UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

STUDIO DI NUOVI METODI STATICI E DINAMICI PER IL PROBLEMA DI "IP ADDRESS LOOKUP" NELL'ISTRADAMENTO EFFICIENTE DEI PACCHETTI IN INTERNET

Candidato
FILIPPO GERACI

Relatore:
Prof. ROBERTO GROSSI

Controrelatore:
Prof. MAURIZIO BONUCCELLI

ANNO ACCADEMICO 2002-2003

Indice

Capitolo 1

Introduzione

La continua crescita del bacino di utenti di Internet e l'incessante incremento delle necessità di questi hanno fatto in modo che il traffico di informazioni in rete stia subendo un fattore di crescita constante nel tempo.

Le informazioni su Internet viaggiano sotto forma di pacchetti che vengono spediti da un server a un altro fino al raggiungimento della destinazione finale. I computer che si occupano dello smistamento dei pacchetti si chiamano *router*. Per soddisfare le attuali esigenze di traffico, è necessario che i router riescano a smistare milioni di pacchetti al secondo.

Ricevuto un pacchetto, un router deve scegliere a quale vicino (*next hop*) mandarlo affinché raggiunga la propria destinazione. Il problema di stabilire quale percorso far seguire a un pacchetto si chiama *routing*. Per effettuare la scelta del next hop, i router consultano delle tabelle, chiamate *tabelle di routing*, utilizzando l'indirizzo di destinazione come chiave di accesso. Tale operazione di consultazione in una tabella di routing si chiama *IP address lookup*.

Il problema dell'IP address lookup è considerato uno dei maggiori colli di bottiglia nella realizzazione di router con alte prestazioni. L'operazione di lookup può essere descritta come un problema di *longest matching prefix*, ovvero la ricerca di una

stringa, appartenente a un determinato insieme, che sia il piú lungo prefisso della stringa fornita in ingresso.

Fino a poco tempo fa, si considerava impossibile ottenere prestazioni dell'ordine di qualche milione di lookup al secondo, come richiesto dalle attuali necessità di internet, se non con l'ausilio di costoso hardware specializzato. Oltre che per il rilevante peso economico, l'impiego di hardware apposito crea problemi di progettazione non alla portata di piccole e medie imprese che vogliano integrare in internet la loro rete aziendale. Inoltre diventa rapidamente obsoleto con l'evolvere della tecnologia.

La ricerca di questi ultimi anni è riuscita a dimostrare che il problema dell'IP address lookup può essere risolto tramite opportune soluzioni software su calcolatori di uso generale ottenendo prestazioni significative.

1.1 Un po' di storia

La storia di Internet inizia nel 1957 quando il governo degli Stati Uniti crea L'Advanced Research Project Agency (ARPA). Nel 1969 l'ARPA inizia la costruzione della prima rete, chiamata ARPANET, la quale collega: l'Università della California a Los Angeles, lo Stenford Research Insititute, l'UCSB e l'Univerità dello Utab.

Nel 1971 ARPANET conta 15 nodi e 25 computer collegati. Viene inventata in quell'anno la posta elettronica per lo scambio di lettere via computer.

Nel 1973 viene stabilito il primo collegamento internazionale con l'University college di Londra e il Royal radar in Norvegia.

Nel 1981 nascono BITNET e CSNET che rappresentano le prime reti non collegate ad ARPANET.

Nel 1983 nasce il TCP/IP per il trasporto delle informazioni in rete e l'interconnessione tra reti. Il TCP/IP diventa la famiglia di protocolli standard dei sistemi UNIX che ne favorisce molto la diffusione.

Nel 1984 Nasce il servizio DNS. Nello stesso anno i computer in rete sono già piú di 1000. La crescita del numero di computer connessi alla rete in quegli anni ha un andamento esponenziale: nel 1987 si superano le 10.000 unità, nell'89 le 100.000 per superare il milione nel 1992.

Gli anni novanta sono caratterizzati dalla nascita ed enorme sviluppo del web. Nato nel 1991, il web si basa sul concetto di ipertesto noto già da diversi anni. Sul web è possibile trovare una varietà di informazioni impensabile prima di allora. Cosí Internet smette di essere uno strumento di ricerca o di studio e si trasforma in un fenomeno sociale. Gli ultimi anni sono caratterizzati dalle chat in cui è possibile comunicare e conoscersi.

Il fenomeno degli ultimi anni è legato alla condivisioni degli archivi ed allo scambio di programmi, musica e quant'altro.

Oggi Internet è un aggregato di migliaia di piccole reti non omogenee con velocità che variano da pochi megabit a qualche gigabit e che comunicano per mezzo di IP.

1.2 Problemi connessi alla velocità di trasmissione

La trasmissione dei dati tra nodi della rete dipende dalla velocità di propagazione dei segnali, dalla capacità dei canali, e dalla velocità di trasmissione delle sorgenti.

La tecnologia delle telecomunicazioni è stata in grado di migliorare notevolmente le prestazioni dei canali di trasmissione. Da un lato, l'introduzione di nuove tecnologie come la trasmissione su fibra ottica o la trasmissione satellitare hanno reso disponibile alle aziende la possibilità di trasmettere informazioni nell'ordine dei gigabit al secondo. Dall'altro lato, un impiego piú efficiente delle tecnologie di trasmissione esistenti ha reso disponibile ai singoli utenti capacità di trasmettere nell'ordine del megabit. Questo è stato possibile mediante il passaggio alla trasmis-

sione digitale su doppino telefonico (DSL) che non ha costi gravosi per gli utenti in quanto non necessita del cablaggio di una nuova linea.

In Internet il traffico è costituito da pacchetti che vengono instradati dai router in base al loro indirizzo destinazione. L'aumento della velocità di trasmissione ha reso necessario un incremento delle prestazioni delle operazioni di routing dei pacchetti, pena il sottoutilizzo della rete. La crescita delle prestazioni dei processori non è riuscita a far fronte alle necessità del routing. Ci si è trovati nella necessità di progettare delle soluzioni hardware specifiche, ma queste si rivelano poco versatili oppure, paradossalmente, troppo onerose rispetto ai costi della rete. Per ovviare a questa situazione, a volte sono stati impiegati protocolli come ATM o il flooding che tentano di non elaborare a costo di intasare la rete con pacchetti inutili.

Si è già detto che nelle elaborazioni necessarie per il routing quella che limita maggiormente il raggiungimento di prestazioni elevate è il lookup. Questo avviene in quanto dietro al lookup si nasconde il problema del longest matching prefix che è di difficile soluzione.

L'aumento delle prestazioni delle trasmissioni di rete rischia di diventare inutile se non supportato da un efficiente routing. Per questa ragione il problema dell'IP address lookup è diventato un cosí importante oggetto di studio.

1.3 L'aumento del traffico e le nuove esigenze

L'avvento del World Wide Web ha avuto un impatto sociologico ed economico inaspettato. La possibilità di avere un sito, a costi ridotti, con il quale farsi conoscere ovunque è stato visto molto di buon grado anche dalle aziende di piccole dimensioni. Per la gente comune, invece, il web è diventato un modo semplice ed alla portata di tutti di reperire qualunque tipo di informazione e di comunicare. Con l'avvento delle chat e dei gruppi di discussione Internet è diventato un fenomeno sociale e

culturale. La possibilità di una maggior banda anche da casa e la diminuzione dei costi di connessione ha fatto nascere la moda della condivisione degli archivi. Questi fattori hanno portato prima ad un ingente aumento del bacino di utenza di internet e poi ad un sempre crescente aumento delle informazioni scambiate.

Per i router, invece, l'apertura di Internet alle masse è significato dover smistare una quantità sempre crescente di pacchetti per soddisfare le esigenze e doverlo fare quanto prima possibile. Dal punto di vista dei provider (ISP) che forniscono l'accesso alla rete, il problema della velocità è molto importante in quanto la concorrenza tra gestori non si basa soltanto sui prezzi, ma soprattutto sulla banda che si riesce a garantire agli utenti. Per questa ragione per gli ISP oltre al problema delle linee di trasmissione diventa cruciale il problema di un routing efficiente e quindi del lookup.

1.4 Contributi di questa tesi

In questa tesi vengono affrontati diversi problemi collegati a un veloce algoritmo di lookup esistente in letteratura [4], indicato con CDG dalle iniziali degli autori. In primo luogo, si fornisce una realizzazione efficiente per la costruzione della struttura di CDG. A partire da questo algoritmo, sono investigati metodi alternativi per ottenere prestazioni migliori in termini di velocità nell'effettuare le operazioni di lookup e di risparmio di spazio nella costruzione della tabella delle decisioni. In particolare, si descrive la progettazione e la realizzazione di un'evoluzione della struttura CDG di dimensioni ridotte e di un algoritmo per effettuare i lookup con prestazioni migliori.

Un ulteriore contributo consiste nella realizzazione dinamica della struttura CDG, evidando di doverla ricostruire interamente a partire dalla risultante tabella di routing. In questa tesi si discutono i problemi intrinseci dell'aggiornamento dinamico della struttura CDG. Si fornisce un algoritmo che, invece di dover ricostruire tutto,

utilizza una struttura intermedia compatta e facile da aggiornare. La procedura di aggiornamento è progettata per essere indifferentemente utilizzata sia nella versione originale dell'algoritmo che in quella proposta nella tesi.

Di tutti gli algoritmi proposti, oltre ad una descrizione logica, si fornisce una realizzazione scritta in ANSI C con la quale sono stati effettuati una serie di test intensivi. Inoltre, una problematica significativa riguarda la correttezza degli algoritmi per una particolare istanza (ovvero, una data tabella di routing). Si propone uno strumento di verifica che, presa in ingresso una tabella di routing, fornisce un elenco ristretto di interrogazioni con le quali asserire la correttezza delle risposte fornite dalla procedura di lookup per la tabella in questione. Il metodo si applica utilizzando le proprietà degli algoritmi di costruzione delle strutture qui proposte e potrebbe essere generalizzato, con opportune modifiche, ad altri metodi di lookup.

1.5 Struttura della tesi

Dopo una breve introduzione nel primo capitolo, viene presentato il problema dell'IP address lookup nel secondo capitolo, discutendo le problematiche connesse alla progettazione di algoritmi efficienti per la sua realizzazione.

Nel terzo capitolo viene presentata una rassegna delle piú importanti tecniche per la soluzione del problema del lookup focalizzando l'attenzione sulle tecniche apparse piú recentemente in letteratura.

Il quarto capitolo fornisce una panoramica degli algoritmi realizzati in questa tesi.

Il quinto capitolo discute la realizzazione di due algoritmi per la generazione della compressione dello spazio degli indirizzi, a partire da una tabella di routing.

Il sesto capitolo descrive la realizzazione della struttura chiamata CDG e propone una versione piú compatta chiamata SCDG.

Nel capitolo sette viene mostrato come effettuare il lookup e viene proposta una procedura piú efficiente che lo realizza sfruttando la cache di primo livello dei calcolatori.

Nel capitolo ottavo si descrive un meotodo efficiente per l'aggiornamento delle strutture, risolvendo alcuni dei problemi connessi.

Il nono capitolo affronta il problema della correttezza degli algoritmi, le cui prestazioni sono misurate nel capitolo dieci. Infine, nell'ultimo capitolo si traggono le conclusioni.

Il contributo del lavoro originale di tesi viene descritto nei capitoli che vanno dal quarto al decimo.

Capitolo 2

Il problema dell'IP address lookup

2.1 Il problema dell'IP address lookup

Gli algoritmi di IP address lookup [15] utilizzano l'indirizzo IP di destinazione di ogni pacchetto ricevuto come chiave per la ricerca del next hop nella propria tabella di routing.

Definizione 2.1.1. *Sia m la lunghezza in bit degli indirizzi IP. Per un determinato router, sia H il numero dei router ad esso collegati Una tabella di routing T è un insieme di coppie (P, hop), dove P è una stringa in $(0,1)*$ lunga al massimo m, hop è un intero compreso tra 1 e H, oppure il valore speciale di "no route to host".*

*In una tabella di routing ogni prefisso P contiene soltanto i bit che identificano una rete (*address prefix*) e non quelli che identificano gli host connessi alla rete.*

Su una tabella di routing sono consentiti due tipi di operazione: il lookup e l'update.

Definizione 2.1.2. *Dato un indirizzo IP x ed una tabella di routing T, il problema dell'IP address lookup è quello di trovare $(P_x, hop_x) \subset T$ tale che:*

- *P_x è un prefisso di x*

- $|P_x| > |P_j|$ *per ogni altra coppia* $(P_j, hop_j) \subset T$ *tale che* $(P_j, hop_j) \neq (P_x, hop_x)$ *e* P_j *è un prefisso di* x

L'operazione di *update* prevede la possibilità di inserire o eliminare una coppia dalla tabella di routing T

Definizione 2.1.3. *Sia* T *una tabella di routing:*

- *l'inserimento di una nuova coppia* (P, hop) *in* T *è l'operazione che genera una nuova tabella di routing*
- $T' = T \cup (P, hop)$,
- *l'eliminazione di una coppia* (P, hop) *da* T *è l'operazione che genera una nuova tabella* $T' = T - (P, hop)$.

2.2 Distribuzione degli indirizzi IP

Non essendo ancora disponibili tabelle di routing reali su IPv6 la sperimentazione è stata condotta su IPv4. Da questo momento in poi è a quest'ultimo che si farà riferimento.

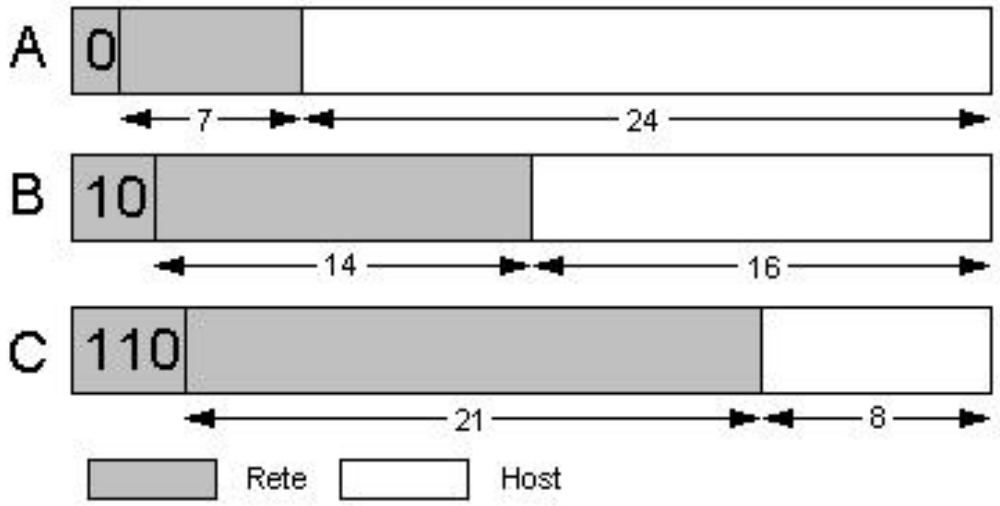
Definizione 2.2.1. *Un Indirizzo IP è un numero composto da 32 bit. Esso viene solitamente rappresentato da 4 numeri appartenenti all'intervallo 0, 255 separati da punto (.)*

Tale indirizzo è diviso in 2 parti: la prima identifica una rete (address prefix), la seconda un host all'interno di questa.

Scrivendo un address prefix in forma binaria è convenzione far seguire questo da un asterisco (*).

Esempio 2.2.1. *100* indica il prefisso lungo 3 bit che inizia con la sequenza 1,0,0. In notazione decimale un address prefix viene scritto come sequenza di numeri compresi tra 0 e 255 separati da punto (.) e seguiti da slash (/) e un intero che rappresenta la lunghezza in bit del prefisso. In notazione decimale il prefisso 100* si trasforma in 128/3 oppure 128.0.0.0/3 (128 in binario viene rappresentato da 10000000)*

Inizialmente in internet gli indirizzi IP erano stati suddivisi in classi chiamate A, B, e C; a seconda della classe di appartenenza l'address prefix era lungo 7, 14 o 21 bit.



Come mostrato in figura, un indirizzo era suddiviso in tre parti:

- uno header, di lunghezza variabile, utilizzato per stabilire a quale classe appartenava l'indirizzo
- un campo, chiamato *address prefix*, contenente l'indirizzo della rete
- un campo contenente l'indirizzo dell'host all'interno della rete.

I valori dell'header sono quelli mostrati in figura; si osservi che questi rappresentano un codice istantaneo ed univocamente decifrabile[13].

Con questo metodo gli indirizzi possono essere conservati in 3 tabelle diverse e l'operazione di lookup consiste nella ricerca di un match esatto nella tabella appropriata.

I difetti della divisione in classi riguardano: l'eccessiva crescita delle tabelle di routing e l'inefficiente uso dello spazio di indirizzamento.

Per ovviare a questi inconvenienti è stato introdotto un nuovo schema di indirizzamento, tuttora in uso il CIDR (Classless Inter-Domain Routing) [6]. Questo schema prevede la possibilità di avere address prefix di qualsiasi lunghezza e quindi di poter aggregare le reti a qualunque livello. In questo modo i router, nelle loro tabelle, non devono memorizzare le informazioni riguardante ogni rete, ma soltanto quelle relative alle reti aggregate.

