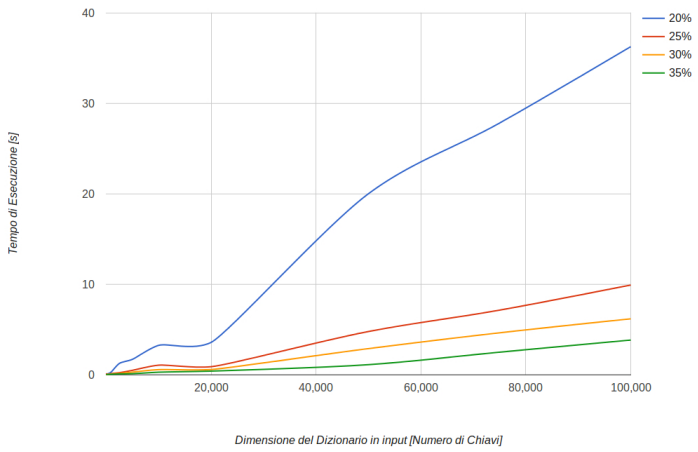

Figura 3.1: Relazione tra tempo di esecuzione, dimensione del set di chiavi e lunghezza percentuale della tabella intermedia

Il grafico in Figura 3.1 riassume i risultati raccolti nella fase di misurazione delle performance, e più in particolare mette in relazione il tempo di esecuzione dell'algoritmo di generazione

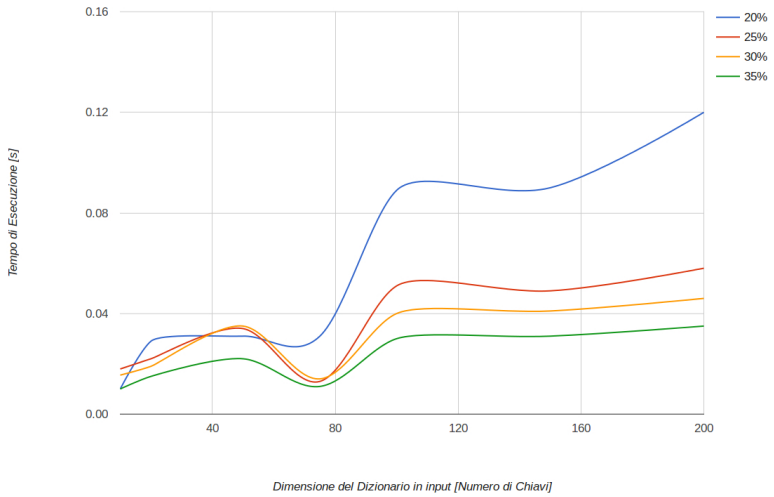

Figura 3.2: Relazione tra tempo di esecuzione, dimensione del set di chiavi e lunghezza della tabella intermedia per piccoli set

con la quantità di chiavi presenti nel dizionario, al variare della dimensione della tabella di disambiguazione, considerata in percentuale sulla dimensione della tabella finale di hashing. Dalla Figura 3.1 si nota un andamento lineare del tempo di esecuzione rispetto alla quantità di chiavi da collocare, e il coefficiente di linearità diminuisce sensibilmente all'aumentare della dimensione percentuale della tabella intermedia, fino quasi a diventare orizzontale quando il rapporto tra le dimensioni delle due tabelle è 1 : 3.
Per quanto riguarda il campo di applicazione dello strumento all'interno di Papageno si deve considerare che l'insieme delle chiavi altro non è che il set di *right hand sides* della grammatica, il quale, è tipicamente un oggetto di dimensioni molto limitate, solitamente dell'ordine di grandezza delle decine o centinaia di elementi. Nel grafico in Figura 3.2 è riportato l'andamento del tempo di esecuzione dell'algoritmo di generazione al variare della dimensione del dizionario in input, in un range limitato in modo verosimile rispetto all'impiego del tool all'interno di Papageno. Ulteriori considerazioni per l'interpretazione dei due grafici vanno svolte sul fatto che essendo l'algoritmo in parte randomizzato è in generale possibile che set di identica dimensione impieghino tempi molto differenti per essere collocati. I dati rappresentano la media di circa 10 misurazioni effettuate per ciascuna combinazione di valori, e nelle rilevazioni effettuate è stata notata una scarsa incidenza del fattore *random* sulla varianza delle cifre osservate; la deviazione standard percentuale assume valori che rientrano tra il 20% del valore medio e il 400% del valore medio, raggiungendo cifre superiori al 100% solamente nel caso di set in input molto piccoli sui quali il fattore casuale ha un impatto maggiormente elevato.

4 Conclusione

Perfect hash tables e minimal perfect hash tables sono strumenti che possono trovare notevoli applicazioni in svariati ambiti, grazie alle proprietà esposte in precedenza, prima tra tutte quella di poter essere generate in modo automatico.
L'impiego di strutture di questo tipo può fornire dei significativi miglioramenti in termini di performance al sistema in quanto esse sono specificatamente ideate per minimizzare il numero di accessi in memoria che, come noto, rappresenta uno dei fattori che causano maggiore rallentamento all'esecuzione di un processo.
Grazie al loro largo impiego in svariate applicazioni diffuse al giorno d'oggi è possibile reperire molti algoritmi di hashing dalle caratteristiche più differenti, selezionare quello ritenuto più opportuno ed adattarlo adeguatamente al dominio di applicazione desiderato.

Oltre ad un grande bagaglio relativo alle tecniche di hashing, questo progetto mi ha permesso e costretto ad approfondire notevolmente le mie conoscenze del linguaggio C, ad utilizzare il linguaggio Python, del quale ho potuto apprezzare le enormi potenzialità e che, grazie all'attività svolta, ora padroneggio in modo discreto, ad entrare a contatto con delle nozioni di base del processo di compilazione, nonchè mi ha fornito l'opportunità per mettere in pratica molte delle conoscenze teoriche acquisite tramite diversi corsi del percorso di laurea triennale, a partire dalla programmazione fino ai concetti dell'ingegneria del software.

L'attività di progetto collocata nell'ambito del corso da 5 CFU Progetto di Ingegneria Informatica, svolta tra Marzo e Luglio 2016, si conclude con il raggiungimento degli obiettivi preposti.

Un ringraziamento sentito va al Professor Barenghi per avermi supportato, aiutato e per tutto quello che mi ha insegnato durante lo svolgimento del progetto, e al Professor Mandrioli per avermi offerto questa opportunità.