Esempio 2.2.2. *Supponendo che le reti comprese tra il prefisso 192.168.160/22 ed il prefisso 192.168.188/22 siano riferite allo stesso next hop, invece di memorizzare nella tabella di routing le informazioni relative ad ognuna di esse, è possibile aggregarle nell'unica sottorete 192.168.160/19.*

Ammettendo adesso che la rete 192.168.172/22 non venga piú raggiunta tramite la stessa uscita, ma tramite una diversa, l'aggregazione sembra non essere piú possibile. Il problema viene facilmente risolto inserendo nella tabella di routing, la rete aggregata, (192.168.160/19) e l'eccezione generata dalla rete 192.168.172/22.

Dal momento che, nelle tabelle di routing possono essere presenti delle eccezioni, il problema del lookup si trasforma da un problema di match esatto ad uno di longest matching prefix.

2.3 Tabelle BGP

Il BGP (Border Gateway Protocol) [14, 17] è il protocollo di routing attualmente piú usato su internet per la sua robustezza e scalabilità.

I sistemi autonomi AS sono collegati tramite strutture chiamate Internet Service Provider (ISP).

Il protocollo BGP costruisce un grafo di AS basato sulle informazioni che si scambiano i router: questo grafo viene chiamato anche albero e ciascun AS viene identificato con un numero univoco. La connessione tra due AS si chiama *percorso.* Una collezione di percorsi forma a sua volta un percorso che viene utilizzato per raggiungere la destinazione

Le connessioni tra router adiacenti avvengono tramite il protocollo TCP che garantisce l'affidabilità delle connessioni; non è perciò necessario occuparsi della ritrasmissione di eventuali informazioni perse. Il difetto dell'impiego del TCP è che potrebbe nascondere momentanei malfunzionamenti dei link.

Le destinazioni IP sono espresse in termini di prefissi di indirizzo, ad ognuno dei quali è fornita la sequenza degli AS da attraversare. Per calcolare questa sequenza si usa il protocollo Path Vector il quale è una variante del protocollo Distance vector.

Nel caso non venga sempre scelto da tutti i nodi il percorso piú breve, con il protocollo Distance Vector vi è la possibilità che si verifichino dei fenomeni di divergenza. Il protocollo Path Vector, invece, prevede la possibilità di memorizzare l'intero percorso tra due reti, in modo da poter intercettare eventuali cicli.

Usualmente i router aggregano le informazioni di routing prima di propagarle, questo ha il duplice vantaggio di: minimizzare il traffico di routing e ridurre le dimensioni delle tabelle di routing.

Due router di due AS diversi collegati direttamente vengono chiamati *exterior*, mentre due router dello stesso AS, che hanno bisogno di una esplicita connessione

interna per trasmettersi le informazioni di routing, si chiamano *interior*.

Due router BGP si scambiano tutte le informazioni di routing quando viene stabilita la prima connessione.

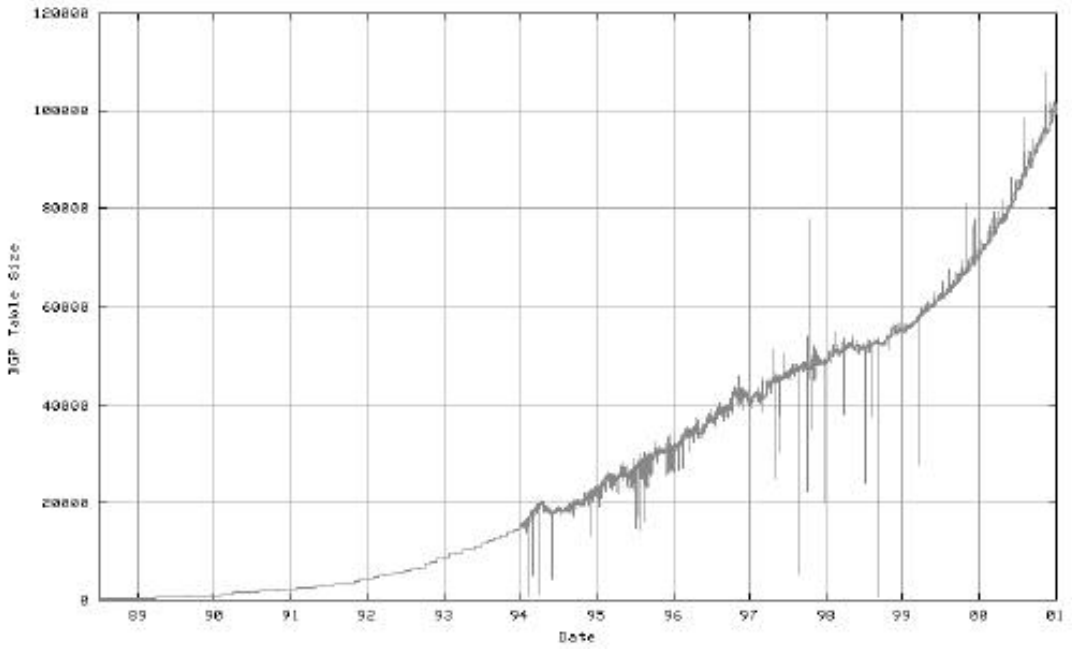
Gli aggiornamenti non vengono inviati periodicamente, ma soltanto quando occorrono dei cambiamenti nelle tabelle di routing e vengono inviate soltanto le route che sono cambiate.

Ad ogni router BGP vengono associate delle proprietà chiamate attributi, i piú importanti sono la "lista degli AS attraversati" e la "lista delle reti raggiungibili" Quando esistono diversi percorsi per il raggiungimento di una destinazione, per stabilire quale sia il migliore, interviene la valutazione di questi.

2.4 La crescita delle tabelle di routing

Le prestazioni degli algoritmi di IP address lookup vengono fortemente influenzate dalle dimensioni delle tabelle di routing; anche per questa ragione, a partire dal 1988, sono stati condotti degli studi sulle dimensioni e sul trend di crescita di queste. [8]

Nella fase tra il 1988 ed il 1994 la crescita delle tabelle di routing ha subito un andamento esponenziale soprattutto a causa alla crescita nell'utilizzo di indirizzi di classe C, ovvero dei prefissi di 24 bit. In risposta a questi ritmi di crescita, è stato introdotto lo schema CIDR. Per effetto di tale introduzione nel 1994 la dimensione delle tabelle di routing è rimasta stabile intorno alle 20.000 unità. Fino al 1998 la crescita delle tabelle è stata lineare e si è aggirata sulle 10.000 unità l'anno. Per ottenere questo andamento di crescita, però, si è resa necessaria una rinumerazione su larga scala delle reti. Dalla fine del 1998 la crescita delle tabelle di routing ha ricominciato ad essere esponenziale. Attualmente il numero di prefissi contenuto nelle tabelle BGP sfiora le 100.000 unità.

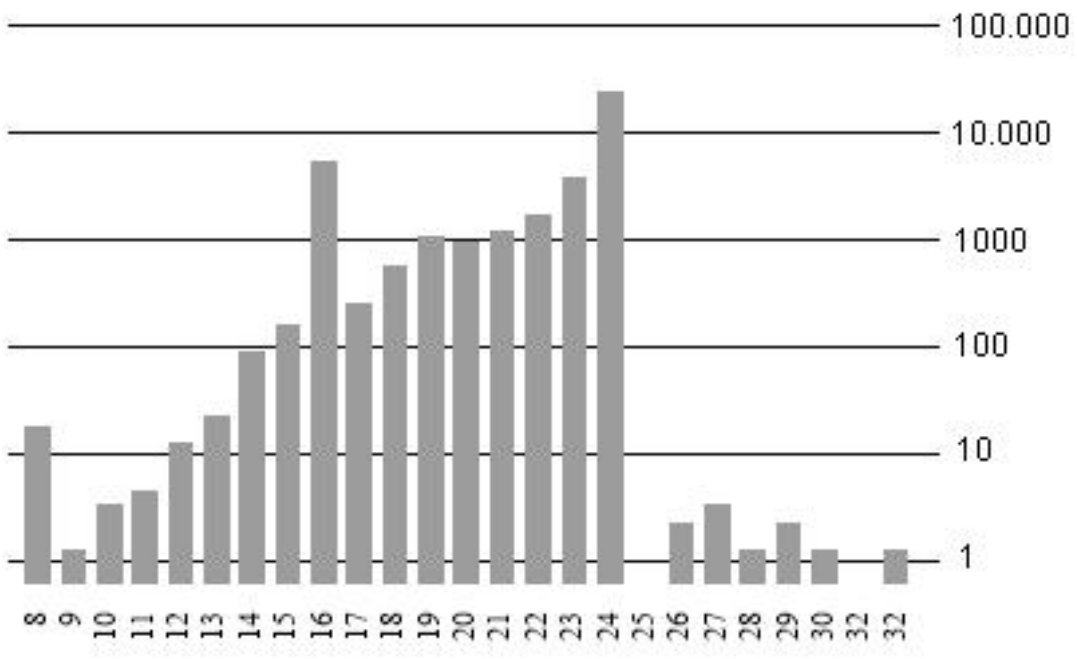


2.5 Distribuzione dei prefissi in base alla lunghezza

La crescita delle dimensioni delle tabelle di routing [8] non è uniforme, ma varia a seconda della lunghezza dei prefissi. L'uso estensivo degli indirizzi di classe C è da diversi anni scoraggiato per passare a blocchi di maggiori dimensioni. (/19 e piú recentemente /20). Ciò nonostante, è interessante notare che attualmente i prefissi lunghi 24 bit sono circa il 25% del totale nelle tabelle di routing e risultano essere quelle con il maggior tasso di crescita in assoluto. Le politiche di registrazione delle nuove reti prevedevano un limite minimo di /19 ciò ha fatto si che circa il 10% delle voci abbiano dimensione compresa tra /17 e /19. Oggi tale limite è /20 e rappresenta circa il 4% del totale delle voci e costituisce, in termini relativi, uno degli insiemi con maggiore crescita. Le voci relative a blocchi compresi tra /25 e /32 sono in proporzione quelle che stanno crescendo di piú. Esse cercano di fare in modo che il filtraggio delle voci in base alla lunghezza dei prefissi venga limitata quanto più possibile. Se non interverrà una correzione di questo fenomeno, probabilmente il

numero di queste voci nelle tabelle è destinate ancora a crescere.

Di seguito viene riportato un grafico rappresentante la distribuzione media delle route in base alla lunghezza dei prefissi.[18]



2.6 Filtraggio delle tabelle di routing

Il problema delle dimensioni delle tabelle di routing è importante per gli algoritmi di IP address lookup.

Tabelle di routing piú grandi significano maggiore occupazione di memoria e difficoltà nell'uso delle cache dei processori per migliorare le prestazioni degli algoritmi.

Il filtraggio [1] delle tabelle di routing è oggi uno dei piú attuali campi di ricerca. L'obiettivo è quello di eliminare errori, inconsistenze e ridondanze dalle tabelle di routing. Il prezzo che si paga con queste tecniche è quello di rischiare di rendere invisibile una piccola parte dello spazio di IP. Per quanto il problema del filtraggio,

può svolgere un ruolo importante nel miglioramento delle prestazioni di un algoritmo di lookup, esso si pone ad un livello piú alto del lavoro presentato in questa tesi.

La scelta di non occuparsi di una fase di pre-filtraggio delle tabelle di routing, è in linea con quella fatta dagli altri algoritmi di lookup. Questo ha anche reso possibile il confronto delle prestazioni dell'algoritmo presentato in questa tesi con altri algoritmi presenti in letteratura.

2.7 Proprietà dei prefissi

Si immagini l'intero spazio degli indirizzi IP come foglie di un albero binario bilanciato e completo dove ad ogni arco è associato il valore 0 o 1 [9]. La stringa che si ottiene, concatenando il valori dalla radice ad una foglia, è il valore binario dell'indirizzo IP rappresentato dalla foglia stessa. I nodi interni, ad eccezione della radice, rappresentano tutti i prefissi possibili. Le foglie rappresentano i prefissi degenere con prefix length uguale a 32.

Un prefisso, a questo punto, può essere visto come la radice di un sottoalbero le cui foglie hanno tutte lo stesso valore per il next hop.

Un modo alternativo di vedere un prefisso è il seguente:

Definizione 2.7.1. *Sia m la lunghezza di un indirizzo IP ($m = 32$ in IPv4) e siano p_1, p_2, ... cifre binarie. Un prefisso P lungo k bit si può rappresentare tramite $p_1p_2 \ldots p_k$. Siano $P_l = p_1p_2 \ldots p_k0 \ldots 0$ e $P_h = p_1p_2 \ldots p_k1 \ldots 1$ i due interi ottenuti aggiungendo $m - k$ bit 0 o bit 1 a P. Questi due numeri sono univocamente determinati per ogni prefisso. Essi rappresentano un intervallo $[P_l, P_h]$ di tutti gli indirizzi IP appartenenti al prefisso P.*

Definizione 2.7.2. *Due intervalli si intersecano se:*

- *si sovrappongono*
- *non sono contenuti l'uno dentro l'altro*

Lemma 2.7.1. *Due intervalli che rappresentano prefissi IP non si intersecano.*

Il problema dell'IP address lookup, da questo punto di vista, diventa, dato un indirizzo x, trovare il piú stretto intervallo in cui cade x.

Un'altra proprietà dei prefissi da mostrare in questa sede è la seguente:

Lemma 2.7.2. *Se un prefisso x lungo m copre l'intervallo $[P_l, P_h]$, i due prefissi $x0$ e $x1$, di lunghezza $m+1$, coprono lo stesso intervallo.*

Capitolo 3

Algoritmi per l'IP address lookup

Come si è visto il problema dell'IP address lookup è riconducibile al problema del *longest matching prefix*, per risolvere il quale esistono già degli algoritmi efficienti. Nello sviluppo di router con elevate prestazioni, però, si è riscontrata la necessità di costruire degli algoritmi ad hoc.

Gli approcci al problema sono di tre tipi:

- strutture dati software ed algoritmi di ricerca
- tecniche basate su hardware specifico
- tecniche per evitare il lookup tramite l'aggiunta di informazioni negli header dei pacchetti.

Nel seguito di questo capitolo si analizzano sinteticamente gli approcci classici alla soluzione del problema ed alcuni tra i piú importanti nuovi metodi.

Il capitolo si conclude con una breve panoramica delle soluzioni basate su hardware specializzato, su alcune tecniche per evitare il lookup e alcune considerazioni sull'influenza dell'hardware dei calcolatori general purpose sui moderni algoritmi di IP address lookup.

3.1 Gli approcci classici

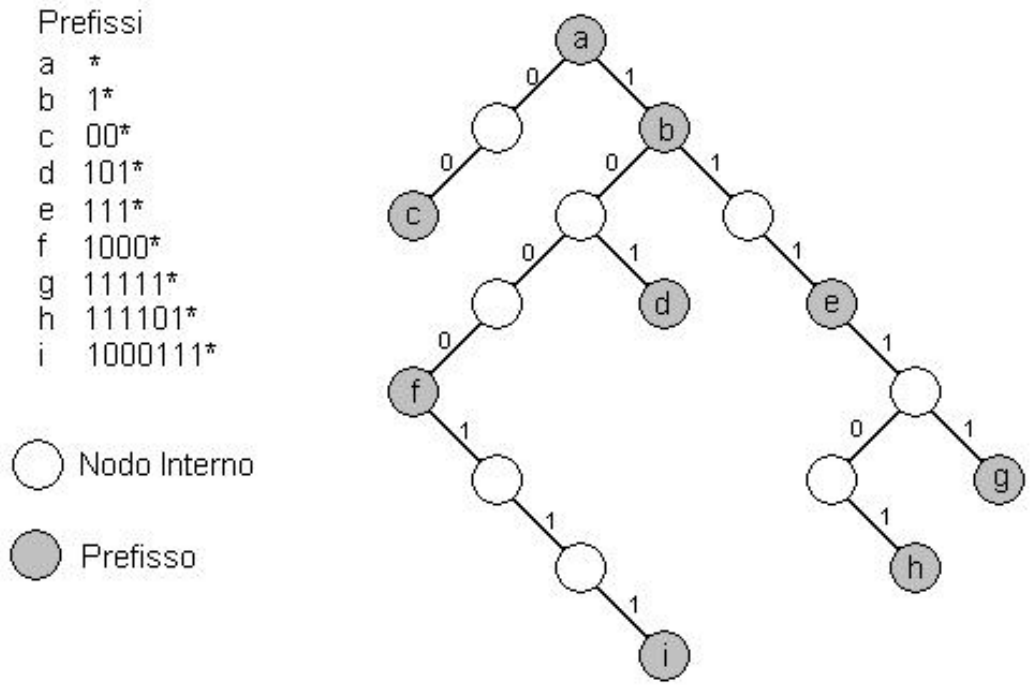
Gli approcci classici al problema dell IP address lookup, prevedono l'utilizzo dei trie o di loro evoluzioni. Un trie è un albero i cui archi sono etichettati.

3.1.1 Binary trie

Il modo piú semplice di memorizzare una tabella di routing è quello di usare un binary trie [15] ed etichettare i suoi archi con 0 e 1. Un nodo al livello L del binary trie, contiene l'insieme degli indirizzi che hanno come prefisso la stringa di L bit rappresentata dalle etichette degli archi tra la radice ed il nodo stesso. I nodi che rappresentano un address prefix contenuto nella tabella di routing, vengono marcati come prefissi, e contengono le informazioni aggiuntive necessarie per il lookup. E' possibile che questi nodi non siano foglie del binary trie, a causa delle eccezioni nell'aggregazione delle reti contenute nella tabella di routing.

L'operazione di lookup consiste in questo caso di una ricerca nel trie guidata dall'indirizzo di destinazione. Ogni volta che nella discesa si trova un nodo marcato come prefisso si tiene traccia di esso e se possibile, si continua la ricerca. Il nodo a profondità maggiore marcato come prefisso che si è raggiunto, è il longest matching prefix.

Sia l'operazione di lookup che quelle di insert e delete iniziano con una ricerca all'interno del trie. Per quanto riguarda il lookup, in particolare, è necessario fare tanti accessi alla memoria quanto vale il prefix length del longest matching prefix.

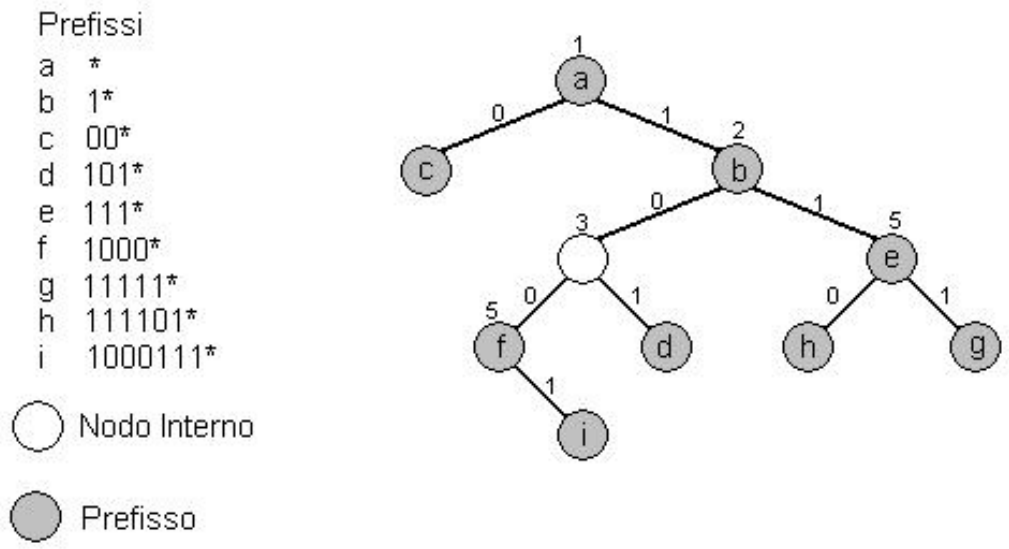


In figura viene mostrato un esempio di tabella di routing ed il binary trie ad essa associata. I nodi in grigio rappresentano i prefissi. Si può notare che alcuni prefissi non sono associati a foglie dell'albero, (nodi a,b,e,f). Questo avviene perché tali prefissi rappresentano una rete aggregata nella quale sono presenti delle eccezioni.

3.1.2 Path-compressed Trie

Nei binary trie capita spesso che un nodo o una sequenza di nodi abbiano un solo figlio. Nelle operazioni di ricerca, il bit associato all'arco uscente da nodi con un solo figlio non ha effetto sulle scelte del percorso. La compressione dei percorsi consiste nel collassare tutti i nodi con un solo figlio in un unico nodo. In questo modo però, la profondità di un nodo non corrisponde piú alla posizione del bit nell'indirizzo destinazione che determina la scelta. Per ovviare a questo problema ai nodi viene aggiunta un campo che indica qual'è il prossimo bit da esaminare.

In figura si vede un path-compressed trie equivalente a quello mostrato nel paragrafo precedente.



Dalla figura si nota che i nodi sono diminuiti da 17 a 10, è presente un solo nodo non rappresentante un prefisso contro gli otto presenti nel trie ed inoltre, la profondità massima dell'albero è scesa da 7 a 4.

Le prime due osservazioni mostrano la riduzione di occupazione di memoria, l'ultima, invece, mette in risalto come la procedura di lookup diventi più veloce, visto che i percorsi dalla radice ai prefissi risultano essere più brevi.

La procedura di aggiornamento della struttura risulta simile al caso precedente e non viene discusso nel dettaglio.

3.2 Nuovi approcci per risolvere il problema dell'IP address lookup

In questa sezione viene fornita una panoramica sulle strategie usate dai piú recenti algoritmi di IP address lookup.

Diversi algoritmi utilizzano la ricerca binaria per reperire le informazioni nelle loro strutture dati ma con due approcci significativamente diversi, infatti la ricerca può essere fatta:

- in base al valore
- in base alla lunghezza del prefisso.

Altre tecniche invece, puntano alla compressione delle tabelle per ridurne la ridondanza ed all'uso delle gerarchie di memoria del computer per ottenere un miglioramento delle prestazioni.

3.2.1 Ricerca in base al valore

La ricerca in base al valore è possibile solo a condizione che la lunghezza dei prefissi sia costante. Per far ciò il metodo piú semplice consiste nell'espandere a 32 bit tutti i prefissi. Questa scelta, però, non è praticabile a causa dell'ingente quantità di memoria che verrebbe occupata. In realtà non è necessario memorizzare tutti e 2^{32} gli indirizzi, in quanto ogni prefisso individua un intervallo di indirizzi IP con lo stesso next hop. E' quindi sufficiente memorizzare gli estremi dell'intervallo ed il next hop ad esso associato. A causa delle eccezioni presenti nelle tabelle di routing e dell'aggregazione, che può avvenire a vari livelli, due intervalli possono sovrapporsi. E' facile notare però, che in ogni caso questi intervalli, possono essere frammentati in maniera tale da ottenere una partizione dell'intero spazio degli indirizzi.

Gli intervalli partizionati possono essere memorizzati in un segment tree. Questo non è altro che un albero binario bilanciato i cui nodi contengono due valori, i quali rappresentano gli estremi di un intervallo.

Esempio 3.2.1. *Siano a, b, c interi tali che $a < b < c$, siano $[a, b]$ $[b + 1, c]$ due intervalli contigui. Essi rappresentano i figli di un nodo che memorizza l'intervallo $[a, c]$.*

A questo punto la procedura di lookup non è altro che una ricerca binaria nel segment tree volta a trovare in quale intervallo cade l'indirizzo destinazione.

3.2.2 Ricerca in base alla lunghezza del prefisso.

Come si è appena visto, il modo piú semplice di fare una ricerca rispetto alla lunghezza del prefisso, è quello di usare un trie. Il difetto dei trie, è quello di analizzare un solo bit alla volta. Nel seguito si mostra un esempio di algoritmo che riesce ad ovviare a questo problema.

3.2.2.1 Ricerca in base alla lunghezza del prefisso utilizzando i multibit trie.

In questo paragrafo descriviamo solo le idee che stanno alla base dei multibit trie [15], tralasciando di analizzare le scelte e le ottimizzazioni che si possono fare su questo schema.

La principale caratteristica del multibit trie è quella di analizzare ad ogni confronto, non un solo bit, ma piú bit simultaneamente. Il numero di bit simultaneamente analizzato si chiama **stride**. In un multibit trie ogni nodo ha 2^k figli se lo stride è di k bit. I multibit trie non consentono prefissi di lunghezza arbitraria, ma soltanto della lunghezza degli stride. Per questa ragione l'insieme dei prefissi deve essere trasformato in un insieme equivalente con le lunghezze appropriate. La trasformazione è molto semplice infatti sfrutta, grazie al lemma 2.7.2 il fatto di poter sostituire un prefisso di lunghezza k con due prefissi di lunghezza $k + 1$, dove il (k+1)-esimo bit

vale una volta 0 e l'altra 1. Questa operazione può essere effettuata ricorsivamente fino ad ottenere un insieme di prefissi della dimensione desiderata.

Esempio 3.2.2. *Il prefisso* $1*$ *può essere sostituito con i due prefissi* $10*$ *e* $11*$ *oppure con i quattro prefissi* $100*$, $101*$, $110*$, $111*$ *e cosí via.*

L'operazione di ricerca in un multibit trie è simile a quella che si effettua su un trie. La lunghezza degli stride influenza il numero di confronti necessari per il lookup e l'occupazione di memoria. Stride piú grandi prevedono un minore numero di confronti, ma un maggiore impiego di memoria. L'approccio seguito in questa tesi sfrutta anche questa idea.

3.2.2.2 Ricerca binaria in base alla lunghezza del prefisso

Si immagini una tabella di routing come un vettore disordinato, il lookup non è altro che una ricerca lineare in questa tabella, del valore che rappresenta il longest matching prefix. La complessità di questa ricerca è $O(N)$ dove N è il numero delle route. Vedremo in seguito un esempio di algoritmo che sfrutta la ricerca binaria per risolvere il problema.

La difficoltà di avere prefissi di lunghezza arbitraria è quella di non sapere quanti bit dell'indirizzo destinazione sono necessari per l'operazione di lookup e quindi di dover effettuare una ricerca lineare in base alla lunghezza dei prefissi. Questo tipo di problema è noto in letteratura come "predecessor problem".

La ricerca binaria in base alla lunghezza del prefisso, prevede di dividere l'insieme dei prefissi in gruppi della stessa lunghezza. In ogni sottoinsieme, è così possibile usare una funzione di hash per la ricerca. Nella normale ricerca binaria viene usato un valore chiave per decidere da quale parte bisogna procedere con la ricerca; in questo caso si deve usare come chiave il fatto di aver trovato o meno un match. Se il match viene trovato, la ricerca esclude i prefissi con una lunghezza minore, altrimenti

la scelta non è possibile. Sono state proposte delle tecniche, che non verranno trattate nel dettaglio, le quali risolvono questo problema grazie all'introduzione di alcuni prefissi speciali.

Ammettendo di avere una funzione di hashing perfetto che effettua la ricerca in tempo $O(1)$, questi algoritmi hanno una tempo di lookup di $O(\log_2 32)$ nel caso di IPv4.

3.2.3 Altre tecniche

Un'analisi piú accurata delle tabelle di routing ha mostrato che è possibile memorizzare tutte le informazioni di routing in strutture dati molto piccole con l'ausilio di sofisticate tecniche di compressione. L'impiego di strutture dati piccole rende anche possibile la memorizzazione di queste, o parte di queste, nelle cache dei processori, incrementando notevolmente le prestazioni degli algoritmi.

Da un altro punto di vista però, queste strutture dati sono tipicamente molto complesse da utilizzare e richiedono maggiori risorse di calcolo rispetto agli algoritmi fin qui presentati. E' perciò importante cercare il giusto compromesso tra dimensione delle strutture dati e risorse di calcolo impiegate.

Si vogliono mostrare alcuni esempi di algoritmi che risolvono il problema dell'IP address lookup, facendo delle considerazioni molto diverse da quelle degli algoritmi sopra esposti.

3.2.3.1 CDG lookup

Il CDG lookup [4] si pone tre importanti obiettivi

- limitare le dimensioni della struttura dati usata per il lookup in maniera tale che questa possa essere memorizzata nella cache L2 di un processore

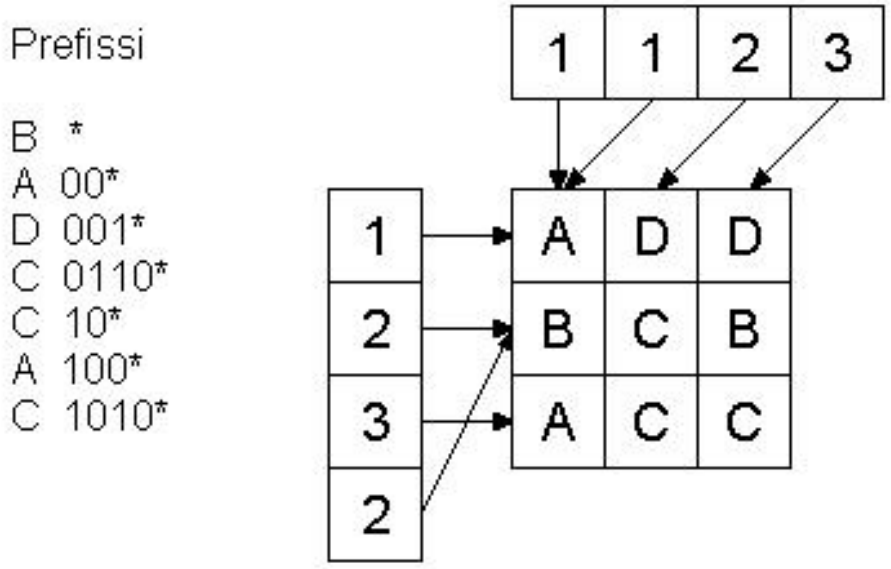
- limitare il numero di accessi alla memoria
- evitare di utilizzare istruzioni con latenza alta.

La struttura per il CDG lookup può essere vista come un multibit trie dove gli stride sono lunghi 16 bit. I figli della radice di questo trie saranno due vettori di 2^{16} elementi chiamati *riga* e *colonna*. Tutte le foglie vengono memorizzate in un'unica tabella, la tabella delle decisioni, alla quale si accede utilizzando i vettori riga e colonna.

Per l'accesso al vettore riga si utilizzano i primi 16 bit dell'indirizzo destinazione e per accedere al vettore colonna gli ultimi 16 bit. La tabella delle foglie è una tabella di 2^{16} x 2^{16} elementi. Non è pensabile mantenere una struttura di queste dimensioni in memoria, a causa dell'enorme dimensione che essa avrebbe. Per ridurne le dimensione le righe ripetute vengono eliminate e quelle restanti vengono compresse utilizzando l'algoritmo RLE.

I test condotti dagli autori, per piú di tre anni, hanno dimostrato che la struttura cosí ottenuta ha dimensioni nell'ordine del megabyte ed è quindi memorizzabile nella cache L2 di un processore.

Per quanto riguarda la procedura di lookup essa non fa nessuna elaborazione, si limita soltanto a fare tre accessi alla memoria.



In figura viene mostrato un esempio di struttura CDG. La tabella quadrata contenente i valori dei next hop è chiamata **tabella dei next hop** o **tabella delle decisioni**, il vettore contenente i riferimenti alle righe di tale tabella viene chiamato **vettore delle righe**, il vettore contenente gli offset delle colonne della tabella delle decisioni si chiama **vettore delle colonne**.

Si rimanda al capitolo 4 una descrizione piú accurata di questo algoritmo. In questa tesi, viene proposta un'implementazione piú efficiente dell'algoritmo per la costruzione della struttura dati del CDG lookup ed una sua evoluzione con lo scopo di migliorare le prestazioni di lookup e di minimizzare le dimensioni della tabella dei next hop.

3.2.3.2 Degermark lookup

L'algoritmo proposto da Degermark et al. [5] si pone come principale obiettivo quello di minimizzare il numero di accessi in memoria e di rendere la struttura dati quanto piú piccola possibile, affinché essa possa risiedere nella cache L2 di un processore.

La struttura dati proposta è essenzialmente un albero con tre livelli. Si consideri dapprima l'albero binario completo con 2^{32} foglie: un prefisso non è altro che la radice di un sottoalbero, le cui foglie contengono lo stesso next hop. L'albero binario che memorizza tutti i prefissi si chiama prefix tree. Il primo livello della struttura dati copre il prefix tree fino a profondità 16, il secondo da 17 a 24 e il terzo da 25 a 32. La ricerca in un livello della struttura dati richiede da 1 a 4 accessi in memoria e circa 50 istruzioni.

Questo tipo di struttura dati ha il pregio di essere molto piccola e quindi di poter essere facilmente memorizzata nella cache di un processore. Il numero di accessi per il lookup dipende dall'esito della ricerca nei vari livelli.

Considerando il numero esiguo di indirizzi da essa coperti, il terzo livello della struttura dati resterà spesso inutilizzato. Il difetto maggiore riguarda la necessità di utilizzare troppe istruzioni per la ricerca attraverso un livello della struttura.

La qualità importante che hanno strutture modulari come questa è il fatto che sembrano scalare bene su IPv6.

3.2.3.3 Retrie

In un recente studio Buchsbaum et al. propongono una struttura dati chiamata *retrie* [3]. Questa struttura dati è pensata per risolvere il problema del longest matching prefix su stringhe composte da caratteri appartenenti ad un alfabeto finito; per questo essa trova naturale applicazione anche sulle reti telefoniche.

L'idea che sta alla base del retrie è un'estensione della definizione 2.7.1, ovvero del fatto che due intervalli assocuati ad indirizzi IP non si intersecano, a stringhe su un alfabeto finito.

Il retrie è un albero costruito ricorsivamente nel quale ogni nodo interno è una

tabella indicizzata da uno stride di bit della stringa di cui si vuole fare il lookup. Le foglie sono contenute in un'altra tabella e contengono il valore del next hop.

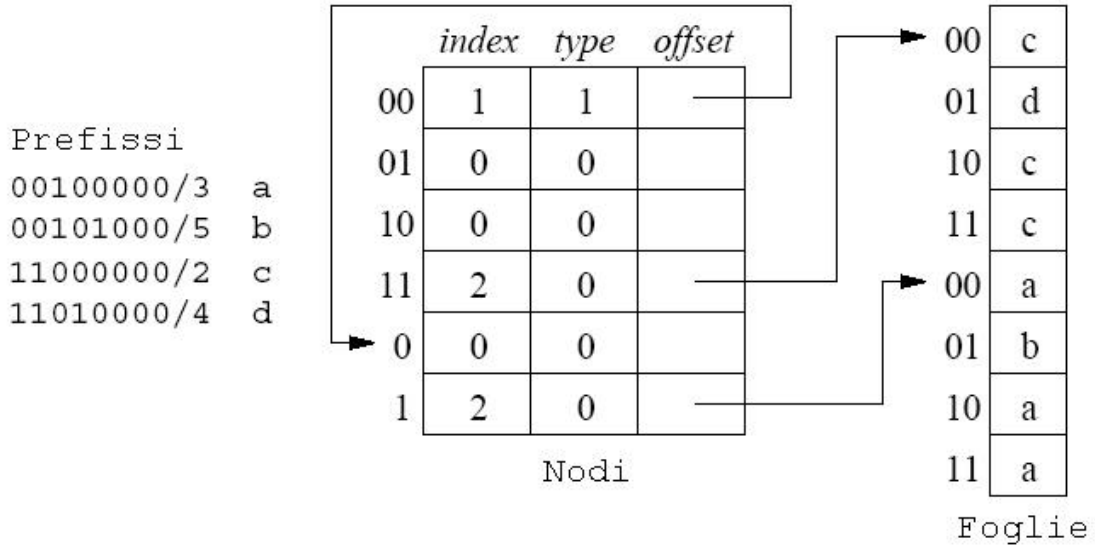
Il vettore dei nodi è formato da record con tre campi:

- **index** che indica la lunghezza dello stride nella tabella al livello successivo
- **type** è un bit che vale 0 se il prossimo nodo è interno e 1 se è una foglia
- **offset** è un puntatore al livello successivo

La profondità del retrie viene calcolata tramite un algoritmo di programmazione dinamica che ottimizza la lunghezza degli stride.

Per minimizzare la consumazione di memoria i nodi interni e le foglie vengono compressi attraverso la tecnica della shortest common superstring.

In figura viene mostrato un retrie di profondità tre e l'insieme delle stringhe di otto bit a partire dal quale è stato costruito.



3.3 Tecniche basate su hardware dedicato

Fino ad ora, si sono visti algoritmi per l'IP address lookup i quali affrontano il problema senza l'impiego di hardware dedicato. L'obiettivo della ricerca, in questo momento, è quello di ottenere prestazioni elevate su computer general purpose. Tuttavia in questo paragrafo si vuole dare un'idea di massima su una importante soluzione che richiede hardware dedicato.

Questa tecnica ha una performance eccellente rispetto alle soluzioni software; ma mostra grossi problemi in termini di progettazione e di scalabilità.

3.3.1 CAMs

I CAMs (Content-Addressable Memories) [12] sono delle memorie nelle quali l'accesso non viene fatto tramite un indirizzo ma comparando un valore chiave con tutti gli altri simultaneamente. Agli elementi contenuti in un CAM viene associata una priorità in maniera tale che, in caso di match con piú elementi venga scelto quello con priorità maggiore. La scalabilità delle dimensioni di un CAM non è un problema semplice, infatti aumentare le dimensioni significa dover fare più confronti simultanei e quindi aumentare il ritardo nella ricerca.

Per ottenere strutture di grosse dimensioni, i CAM possono essere messi in cascata.

Mediante i CAM si possono ottenere su IPv4 fino a 20 milioni di lookup per secondo. L'operazione di update è piú complicata ma esistono diverse tecniche che effettuano gli aggiornamenti in tempi abbastanza brevi.

3.4 Tecniche per evitare o ridurre il lookup

Il lookup, nella selezione del next hop, può essere evitato utilizzando come parametri di scelta l'indirizzo IP destinazione ed eventuali informazioni aggiuntive, presenti negli header dei pacchetti. Un router che adotti questo tipo di strategia, effettua le sue scelte indipendentemente dagli altri; per questa ragione non può far parte di una rete BGP.

3.4.1 Flooding

Il flooding [2] differisce dagli altri protocolli di routing per l'assenza della tabella di routing, infatti un router che adotti questo metodo trasmette ognii pacchetto ricevuto a qualunque altro router a cui è collegato, al eccezione di quello da cui gliè giunto.

Questo algoritmo è robusto e ottimo anche se tende a saturare la rete molto in fretta, infatti genera un inutile traffico nella maggior parte delle linee e, se non vengono adottate adeguate limitazioni del traffico, può portare ad una completa congestione della rete.

Per evitare che i pacchetti vengano instradati indefinitamente si possono utilizzare diverse tecniche:

- ad ogni pacchetto viene associato un valore di **timestamp** il quale viene decrementato ad ogni hop; questo valore deve essere non minore del massimo numero di hop necessari per raggiungere la destinazione. Quando il valore del timestamp arriva a zero il pacchetto è sicuramente arrivato a destinazione e quindi tutte le copie possono essere eliminate. Il valore piú elevato che può assumere il timestamp equivale alla distanza massima tra i router piú lontani presenti nella rete, tale valore si chiama *diametro della rete*.

- all'interno di ogni pacchetto viene inserito un numero seriale, unitamente all'identificativo del router che lo ha generato. Ogni router può cosí tenere traccia di quali pacchetti ha già inoltrato e scartare i duplicati.

3.4.2 Flow based routing

L'idea di fondo di questo algoritmo è il calcolo anticipato del traffico di ogni collegamento, basandosi naturalmente su ipotesi. A partire dal calcolo del ritardo medio di ogni linea, si può stabilire una stima del ritardo medio dell'intera rete. Il procedimento viene iterato applicando algoritmi di routing diversi fino a selezionare l'algoritmo che rende minimo il ritardo medio della rete.

3.4.3 Deflection

Il protocollo di deflection [2] è usato nel caso in cui il collo di bottiglia sia l'elaborazione. Questo metodo prevede di inviare un pacchetto semplicemente al router adiacente piú libero. Esistono delle tecniche correttive atte ad evitare il formarsi di cicli. Il protocollo è isolato nel senso che il router che lo impiega non ha informazioni sulla topologia della rete.

3.4.4 Interval routing

Il protocollo di interval routing [7] prevede che un router assegni un intervallo compreso fra 0 e 1 ad ognuno dei suoi vicini. Tali intervalli devono rappresentare una partizione di [0,1]. Ad ogni pacchetto viene assegnato un numero compreso tra 0 e 1. Il next hop è il vicino nel cui intervallo cade il numero associato al pacchetto.

3.5 Considerazioni sull'hardware e sulle prestazioni

Se fosse possibile conservare in un vettore il next hop associato ad ogni possibile indirizzo IP, il lookup non sarebbe altro che un singolo accesso a tale vettore, utilizzando come indice l'indirizzo IP di cui si sta facendo il lookup.

Sfortunatamente questo non è possibile a causa delle dimensioni che questo vettore dovrebbe avere, infatti, se fosse sufficiente un solo byte per conservare le informazioni di routing relative ad un singolo indirizzo IP, occorrerebbero 2^{32} byte, ovvero 4GB.

3.5.1 Gerarchie di memoria

Le memorie degli elaboratori moderni sono divise in gerarchie in base alla velocità ed alle dimensioni. Al livello piú basso si trovano i registri del processore sui quali, però, non si ha controllo; essi vengono utilizzati per i calcoli "momentanei" durante l'esecuzione di un processo.

Al livello successivo si trova la cache L1 che è molto prestante in termini di velocità di trasmissione ma di dimensioni molto ridotte.

Ad un livello ancora superiore si trovano le cache L2, le cui dimensioni attualmente raggiungono l'ordine dei MB e la cui latenza è ancora abbastanza bassa. Successivamente si passa alla memoria RAM che oggi raggiunge dimensioni che arrivano anche al GB, ma la cui latenza, è già inaccettabile se si vogliono ottenere prestazioni nell'ordine del milione di lookup per secondo.

Per questa ragione, le strutture dati utilizzate dagli algoritmi di IP lookup moderni devono essere quanto piú piccole possibile in modo da poter risiedere nelle cache. La differenza di latenza tra la cache L1 ed L2 è tale da giustificare avvolte una maggiore elaborazione per ottenere strutture dati piú piccole.

3.5.2 Gestione cache

La gestione della cache è un aspetto fondamentale del problema dell'IP address lookup.

Alcuni algoritmi prevedono l'impiego di due strutture dati distinte: la prima, molto piccola, capace di risiedere nella cache L1 contenente le informazioni relative alle ultime richieste di lookup, la seconda, piú grande, viene utilizzata soltanto in caso di fallimento della ricerca nella prima struttura.

Come mostrato da diversi studi, a regime questo approccio sembra funzionare bene a causa della località, probabilmente dovuta al traffico web, delle richieste di lookup. Ciò nonostante non esiste alcun criterio universale per stabilire quali informazioni conservare nella cache.

Capitolo 4

Descrizione dell'algoritmo di lookup

In questo capitolo viene presentata una descrizione a livello logico sia dell'algoritmo di CDG lookup che dello Split CDG lookup, focalizzando l'attenzione sulle problematiche ad essi collegate. Si lascia ai capitoli successivi la descrizione dei dettagli implementativi.

Negli esempi di questo capitolo si considerano indirizzi di quattro bit e si fa riferimento alla seguente tabella di routing:

Prefisso	Next Hop
ϵ	B
00*	A
001*	D
0110*	C
10*	C
100*	A
1110*	C

1 Tabella di routing di esempio

4.1 Descrizione logica dell'algoritmo di costruzione della tabella

Il CDG lookup prevede la costruzione di tre strutture:

- Una tabella chiamata **tabella dei next hop** o **tabella delle decisioni** contenente i next hop compressi tramite l'algoritmo RLE
- Un vettore di puntatori alle righe della tabella delle decisioni chiamato **vettore delle righe**
- Un vettore di offset chiamato **vettore delle colonne**.

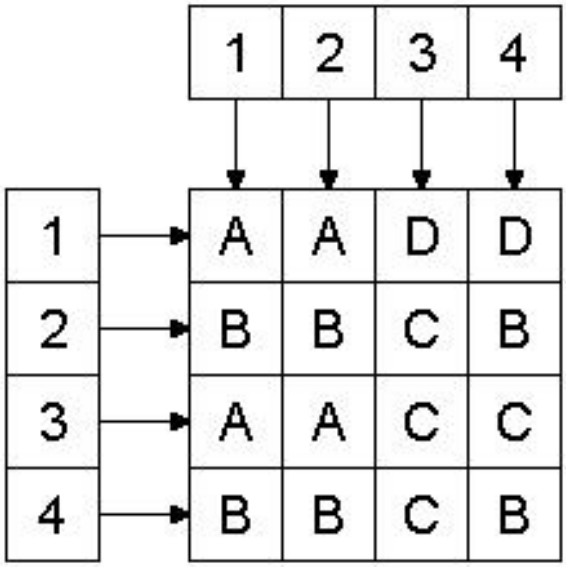
La costruzione di queste strutture dipende da una costante k, chiamata **K_SIZE**.

Scelto $k \in [0, 32]$, una tabella di routing può essere immaginata come composta da 2^k righe e 2^{32-k} colonne. Gli elementi di questa tabella contengono i riferimenti agli hop successivi. Per accedere ad una riga della tabella si utilizzano i primi k bit dell'indirizzo destinazione; l'offset all'interno della riga è rappresentato dai restanti $32 - k$ bit di tale indirizzo.

E' evidente che, indipendentemente da come viene scelto k, la tabella contiene 2^{32} elementi. Se k vale 16 la tabella è quadrata, ovvero ha tante righe quante colonne, per $k = 0$ la tabella si trasforma in un vettore riga, per $k = 32$ in un vettore colonna.

Invece di accedere direttamente alla tabella delle decisioni, utilizzando i bit dell'indirizzo destinazione, si possono creare due vettori **riga** e **colonna** che tengono traccia dell'ordinamento di righe e colonne della tabella delle decisioni. Naturalmente il vettore riga contiene 2^k elementi ed il vettore colonna 2^{32-k}.

In figura si mostra la struttura appena descritta, immaginando gli indirizzi IP lunghi 4 bit e scegliendo k uguale a due.



In questo modo, non solo è possibile permutare a proprio piacimento sia le righe chee le colonne della tabella delle decisioni, ma si possono eliminare righe e colonne ridondanti, semplicemente modificando i riferimenti contenuti nei vettori riga e colonna in maniera appropriata.

Si osservi che il numero di righe ridondanti tende a crescere con il decrescere della lunghezza delle righe e quindi col tendere di k a 32, ma questo ha effetti negativi sulla possibilità di ridurre il numero di colonne e viceversa se si fa tendere k a 0. Da questa considerazione si evince che il problema di scegliere k in maniera tale da minimizzare le dimensioni della tabella delle decisioni, è di cruciale importanza. Di contro la logica e alcuni dati sperimentali suggeriscono di cercare il valore di k intorno al 16.

4.1.1 Tabella delle decisioni CDG

Sia T una tabella di routing come in 2.1.1, essa può essere espansa in maniera tale che tutti i prefissi siano di lunghezza 32 grazie al lemma 2.7.2. In questo modo T

può essere vista come una tabella F i cui 2^{32} elementi sono coppie $C_i = (P_i, hop_i)$ per ognuna delle quali vale che $|P_i| = 32$. Sia F' la tabella che si ottiene da F ordinando tutti i C_i in maniera tale che i P_i risultino in ordine crescente.

Definizione 4.1.1. *Dati $k, n \in \mathbb{N}$:*

- *$first(k, n)$ restituisce un intero composto dai primi k bit di n*
- *$last(k, n)$ restituisce un intero composto dagli ultimi k bit di n*

Fissato ora un intero $k \in [0, 32]$, si creano 2^k insiemi T'_i nel seguente modo:

$\forall C_j \in F'$, se $first(k, P_j) = i$ e $last(k, P_j) = l$ allora $T'_{i,l} = (P_{i,l} = l, hop_{i,l} = hop_j) \in T'_i$.

Gli insiemi T'_i vengono chiamati **cluster.**

Esempio 4.1.1. *Dalla tabella 1 otteniamo i seguenti cluster*

$$T'_{00} = \{(00, A); (01, A); (10, D); (11, D)\}$$

$$T'_{01} = \{(00, B); (01, B); (10, C); (11, B)\}$$

$$T'_{10} = \{(00, A); (01, A); (10, C); (11, C)\}$$

$$T'_{11} = \{(00, B); (01, B); (10, C); (11, B)\}$$

Si costruisce adesso un vettore *Row*, di 2^k elementi, per ognuno dei quali si applicano le seguenti regole:

- $Row_i = T'_i$
- Se esiste $j < i$ tale che $T'_j = T'_i$, allora $Row_i = T'_j$ si elimina il cluster T'_i.

Sia α_k il numero di cluster che non sono stati eliminati, la tabella delle decisioni conterrà α_k righe.

Esempio 4.1.2. *Dai cluster creati nell'esempio 4.1.1 si ottiene:*

$$Row_1 = T'_{00}$$

$$Row_2 = T'_{01}$$

$$Row_3 = T'_{10}$$

$$Row_4 = T'_{01}$$

Il cluster T'_{11} viene eliminato perché uguale a T'_{01}, quindi α_k vale tre.

Per costruzuine, gli elementi di ogni cluster T'_i vengono ordinati in maniera tale che se $k < j$ $T'_{i,k}$ precede $T'_{i,j}$.

Per ogni T'_i viene creata una sequenza s_i che rappresenta la compressione RLE del cluster. Si procede nel seguente modo:

- si rimpiazza ogni sequenza massimale $T'_{i,j}$, $T'_{i,j+1}$,..., $T'_{i,j+l}$ tale che $hop_{i,j} = hop_{i,j+1} = \ldots = hop_{i,j+l}$ con la coppia $(hop_{i,j}, l_{i,j} = l + 1)$. Il numero $l + 1$ si chiama **run length.**

Esempio 4.1.3. *Riprendendo l'esempio precedente si ottiene:*

$$s_{00} = \{(A, 2); (D, 2)\}$$

$$s_{01} = \{(B, 2); (C, 1); (B, 1)\}$$

$$s_{10} = \{(A, 2); (C, 2)\}$$

Lemma 4.1.1. *Una coppia (hop, l) può essere scissa in due coppie (hop, l_1), (hop, l_2) tali che $l = l_1 + l_2$.*

Le coppie contenute in tutti gli s_i tramite il lemma 4.1.1, possono essere scisse in maniera tale da soddisfare la seguente proprietà:

Proprietà 1. $\forall i \in [1, \alpha_k]$ *i valori* $l_{i,j}$ *sono tutti uguali per* $j = 1, 2, \ldots$

Esempio 4.1.4. *Riprendendo l'esempio precedente si ottiene:*

$$s_{00} = \{(A, 2); (D, 1); (D, 1)\}$$

$$s_{01} = \{(B, 2); (C, 1); (B, 1)\}$$

$$s_{10} = \{(A, 2); (C, 1); (C, 1)\}$$

Questa proprietà fa si che il valore di l delle coppie in una data posizione sia uguale per tutti gli s_i.

Sia β_k il numero di coppie presente in un qualunque s_i, risulta evidente che β_k è univocamente determinato per qualunque $i \in [1, \alpha_k]$. Si nota inoltre che la sequenza dei valori l_i di s_i è uguale a quella di s_j per ogni $i, j \in [1, \alpha_k]$.

Esempio 4.1.5. β_k *vale 3.*

Si costruisce un vettore M, chiamato **metrica**, contenente la sequenza dei β_k valori di l di un qualsiasi s_i.

Esempio 4.1.6. *Gli elementi della metrica risultano cosí determinati*

$$M_1 = 2$$

$$M_2 = 1$$

$$M_3 = 1$$

Ogni s_i viene trasformato nella stringa s'_i formata dalla concatenazione di tutti i suoi hop.

Esempio 4.1.7. *Dall'esempio precedente si ha::*

$$s'_1 = ADD$$

$$s'_2 = BCB$$

$$s'_3 = ACC$$

Il vettore row viene aggiornato in maniera tale che $\forall i \in [1, \alpha_k]$, ogni riferimento a T'_i si trasformi in uno a s'_i. Il vettore row cosí modificato si chiama **vettore delle righe**, la tabella di α_k righe e β_k colonne, che ha ha come righe s'_i per $i \in [1, \alpha_k]$, si chiama **tabella delle decisioni**.

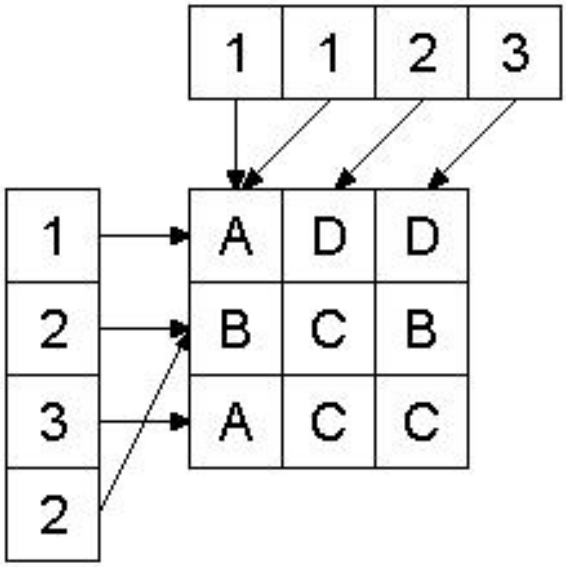
Sia $h = 32 - k$, si costruisce il **vettore delle colonne** composto da 2^h elementi nel modo seguente:

- per $i \in [1, \beta_k]$ si inserisce $M[i]$ volte il valore i nel vettore delle colonne a partire dalla prima posizione non ancora assegnata.

Risulta evidente che per costruzione gli elementi del vettore delle colonne, *col*, godono delle seguenti proprietà:

- risultano in ordine crescente
- $\forall i \in [1, 2^h]\ |col[i+1] - col[i]| \leq 1$

La struttura che si ottiene dall'algoritmo sopra esposto è la seguente:



4.1.2 Tabella delle decisioni Split CDG

In questa tesi viene proposto un metodo alternativo per costruire la tabella delle decisioni basato sul fatto che la probabilità di trovare righe ridondanti cresce al decrescere del numero di elementi che queste contengono. Il decrescere delle righe nel metodo CDG corrisponde a scegliere un valore per K_SIZE grande. Il problema che si crea è che al decrescere della lunghezza delle righe, aumenta la dimensione delle colonne; di conseguenza decresce la probabilità di trovare colonne ridondanti.

Il metodo che verrà adesso descritto, chiamato Split CDG (SCDG in breve), riesce a dimezzare la lunghezza delle righe mantenendo invariato il numero di colonne ridondanti.

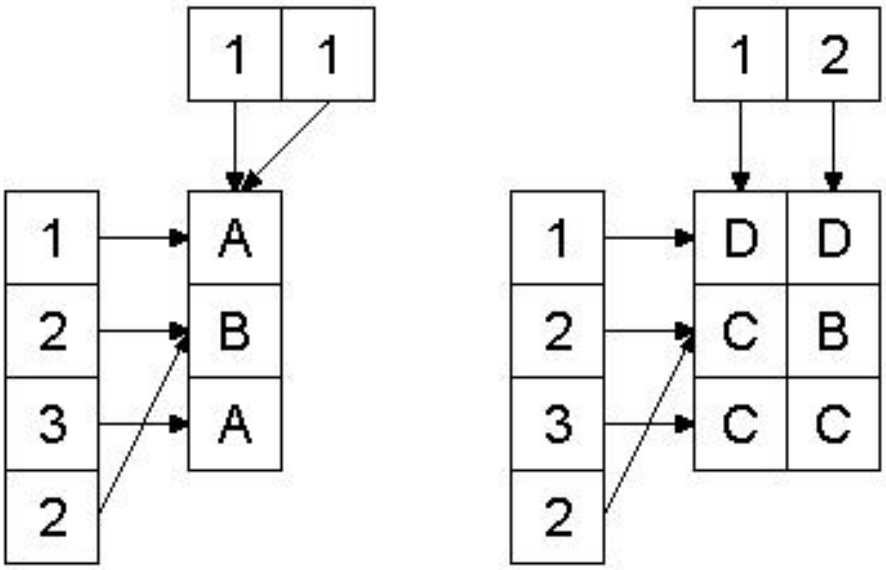
La tabella delle decisioni SCDG viene costruita a partire dalla tabella delle decisioni CDG appena discussa.

4.1.2.1 Costruzione delle tabello split

Sia $h = 32 - k$, allora il vettore delle colonne ha 2^h elementi ognuno dei quali fa riferimento ad una colonna della tabella delle decisioni. Per costruzione gli elementi del vettore delle colonne sono una sequenza di interi consecutivi ordinati progressivamente.

Sia S_1 l'insieme delle colonne puntate dai primi 2^{h-1} elementi del vettore delle colonne ed S_2 l'insieme delle colonne riferite dai restanti 2^{h-1} elementi. E' importante notare che l'intersezione di questi due insiemi non è necessariamente vuota. Si costruiscono due vettori row_1 e row_2 inizialmente copie del vettore delle righe. Si costruiscono altri due vettori col_1 e col_2 di 2^{h-1} elementi contenenti: col_1 i primi 2^{h-1} elementi del vettore delle colonne e col_2 i restanti 2^{h-1} elementi. Siano $|S_1|$ ed $|S_2|$ le cardinalità di S_1 ed S_2, agli elementi di col_2 deve essere sottratto il valore $|S_1|$.

Esempio 4.1.8. *In figura viene mostrato come si trasforma la struttura CDG presentata nel paragrafo precedente*



A questo punto, sono state ottenuto in pratica due differenti tabelle delle decisioni di 2^k righe e 2^{h-1} colonne.

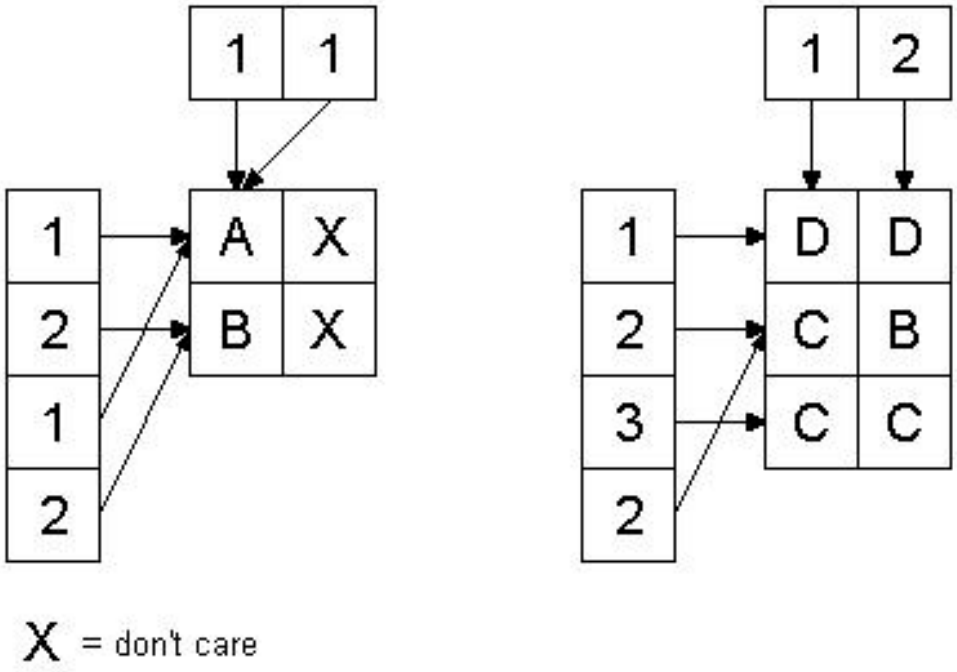
E' possibile che in almeno una delle due tabelle delle decisioni esistano ora delle righe uguali e quindi queste possano essere eliminate ricordando di aggiornare in maniera appropriata il vettore row associato.

4.1.2.2 Unione delle tabelle delle decisioni

Sia $|row_1|$ il numero di righe in S_1 e $|row_2|$ il numero di righe in S_2 dopo aver eliminato le righe ridondanti; S_1 ed S_2 possono essere unite concatenando ad ogni colonna di S_1 una colonna distinta di S_2, concatenando row_2 a row_1 e col_2 a col_1. Per effetto della concatenazione agli elementi di row_2 deve essere sommato il valore $|row_1|$.

Per unire S_1 ed S_2 questi devono avere la stessa cardinalità. Per ottenere ciò basta aggiungere all'insieme di cardinalità minore delle colonne fino ad ottenere la stessa cardinalità. Gli elementi delle colonne aggiunte conterranno il valore speciale di **don't care** il quale indica che tali elementi non saranno mai raggiungibili tramite la procedura di lookup.

Esempio 4.1.9. *Si supponga* $|row_1| < |row_2|$*, allora aggiungo ad* S_1 $|row_2| - |row_1|$ *colonne.*



col_1 e col_2 tengono traccia dell'ordinamento delle colonne di S_1 ed S_2 rispettivamente, è quindi possibile stabilire un ordinamento per S_1 ed S_2 con cui effettuare l'unione.

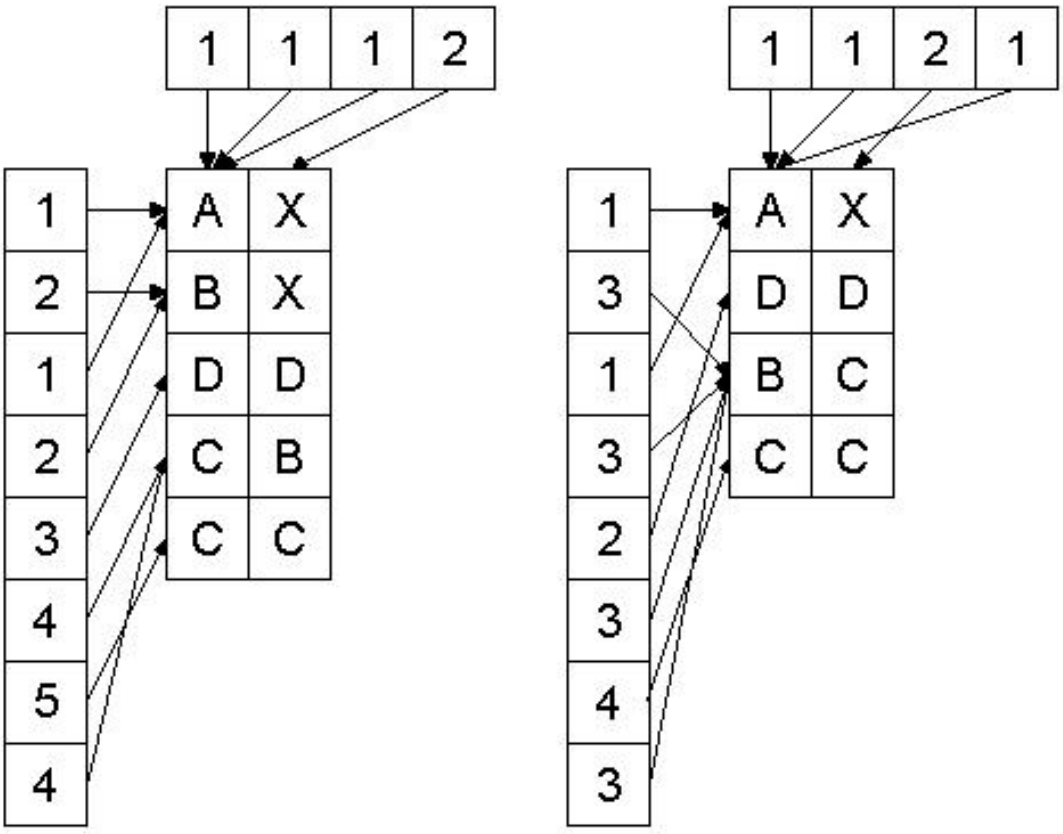
Sia S la tabella delle decisioni ottenuta dall'unione di S_1 ed S_2, essa potrebbe contenere delle righe uguali che potrebbero essere eliminate. Nel confronto tra le righe si stabilisce che il carattere don't care ha un match con qualunque carattere.

Risulta evidente che l'ordinamento di S_1 ed S_2 hanno un ruolo determinante nella compressione della tabella.

Esempio 4.1.10. *Nel esempio di questo capitolo, fissato un ordinamento per S_1, gli ordinamenti possibili di S_2 sono soltanto due. Nel primo caso si unisce alla prima colonna di S_1 la prima di S_2 ed alla seconda colonna di S_1 la seconda di S_2. Nel secondo caso, invece, si concatena alla prima colonna di S_1 la seconda di S_2 ed alla seconda colonna di S_1 la prima di S_2.*

La figura seguente mostra la tabella in cui sono state eliminate le righe ridondanti

dopo l'unione di S_1 ed S_2. La figura sinistra è riferita al primo caso mentre la destra al secondo. Si osserva che la tabella sulla destra ha dimensioni minori di quella sulla sinistra. Questo vuol dire che l'ordinamento di S_2 da cui si è ottenuta tale tabella è migliore.



Si deve a questo punto cercare il migliore ordinamento di S_1 ed S_2, dove per migliore ordinamento si intende quello che genererà la tabella delle decisioni piú piccola. Siano $s_1 \in S_1$ ed $s_2 \in S_2$ due colonne, l'effetto della concatenazione di s_1 ed s_2 sul numero di righe che avrà la tabella delle decisioni non dipende dalla loro posizione ma dagli elementi in esse contenuti. Fissato un ordinamento per S_1, questa proprietà riduce il problema alla ricerca della permutazione degli elementi di S_2 che renda minime le dimensioni della tabella delle decisioni.

Il problema di trovare tale permutazione sembra essere NP-completo. Per approssimare la ricerca di un ordinamento ottimo per S_2 è stata introdotta un'euristica. Sia $G = (O \cup D, A)$ un grafo bipartito dove O sono i nodi origine, D i nodi destinazione e A gli archi tra i O e D[16].

Ad ogni nodo di O viene associata un elemento di S_1, ad ogni nodo di D uno di S_2. Da ogni nodo $o \in O$ parte un arco per ogni nodo $d \in D$. Per le considerazioni fatte in precedenza sarà che $|O| = |D| = n$ da cui $|A| = n^2$. Ad ogni arco $a \in A$ viene assegnato un peso calcolato da una funzione che ha come parametri i due nodi da esso collegati. L'accoppiamento tra gli elementi di S_1 ed S_2 avviene nel seguente modo:

- Si calcola il matching bipartito di costo minimo,
- Si eliminano tutti gli archi non appartenenti a tale matching,
- Per ogni arco $a \in A$ non eliminato, si accoppiano gli elementi di S_1 ed S_2 associati ai nodi collegati tramite a.

Rimane aperto il problema di trovare una buona funzione per il calcolo dei pesi. A tal proposito sono stati condotti degli esperimenti sui dati reali per provare a rispondere a questa domanda ottenendo un risultato abbastanza inaspettato. Si noti che la funzione prende in ingresso una colonna di S_1 ed una di S_2 le quali possono essere viste come stringhe di caratteri. Sono stati condotti esperimenti utilizzando le seguenti funzioni:

- Hamming distance
- Hamming distance delle stringhe i cui caratteri sono stati ordinati lessicogrraficamente
- Somma dei match dei caratteri nella stessa posizione sulle due stringhe

- Somma dei caratteri uguali nelle due stringhe
- Calcolo della similarità tramite un algoritmo di programmazione dinamica.

La sorpresa è stata che nessuna di queste funzioni ha sortito buoni effetti, ma l'ordinamento migliore di S_2 trovato sperimentalmente è quello ottenuto tramite l'applicazione della funzione identità.

4.1.2.3 Shuffle del vettore delle righe

L'ultimo passo consiste nell'effettuare uno shuffle degli elementi di row.

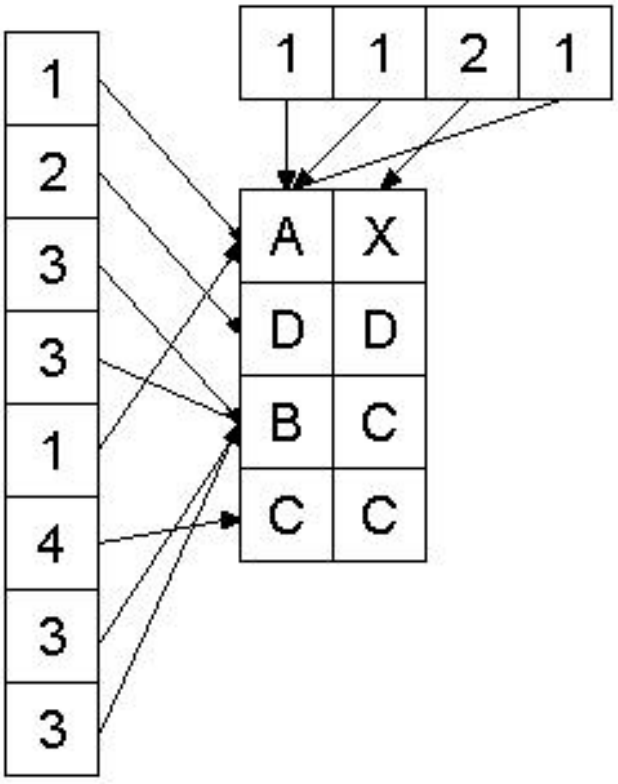
Per costruzione del vettore row, tramite il (k+1)-esimo bit dell'indirizzo destinazione si sceglie a quale metà si vuole accedere tramite i primi k bit. Precisamente se tale bit bale 0 si accede alla prima metà, se vale 1 alla seconda. Quello che si vuole ottenere, invece, è di poter usare i primi $k+1$ bit dell' indirizzo destinazione come indice del vettore row.

Per far questo si riordinano gli elementi di row nel seguente modo:

- $\forall i \in [0, 2^k - 1]\ row[2i] = row[i]$
- $\forall i \in [2^k, 2^{k+1} - 1]\ row[2(i - 2^k) + 1] = row[i]$

Quest'operazione ripristina la corrispondenza tra l'indice con cui si accede al vettore row ed i puntatori alla tabella delle decisioni. Si nota facilmente che questa corrispondenza viene persa con l'aggiornamento di row_1 e row_2. In realtà sarebbe possibile mantenere row_1 e row_2 coerenti durante ogni fase della costruzione della struttura SCDG, ma ciò richiederebbe un overhead di calcolo inutile.

Esempio 4.1.11. *In figura si mostra la struttura SCDG ottenuta.*



L'accesso alla tabella creata con questo metodo richiede l'utilizzo dei primi $k+1$ bit per il vettore delle righe e degli ultimi k per il vettore delle colonne.

Si nota subito che tutte le celle marcate col simbolo *don't care* non sono raggiungibili e che il (k+1)-esimo bit viene utilizzato sia per l'accesso alla riga che per quello alla colonna.

4.1.3 Costruzione del vettore delle colonne

Sia $h = 32 - k$ e sia x un indirizzo IP, il vettore delle colonne è un array lungo 2^h che viene indirizzato tramite $last(k, x)$; esso contiene gli offset delle righe della tabella delle decisioni per ogni possibile valore di $last(k, x)$. Nella tabella SCDG, il vettore delle colonne può essere visto come la concatenazione di due vettori lunghi

2^{h-1}, nel primo per costruzione gli elementi sono in ordine crescente, nel secondo la distribuzione degli elementi dipende da come sono stati permutati gli elementi di S_2. Nel caso si sia scelta la funzione identità per ordinare S_2, anche la seconda metà del vettore delle colonne risulta in ordine crescente.

In entrambi i casi il vettore delle colonne è composto da lunghe sequenze di uno stesso numero che si ripete.

4.1.3.1 Vettore delle colonne

Per la costruzione del vettore delle colonne si utilizza un array M di lunghezze dei run RLE, chiamato metrica. Per le considerazioni appena fatte il caso della tabella SCDG può essere riportato al caso CDG trattando le due metà del vettore delle colonne come due vettori separati ad ognuno dei quali viene associata una metrica. Gli elementi della seconda metà del vettore delle colonne verranno opportunamente permutati in base all'ordinamento di S_2. Grazie a queste considerazioni si può trattare soltanto il caso della tabella CDG .

Il vettore M viene creato a partire dalle sequenze s_i definite nel paragrafo 4.1.1 che rappresentano la compressione RLE dei cluster . Gli elementi di s_i sono coppie $(hop_{i,j},l_{i,j})$ dove $l_{i,j}$ è il run length associato al valore $hop_{i,j}$. Gli s_i sono creati in maniera tale che, scelto j, valga la seguente proprietà: $\forall p, q \in [1, \alpha_k]\ l_{p,j} = l_{q,j}$.

La sequenza dei valori di $l_{i,j}$ di un qualunque s_i rappresenta il vettore M.

Una volta costruita la metrica M, la realizzazione del vettore delle colonne è molto semplice, sia β_k definito come nel paragrafo 4.1.1, $\forall i \in [1, \beta_k]$ si aggiunge al vettore delle colonne $M[i]$ elementi contenenti il valore i.

4.1.3.2 Vettore dei bucket

La struttura descritta in questo paragrafo rappresenta uno dei risultati piú importanti proposti in questa tesi, oltre a ridurre le dimensioni del vettore delle colonne, con questa struttura è stato possibile ottenere delle prestazioni significativamente migliori nella procedura di lookup.

Gli elementi del vettore delle colonne sono numeri maggiori o uguali a zero. Nella realizzazione pratica si può verificare se essi possono essere memorizzati in un byte oppure sono necessari due byte. Per $h = 16$, non si può mai verificare che siano necessari piú di due byte dal momento che una riga non può mai contenere piú di 2^{16} elementi.

Dai dati sperimentali risulta che nella maggior parte dei casi sono necessari due byte per la rabella CDG ed un solo byte per la tabella SCDG. Nel primo caso il vettore delle colonne occupa 2^{h+1} byte che, per $h = 16$, vuol dire 128Kb, nel secondo caso, invece, l'occupazione di memoria è di 2^h byte, ovvero di 64Kb.

Si vorrebbe cercare di ridurre le dimensioni del vettore delle colonne in maniera tale da poterlo memorizzare nella cache L1. Per far questo si introduce l'idea di *bucket*.

Il vettore delle colonne viene partizionato in insiemi della stessa dimensione chiamati bucket. Come dimensione del bucket si sceglie il valore $b = 2^m$ tale che $2^m \leq 2^h/\beta_k < 2^{m+1}$. Il vettore delle colonne è costituito in questo modo da $|B| = 2^h/b$ bucket. Se gli elementi di un bucket non sono tutti uguali, si dice che questo contiene un **fault**. Sia F la somma dei bucket contenenti un fault. Si può adesso costruire il vettore B contenente $|B| + Fb$ elementi. Il vettore dei bucket è logicamente partizionato in due segmenti:

- il primo contenente i primi $|B|$ elementi si chiama **segmento dei riferimenti**
- l'altro, contenente gli elementi restanti, si chiama **segmento dei bucket**.

B viene riempito nel seguente modo:

- se l'i-esimo bucket non contiene fault, allora $B[i]$ contiene il valore di un elemento qualsiasi del bucket.
- se l'i-esimo bucket contiene il j-esimo fault, $B[i] = -(|B|+(j-1)b)$ ed il bucket viene interamente copiato in B a partire dalla posizione $(|B| + (j-1)b)$.

Il valore negativo serve per distinguere se il contenuto della cella esaminata rappresenta il valore contenuto dentro un bucket, ovvero un riferimento al segmento dei bucket.

Si vedrà nel paragrafo 4.2.2 come utilizzare questa struttura al posto del vettore delle colonne.

La dimensione del vettore delle colonne cosí trasformato risulta essere dipendente da $|B|$ e da F. Il diminuire delle dimensioni dei bucket fa aumentare leggermente le dimensioni del vettore dei bucket facendo diminuire le probabilità di trovare dei fault. Ogni bucket contenente un fault va ricopiato in coda al vettore dei bucket facendone aumentare le dimensioni. Per quanto non esista un limite teorico al numero di fault che possono essere trovati, dai dati sperimentali si è osservato che questi sono in numero molto esiguo.

La sperimentazione ha dimostrato che il vettore bucket è sempre molto piú compatto del vettore delle colonne, il quale rappresenta il limite superiore che questo può raggiungere, e che è quasi sempre stato possibile memorizzarlo interamente nella cache L1 del processore con cui è stata condotta la sperimentazione. Si veda a tal proposito il capitolo 10.

4.2 Descrizione logica dell'algoritmo di lookup

4.2.1 La procedura di lookup

La procedura di lookup è molto semplice ed intuitiva. L'indirizzo destinazione, x, viene spezzato in due parti per mezzo di $first(k, x)$ e $last(h, x)$. La prima parte rappresenta l'indice con cui accedere all'elemento del vettore delle righe in cui è contenuto il puntatore alla riga appropriata della tabella delle decisioni. la seconda rappresenta l'indice con cui effettuare l'accesso al vettore delle colonne per ottenere l'offset all'interno della riga selezionata.

Nel caso della tabella SCDG la procedura di lookup non cambia, si ha soltanto che $k = 17$ ed $h = 16$.

La procedura di lookup viene fatta per mezzo della seguente istruzione:

$$lookup = TabellaDecisioni[VettRighe[first(k, x)]][VettColonne[last(h, x)]]$$

Come si può osservare, il lookup non prevede l'impiego di nessun tipo di elaborazione, ci si limita a fare soltanto tre accessi in memoria. Se le dimensioni della tabella delle decisioni sono abbastanza ridotte da poter essere memorizzata nella cache L2, un lookup non fa altro che tre accessi alla cache.

4.2.2 La procedura di bucket lookup

Nel paragrafo 4.1.3.2 si è provato a sostituire il vettore delle colonne con il **vettore dei bucket** di dimensioni ridotte. La proprietà desiderabile del secondo vettore è che questo abbia dimensioni tali da poter essere memorizzato nella cache L1.

Un importante risultato che si è ottenuto in questa tesi tramite l'introduzione del vettore bucket è quello di essere riusciti a migliorare le prestazioni della procedura di lookup del 44.59%.

Per quanto riguarda la procedura di lookup, con l'impiego del vettore dei bucket, cambia il modo di selezionare l'offset; non cambia invece il modo con cui viene scelta la riga all'interno della tabella delle decisioni.

La procedura che viene adesso descritta per il reperimento dell'offset è la medesima sia nel caso della CDG table che nel caso della tabella SCDG.

Utilizzando la notazione del paragrafo 4.1.3.2, si ricorda che $|B| = 2^m$, per accedere al segmneto dei riferimenti si utilizzeranno i primi m bit di $last(h, x)$. Questo valore è facilmente calcolabile grazie all'annidamento delle funzioni $first\,()$ e $last\,()$, ovvero $first(m, last(h, x))$.

Sia $val = bucket[first(m, last(h, x))]$, se val è positivo esso rappresenta l'offset cercato. Nel caso contrario è necessario un ulteriore accesso al vettore dei bucket. Si somma il numero ottenuto dagli ultimi $h - m$ bit di $last(h, x)$ a val cambiato di segno. Formalmente si accede alla posizione $-bucket[first(m, last(h, x))] + last(h - m, last(h, x))$ del vettore dei bucket ottenendo questa volta l'offset desiderato.

Nel caso in cui l'offset venga trovato subito, la complessità dell'algoritmo di lookup non cambia, in caso contrario è necessario effettuare un quarto accesso alla memoria e due operazioni logiche sui bit.

Affinché il vettore bucket sia di dimensioni ridotte, i fault al suo interno devono essere in numero esiguo; i dati sperimentali confermano questa tesi. Si può supporre che il primo accesso al vettore dei bucket termini quasi sempre con un valore positivo. Per questa ragione il sistema di branch prediction dei processori funziona molto bene e fa si, che nella maggioranza dei casi, il costrutto condizionale per il test del segno di val sia del tutto ininfluente nel calcolo delle prestazioni.

Supponendo di aver potuto memorizzare il vettore dei bucket nella cache L1, nella maggior parte dei casi, le prestazioni del lookup migliorano notevolmente.

Non vengono fatte assunzioni sulla politica con la quale viene gestita la cache L1

dal processore, infatti ricevendo almeno un accesso per ogni lookup, se memorizzato in L1, il vettore dei bucket tenderà a continuare a risiedervi indipendentemente da tale politica.

Nel caso in cui val è negativo, il costrutto condizionale ha un peso sul calcolo delle prestazioni assieme alle due operazioni logiche ed al quarto accesso alla memoria. Non è possibile stabilire se un accesso alla cache L2 abbia una latenza maggiore o minore rispetto a 2 accessi ad L1 e all'overhead di elaborazione necessario, perché questa dipende dalla particolare architettura utilizzata.

Nel calcolo delle prestazioni di lookup, si osserva che i casi in cui sono necessari quattro accessi, sono in numero trascurabile rispetto ai casi in cui ne necessitano soltanto tre. Si conclude quindi che se è possibile memorizzare il vettore dei bucket nella cache L1, le prestazioni dell'algoritmo di lookup migliorano notevolmente.

4.3 Descrizione logica dell'algoritmo di update

L'operazione di aggiornamento della tabella di routing da parte di un router BGP avviene molto frequentemente, per questa ragione è necessario che gli algoritmi di lookup abbiano una procedura di aggiornamento delle strutture dati molto efficiente.

L'algoritmo CDG prevedeva di ricostruire l'intera struttura ad ogni aggiornamento. Per far questo senza interrompere le operazioni di lookup, si utilizzavano due banchi di memoria, mentre in uno si effettuavano i lookup, nell'altro si ricostruiva la tabella. Si teneva in memoria una copia della tabella T che veniva aggiornata come descritto dalla definizione 2.1.3. A partire da questa veniva ricostruita l'intera struttura.

Un risultato importante ottenuto in questa tesi è la realizzazione di una procedura di update, utilizzabile sia per il CDG lookup che per SCDG lookup, che risolve parzialmente il problema.

Invece di mantenere in memoria T che occupa molto spazio si utilizza un vettore chiamato V_{RLE} di dimensioni piú compatte. Questo vettore contiene una stringa che rappresenta la compressione RLE dell'intero spazio degli indirizzi IP. Questa struttura presenta diversi vantaggi:

- L'aggiornamento è molto facile
- Occupa poca memoria
- Sia CDG lookup che SCDG lookup possono partire da questa struttura per creare la tabella delle decisioni
- E molto facile ricavare da questa struttura i cluster che risultano essere già compressi

Prendendo in prestito la terminologia dei compilatori, il vettore V_{RLE} può essere visto come una sorta di codice intermedio; quindi esiste un front end che parte da T per ottenere V_{RLE} ed un back end che da questo ottiene la struttura CDG ovvero SCDG.

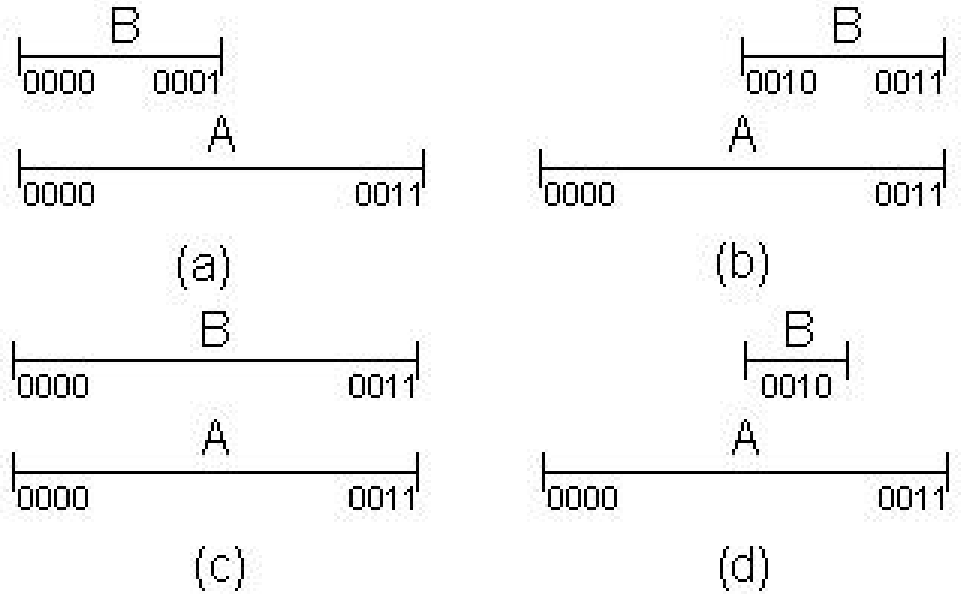
La procedura di aggiornamento qui introdotta ha comunque bisogno di una fase di ricostruzione, ma vengono evitate tutte le operazioni del front end.

Dal punto di vista implementativo, comunque, non sarebbe stato possibile eliminare la ricostruzione in quanto l'inserimento di una riga o di una colonna nella tabella delle decisioni impone un ridimensionamento del buffer che la contiene e quindi la copia fisica delle informazioni da un segmento di memoria ad un altro.

Per l'aggiornamento del vettore V_{RLE} viene utilizzato un trie le cui foglie rappresentano gli estremi degli intervalli in cui è partizionato lo spazio degli indirizzi IP. Gli inserimenti e le cancellazioni vengono realizzate tramite una ricerca nel trie. Grazie ad una visita in profondità del trie è possibile ottenere la versione aggiornata de V_{RLE}.

4.3.1 La procedura di inserimento

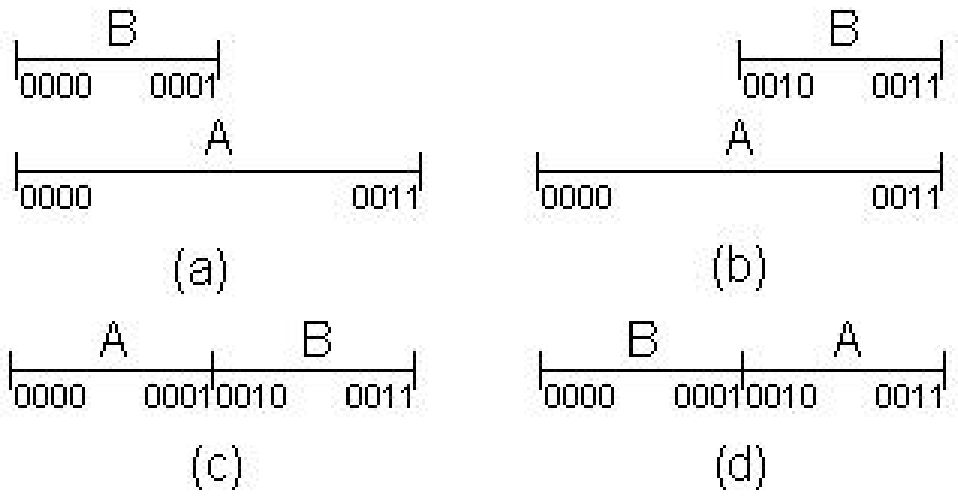
L'inserimento di una nuova coppia (P, hop_P) in T, da luogo alla creazione di un nuovo intervallo nello spazio di indirizzamento. Si possono verificare, come mostrato anche dall'esempio in figura, i seguenti casi:



1. Il prefisso da inserire ha un estremo non in comune con un altro intervallo (a) (b)
2. Il prefisso da inserire fa riferimento ad un intervallo già esistente (c)
3. Il prefisso da inserire fa riferimento ad un intervallo interamente contenuto in un altro. (d)

Il lemma 2.7.1 garantisce che due intervalli generati da indirizzi IP non si intersecano rendendo sufficiente la trattazione dei casi sopra elencati.

Nel primo caso può avvenire che l'estremo sinistro sia in comune con un estremo già esistente o che questo avvenga all'estremo destro. Si noti che, mentre è importante la distinzione tra estremo sinistro e destro nell'ntervallo che si inserisce, tale distinzione è ininfluente per gli estremi degli intervalli già inseriti.



Dalla figura si osserva che: se si inserisce un intervallo il cui estremo sinistro si sovrappone all'estremo sinistro di un altro intervallo, i due sono annidati (a); equivalentemente con intervalli destri (b). Se la sovrapposizione avviene tra intervallo sinistro ed uno destro, i due intervalli sono contigui (c)(d).

Per fare in maniera tale che l'intero spazio di indirizzamento risulti partizionato, quando si crea il trie per l'aggiornamento, al nodo radice viene associato come hop il valore di hop_ϵ riferito ad ϵ. Vengono generati entrambi i figli della radice, che fanno riferimento ai prefissi 0* e 1*, e marcati come prefissi; come valore di hop ereditano il valore della radice.

In realtà questa operazione equivale a partizionare lo spazio di IP nei due intervalli $[0,2^{16}-1]$ e $[2^{16},2^{32}-1]$.

Per l'inserimento di una coppia (P, hop_P) tale che $P \neq \epsilon$ si procede come indicato dai seguenti passi:

1. Si costruisce nel trie il percorso che porta fino al nodo N riferito a P e si marca il nodo N come prefisso. A tutti i nuovi nodi che sono stati eventualmente creati, ad eccezione di N, viene assegnato come hop, il valore del loro nodo padre.

2. A partire da P si calcola il valore $P_h + 1$ dove P_h è come in 2.7.1. Si percorre il cammino suggerito dai bit di $P_h + 1$, a partire da quello piú significativo, finché possibile. Si supponga di essersi fermati al nodo M a profondità k. Sei il (k+1)-esimo bit di $P_h + 1$ vale 0 si aggiunge ad M un figlio sinistro, se vale 1 un figlio destro. Per costruzione questo figlio deve essere una foglia, altrimenti sarebbe stato possibile andare avanti nel cammino. Al campo hop del nodo cosí creato si dà il valore dello stesso campo di M.

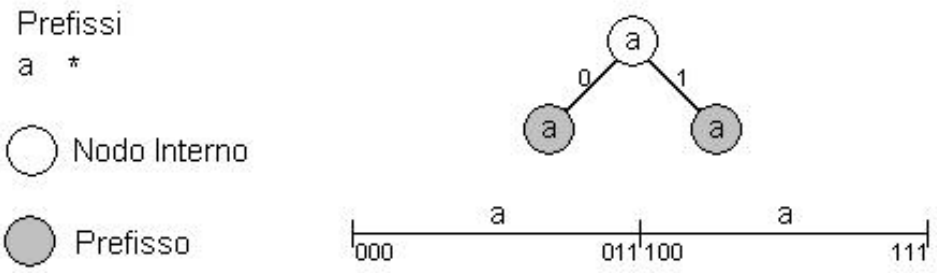
3. A partire da N si effettua una visita in profondità sostituendo il valore hop di ogni nodo con hop_P, se si incontra un nodo marcato come prefisso si ignora l'intero sottoalbero che ha questo come radice.

4. Sia A il nodo marcato come prefisso piú diretto predecessore di N. Seguendo il cammino tra A, compreso, ed N, escluso, si cerca, se esiste, il primo nodo M per cui valga la seguente proprietà: si percorra a partire da M il cammino ottenuto scegliendo sempre il figlio sinistro, il nodo, I, a partire dal quale non è piú possibile andare avanti non è una foglia. Si aggiunge ad I un figlio sinistro che erediti da questo il valore di hop.

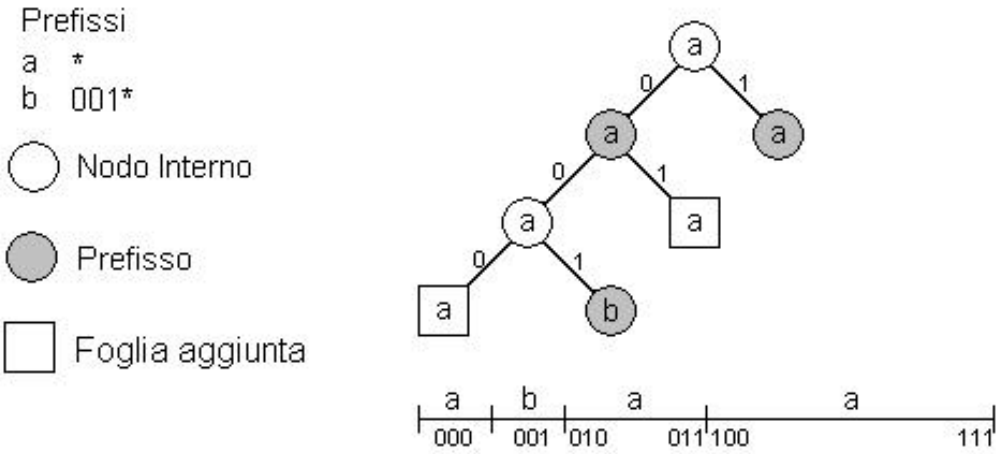
Il passo 1 viene effettuato per inserire l'estremo sinistro dell'intervallo generato da P, il passo 2 inserisce l'estremo sinistro dell'intervallo successivo, il passo 3 assegna il valore appropriato a tutti i frammenti in cui si potrebbe essere diviso l'intervallo. Il passo 4 potrebbe non essere necessario, infatti, viene eseguito soltanto se l'intervallo inserito non aveva l'estremo sinistro condiviso con un altro intervallo.

L'inserimento cosí come descritto non è molto efficiente, in fase di implementazione, però, sono possibili diverse ottimizzazioni che verranno discusse in dettaglio nel capitolo 8.

Esempio 4.3.1. *In figura si può osservare il trie nello stato iniziale. Lo spazio di indirizzamento viene partizionato in due intervalli riferiti agli indirizzi 0* e 1*. I nodi in grigio sono quelli marcati come prefisso, il valore all'interno dei nodi rappresenta il valore del next hop associato. Cosí inizializzato, il trie garantisce la presenza di un predecessore per ogni inserzione e il fatto che lo spazio di indirizzamento sia completamente coperto.*



Nella figura seguente si vede come viene modificato il trie dopo l'inserimento del prefisso 001. I nodi circolari aggiunti vengono creati dal passo 1 I nodi quadrati sono quelli che vengono creati dai passi 2 e 4.*



In entrambe le figure si vede come viene partizionato lo spazio di indirizzamento riferito all'intervallo [000, 111].

4.3.2 La procedura di cancellazione

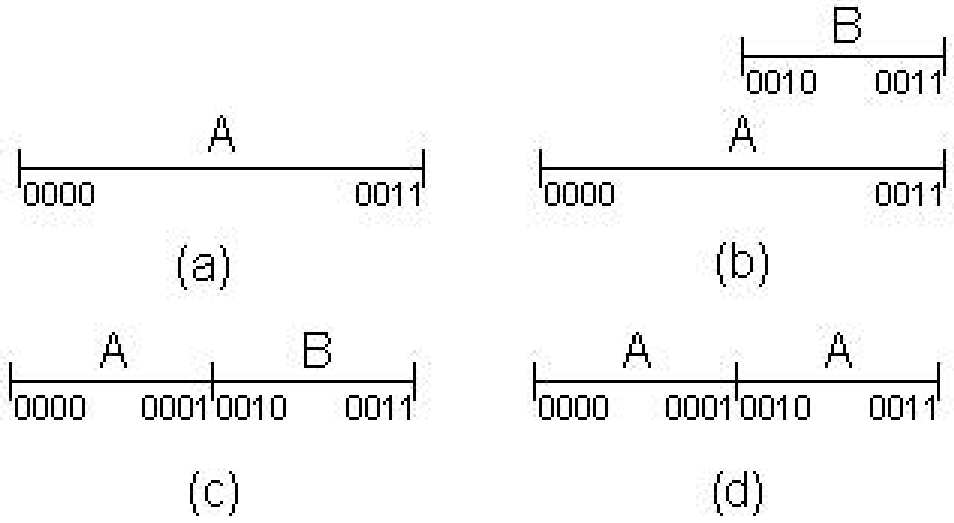
La procedura di cancellazione è molto semplice, si supponga di voler cancellare la coppia (P, hop), il primo passo prevede la ricerca del nodo riferito a P all'interno del trie, se questo non viene trovato o non è marcato come prefisso l'algoritmo termina con un fallimento.

Sia R tale nodo, e sia H il valore del next hop del nodo padre di R, si effettua una visita in profondità dell'albero avente R come radice sostituendo ogni riferimento ad hop con uno ad H, se durante la visita si incontra un nodo N marcato come prefisso, l'abero che ha come radice N non viene visitato ed eventuali riferimenti ad hop al suo interno non vengono modificati.

Il nodo R, a questo punto, non sarà piú marcato come prefisso.

Questo tipo di procedura ha il difetto di introdurre un fenomeno di frammentazione degli intervalli, infatti, anche se i riferimenti a hop vengono sostituiti con quelli ad H, la struttura dell'albero non viene modificata e conseguentemente non cambia il numero degli intervallo ma solo il loro contenuto.

Esempio 4.3.2. *Si supponga di avere indirizzi IP di quattro bit.*



Si immagini di avere inizialmente la coppia $(00, A)$*, essa dà luogo all intervallo* $I = [0000, 0011]$ *che fa riferimento ad A (a), Si inserisca adesso la coppia* $(001, B)$*, essa dà luogo ad un intervallo annidato al primo* $[0010, 0011]$ *riferito a B (b). Se si partiziona lo spazio contenuto nell'intervallo* $[0000, 0011]$ *in maniera tale da non avere intervalli sovrapposti si ottengono i due intervalli* $I_1 = [0000, 0001]$ *riferito ad A e* $I_2 = [0010, 0011]$ *riferito a B (c). Si decida adesso di eliminare la coppia* $(001, B)$*, quello che si vuole ottenere è di ritornare alla situazione originale in cui è presente il solo intervallo I interamente riferito ad A (a). Quello che si ottiene invece e di avere ancora 2 intervalli* $I_1 = [0000, 0001]$ *e* $I_2 = [0010, 0011]$ *questa volta entrambi riferiti ad A (d).*

Si notano immediatamente i seguenti fatti:

- Nella generazione del vettore V_{RLE} I è equivalente a $I_1 \cup I_2$ per il lemma 2.7.2
- Gli intervalli I_1 e I_2 sono generati dai nodi figli del nodo da cui deriva I.

La prima osservazione ci garantisce che la frammentazione non introduce errori nella procedura di cancellazione, la seconda ci rivela un criterio per eliminare il fenomeno della frammentazione.

Lemma 4.3.1. *Se entrambi i figli di un nodo N sono foglie e non sono marcati come prefissi e se riferiscono allo stesso hop di N possono essere eliminati.*

4.4 Divisione in front end e back end

Per la costruzione della tabella delle decisioni di entrambi gli algoritmi presentati in questo capitolo, si parte dalla tabella di routing, F, espansa in maniera tale che tutti i prefissi siano lunghi 32 bit. Come già noto, questa tabella non può essere costruita esplicitamente a causa delle enormi dimensioni che assumerebbe.

Al posto di F, in questa tesi si propone di utilizzare il vettore V_{RLE}, di cui abbiamo già visto alcune proprietà, che contiene la compressione RLE della tabella F.

Grazie all'introduzione del vettore V_{RLE}, l'algoritmo di costruzione delle tabelle di decisione possono essere divisi, in due parti:

- Un front end, comune ad entrambi gli algoritmi, che a partire dalla tabella di routing di input T costruiscono il vettore V_{RLE}
- Un back end per ognuno dei due algoritmi di costruzione della tabella delle decisioni.

4.4.1 Vantaggi della divisione in front end e back end

I vantaggi della divisione in front end e back end sono molteplici: in primo luogo lo stesso front end viene utilizzato sia per l'algoritmo CDG lookup che per SCDG lookup. Nel seguito vengono discussi e confrontati due differenti algoritmi per realizzare il front end.

La dimensione di V_{RLE} è molto inferiore di quella della tabella F, facendo si che V_{RLE} possa essere costruito esplicitamente. Una stima delle dimensioni del vettore V_{RLE} viene fatta nel paragrafo 9.2.

Si è già osservato che per effettuare l'aggiornamento della tabella è necessario ricostruirla per intero. Questo comporterebbe la riesecuzione sia del front end che del back end; questo viene evitato perché V_{RLE} può facilmente essere aggiornato grazie all'impiego di un trie ausiliario.

In questo modo la procedura di aggiornamento non ha piú il compito di ricostruire la tabella, ma semplicemente quello di aggiornare V_{RLE} e chiamare il back end appropriato.

Il vettore V_{RLE} è stato introdotto per separare il front end ed il back end ed ha, prendendo in prestito la terminologia dei compilatori, le stesse funzioni ricoperte dal codice intermedio.

Ogni sequenza contigua di elementi di V_{RLE} la cui somma dei run length è uguale a 2^h rappresenta la compressione RLE di un cluster. Per la precisione la i-esima sequenza rappresenta il cluster T_i. Al contrario di come discusso in 4.1.1, i cluster sono già in forma compressa.

Capitolo 5

Il front end

Nel capitolo precedente abbiamo introdotto la divisione in front end e back end dell'algoritmo per la costruzione delle strutture CDG e SCDG. Il front end viene utilizzato indifferentemente sia per l'una che per l'altra struttura.

In questa fase viene preso in input un file contenente una tabella di routing nel formato specificato nella definizione 2.1.1 e restituito il vettore V_{RLE}.

Nel paragrafo 5.1 viene discussa la realizzazione del front end con il metodo delle quadruple, nel paragrafo 5.2 si propone un nuovo metodo per la realizzazione del front end che ha due importanti vantaggi:

- ha sempre prestazioni migliori in termini di tempo di esecuzione
- al contrario del metodo di costruzione tramite quadruple, realizza contemporaneamente a V_{RLE} la struttura necessaria all'algoritmo di update.

Il capitolo si conclude con il confronto tra le prestazioni dei due metodi mettendo in evidenza i vantaggi del metodo proposto in questa tesi rispetto all'impiego delle quadruple.

5.1 Front end realizzato tramite quadruple

Sia T una tabella di routing composta da coppie del tipo (P, h) dove P è un prefisso e h il next hop associato a questo. Si definisce *netmask* il numero di bit significativi in P e si indica con $|P|$.

Sia $(P, h) \in T$ una generica coppia appartenente alla tabella di routing T ad eccezione di quella in cui $P = \epsilon$. La coppia $(\epsilon, h_\epsilon) \in T$ si chiama *default route.* Sia m il numero di bit da cui è composto un indirizzo IP; per ogni coppia (P, h), siano P_l e P_h i valori definiti come nel lemma 2.7.1 Si costruiscano le due seguenti quadruple:

$< a = P_l, b = |P|, c = 0, d = h >$

$< a = P_h, b = m - |P|, c = 1, d = h >$

Se $|P| = m$, la costruzione delle quadruple è leggermente differente, invece di costruire le quadruple appena descritte si generano le seguenti:

$< a = P_l, b = m - |P|, c = 0, d = h >$

$< a = P_h, b = |P|, c = 1, d = h >$

Questa particolarità è necessaria per garantire la correttezza dell'ordinamento delle quadruple che sarà descritta in seguito.

Sia S una lista composta dalle quadruple create a partire da tutte le coppie appartenenti alla tabella di routing T. Si ordinino lessicogrraficamente, adesso, gli elementi di S ricordando che $S_i =< a_i, b_i, c_i, d_i >$ precede $S_j =< a_j, b_j, c_j, d_j >$ se: $a_i < a_j$ oppure $a_i = a_j \wedge b_i < b_j$ oppure $a_i = a_j \wedge b_i = b_j \wedge c_i < c_j$ oppure $a_i = a_j \wedge b_i = b_j \wedge c_i = c_j \wedge d_i < d_j$.

Si osservi che $a_i = a_j \wedge b_i = b_j \wedge c_i = c_j$ se e soltanto se S_i è stato generato da una coppia (P_i, h_i), S_j è stato generato da una coppia (P_j, h_j) e $(P_i, h_i) \sim (P_j, h_j)$ ovvero $P_i = P_j$ e $|P_i| = |P_j|$.

In questo caso ad una stessa coppia corrispondono piú next hop e ne verrà considerato soltanto uno.

Per inserire le quadruple inerenti alla coppia $(\epsilon, h_\epsilon) \in T$ si procede nel modo seguente:

- si inserisce in testa ad S la quadrupla $< 0, 0, 0, h_\epsilon >$
- si inserisce in coda ad S la quadrupla $< 2^m - 1, m, 1, h_\epsilon >$

Può capitare che esista già una quadrupla con $a = 0$ oppure $a = 2^m - 1$. L'ordinamento dell'insieme S fa un maniera tale che le quadruple appena inserite vengano ignorate se ve ne era già presente una piú significativa.

Esempio 5.1.1. *Si riprende la tabella di routing definita nel capitolo 4 a pagina 41. Nella tabella seguente viene riportato l'insieme S delle quadruple ad essa associate. Si ricorda che in questo esempio m vale 4.*

a	*b*	*c*	*d*
0000	*0*	*0*	*B*
0000	*2*	*0*	*A*
0010	*3*	*0*	*D*
0011	*1*	*1*	*D*
0011	*2*	*1*	*A*
0110	*0*	*0*	*C*
0110	*4*	*1*	*C*
1000	*2*	*0*	*C*
1000	*3*	*0*	*A*
1001	*1*	*1*	*A*
1011	*2*	*1*	*C*
1110	*0*	*0*	*C*
1110	*4*	*1*	*C*
1111	*4*	*1*	*B*

Si osserva che ogni quadrupla segna il limite di un intervallo sullo spazio di indirizzamento. Piú precisamente le quadruple in cui c è uguale a zero fanno riferimento ad un estremo sinistro e quelle in cui c vale uno ad uno destro.

Per costruzione ci saranno tante quadruple in cui c vale zero quanto quelle in cui c vale uno.

Gli intervalli associati a T non coprono l'intero spazio di indirizzamento. E' necessario che, invece, gli intervalli siano una partizione dello spazio degli indirizzi IP.

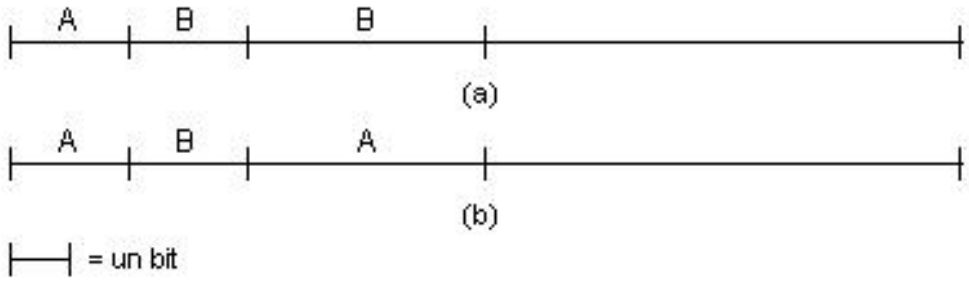
Si può osservare una corrispondenza fra le quadruple e lo spazio di indirizzamento. Piú precisamente sia $S_i = < a_i, b_i, c_i, d_i >$ una qualunque quadrupla e $S_{i+1} =$

$< a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1} >$ la quadrupla successiva, S_i definisce sullo spazio di indirizzamento l'intervallo $[a_i, a_{i+1}]$ con associato il valore d_i. Questo può determinare delle anomalie nel caso di intervalli annidati.

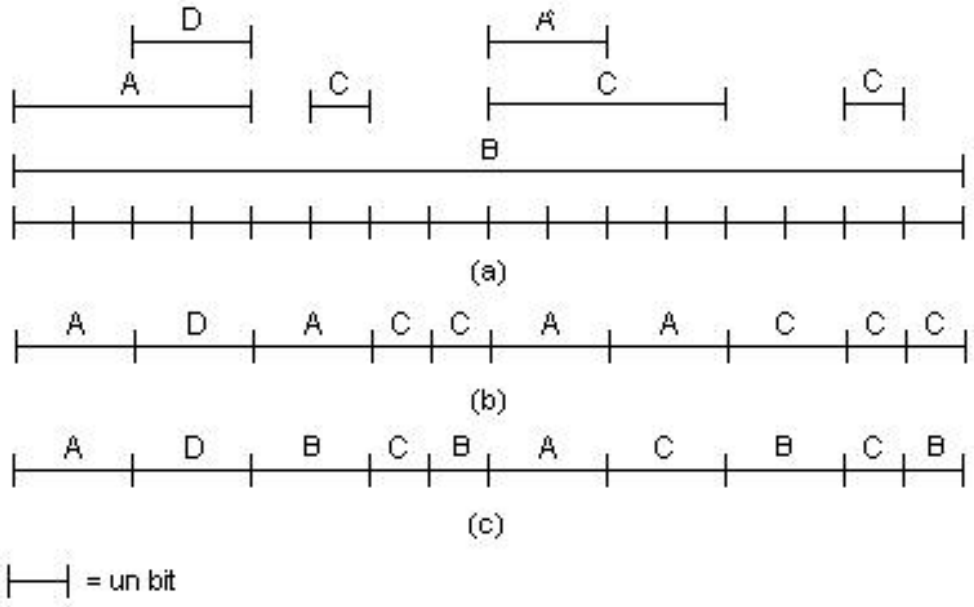
Esempio 5.1.2. *Si supponga di avere indirizzi lunghi 4 bit e di avere la seguente tabella di routing* $T = [(0*, A), (001*, B)]$. *Da questa si ottengono le quattro quadruple mostrate nella tabella seguente:*

a	*b*	*c*	*d*
0000	*1*	*0*	*A*
0010	*3*	*0*	*B*
0011	*1*	*1*	*B*
0111	*3*	*1*	*A*

Nella figura (a) si osserva l'interpretazione delle quadruple a livello di spazio di indirizzamento. Nella figura (b) si vede il risultato che si voleva ottenere.



Esempio 5.1.3. *In figura (a) sono mostrati gli intervalli relativi alla tabella del capitolo 4 a pagina 41. In figura (b) viene mostrata la proiezione dell'insieme S visto nell'esempio precedente. Si nota immediatamente che non c'è corrispondenza tra gli intervalli in (a) e le proiezioni in (b). Il risultato che si vuole ottenere viene mostrato in figura (c).*



Per passare dallo stato della figura (b), a quello desiderato della figura (c), si aggiungono delle quadruple seguendo l'algoritmo che verrà esposto tra breve.

Sia Q una pila inizializzata con il valore h_ϵ, si indica con Q_{top} il valore in testa alla pila.

$\forall i$ tale che $1 \leq i < |S|$ sia S_{i+1} la quadrupla successiva ad S_i, se $c_i = 1 \wedge a_{i+1} - a_i > 1$ e Q non è vuota, si inseriscono tra S_i e S_{i+1} le quadruple $< a_i + 1, 0, 0, Q_{top} >$ e $< a_{i+1} - 1, m, 1, Q_{top} >$. Se $c_{i+1} = 0$ si inserisce nella pila il valore d_{i+1}, altrimenti si elimina da Q il valore in testa.

Esempio 5.1.4. *La lista S delle quadruple definite sopra si trasforma in quella nella tabella seguente, le quadruple aggiunte tramite l'algoritmo appena descritto vengono riportate in grassetto.*

a	b	c	d
0000	*0*	*0*	*B*
0000	*2*	*0*	*A*

0010	3	0	D
0011	1	1	D
0011	2	1	A
0100	**0**	**0**	**B**
0101	**4**	**1**	**B**
0110	0	0	C
0110	4	1	C
0111	**0**	**0**	**B**
0111	**4**	**1**	**B**
1000	2	0	C
1000	3	0	A
1001	1	1	A
1010	**0**	**0**	**C**
1010	**4**	**1**	**C**
1011	2	1	C
1100	**0**	**0**	**B**
1100	**4**	**1**	**B**
1110	0	0	C
1110	4	1	C
1111	4	1	B

Sia $|S|$ il numero complessivo di quadruple. $\forall i$ tale che $1 \leq i < |S|$ si calcola il valore l_i associato alla quadrupla $S_i =< a_i, b_i, c_i, d_i >$, che rappresenta la lunghezza dell'intervallo generato a partire da S_i, nel seguente modo: sia $S_{i+1} =<$

$a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1} >$ la quadrupla successiva ad S_i, $l_i = a_{i+1} - a_i + 1$. Se $c_i = c_{i+1}$ decremento l_i di uno, se $c_i = 1$ e $c_{i+1} = 0$ decremento l_i di due, altrimenti lo lascio invariato.

Si osservi che in questo modo per l'ultima quadrupla, ovvero $S_{|S|}$ non è stato generato nessun valore di l, si definisce $l_{|S|}$ associato a $S_{|S|}$ tale che $l_{|S|} = 2^m - a_{|S|}$.

La condizione $c_i = c_{i+1}$ è equivalente a dire che le quadruple S_i ed S_{i+1} fanno riferimento a due estremi dello stesso tipo ed è quindi necessario escludere un estremo, la condizione $c_i = 1$ e $c_{i+1} = 0$ rappresenta il caso in cui tra S_i ed S_{i+1} si trovi un intervallo e quindi bisogna escludere entrambi gli estremi.

Dopo aver calcolato tutti gli l_i si procede nel seguente modo: $\forall i$ tale che $1 \leq i < |S|$ sia S_{i+1} la quadrupla successiva ad S_i, sia l_{i+1} il valore associato ad S_{i+1} ed l_i il valore associato ad S_i. Allora se $l_{i+1} < 0$ si decrementa l_i del valore l_{i+1} e si assegna 0 ad l_{i+1}.

E' possibile ora costruire un vettore V_{RLE} lungo al piú $|S|$ che contenga la compressione RLE dell'intero spazio di indirizzamento.

$\forall i$ tale che $1 \leq i \leq |S|$, si costruisce una coppia (l_i, d_i) in cui d_i è il valore presente nella quadrupla S_i ed l_i è il valore associato ad essa.

Esempio 5.1.5. *Le coppie generate dalle quadruple dell'esempio precedente sono:*

(0, B); (2, A); (2, D); (0, D); (0, A); (2, B); (0, B); (1, C); (0, C); (1, B); (0, B); (0, C); (2, A); (0, A); (1, C); (1, C); (0, C); (2, B); (0, B); (1, C); (0, C); (1, B)

Il j-esimo elemento di V_{RLE} contiene la j-esima coppia per cui vale $l_j \neq 0$.

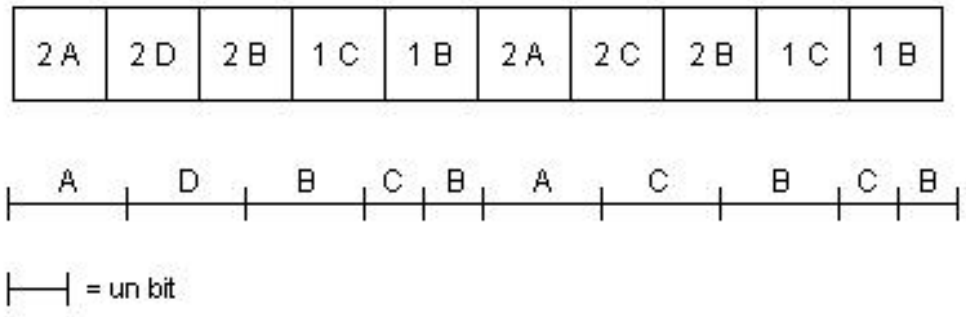
Esempio 5.1.6. *Si riportano le coppie da cui verrà creato il vettore V_{RLE}*

(2, A); (2, D); (2, B); (1, C); (1, B); (2, A); (1, C); (1, C); (2, B); (1, C); (1, B)

Una stima delle dimensioni del vettore V_{RLE} viene fornita nel paragrafo 9.2.

Per costruzione si ha che $\sum_{i=1}^{|S|} l_i = 2^m$, questo è equivalente ad affermare che gli intervalli generano una partizione dello spazio di indirizzamento.

Esempio 5.1.7. *In figura si vede il vettore V_{RLE} riferito agli esempi che abbiamo discusso in questo paragrafo. Viene riportata in figura anche una proiezione degli intervalli sullo spazio di indirizzamento.*



5.2 Front end realizzato tramite trie

In questa tesi viene proposto un metodo alternativo per la realizzazione del front end che ha due importanti vantaggi:

- ha prestazioni migliori rispetto alla realizzazione con quadruple
- realizza il trie necessario per l'algoritmo di update durante la costruzione di V_{RLE}.

La differenza di prestazioni fra i due metodi si è dimostrata sempre abbastanza rilevante, questo aspetto viene esasperato se si considera il fatto che dopo aver creato il vettore V_{RLE}, nel caso del front end realizzato con quadruple, si deve ancora realizzare il trie necessario alla procedura di aggiornamento della tabella delle decisioni.

5.2.1 Inizializzazione del trie

Sia $(\epsilon, h_\epsilon) \in T$ la coppia che genera gli intervalli della *default route*. L'inizializzazione del trie è molto semplice: come primo passo viene creato il nodo radice, ad esso viene assegnato il valore h_ϵ. Vengono creati sia il figlio sinistro che il destro della radice, anche ad essi viene associato il valore h_ϵ. Inoltre i due figli della radice vengono marcati come prefissi. Questa inizializzazione ha il significato intuitivo di creare i due intervalli $[0,\ 2^{m-1}-1]$ e $[2^{m-1}, 2^m-1]$ ai quali viene associato il valore h_ϵ.

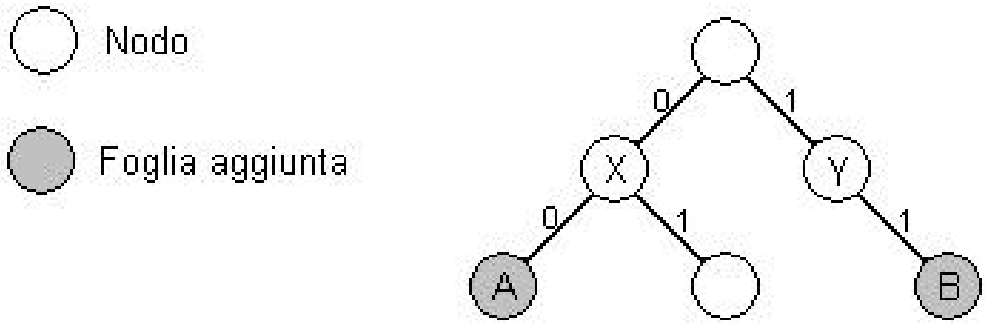
5.2.2 Inserimento dei prefissi

L'inizializzazione del trie descritta nel paragrafo precedente fa sì che questo abbia diverse proprietà importanti che si desidera vengano costantemente mantenute:

- lo spazio degli indirizzi IP risulta partizionato
- il trie mantiene gli intervalli associati alla coppia (ϵ, h_ϵ)
- per ogni nuovo nodo che viene marcato come prefisso, nel trie esisteva già un predecessore di questo marcato come prefisso
- ad ogni foglia del trie corrisponde l'estremo sinistro di un intervallo a cui è associato come next hop il valore dell'etichetta di tale foglia.

L'inserimento di un nuovo prefisso, e quindi di un nuovo cammino nel trie, potrebbe rendere falsa l'ultima proprietà. Infatti, salvo il caso in cui esiste già il cammino tra la radice ed il nodo a cui è associato il prefisso, sarà necessario aggiungere dei nuovi nodi al trie. L'inserimento di un nuovo nodo in un trie potrebbe trasformare una foglia in nodo interno.

Esempio 5.2.1. *Nella figura seguente viene mostrato un trie al quale sono state aggiunti i due nodi marcati in grigio; come si può notare il nodo X era interno già prima dell'inserimento del nodo A. Il nodo Y, invece, era un nodo foglia; l'inserimento di B lo fa diventare interno.*



Il trie viene costruito incrementalmente inserendo ad uno ad uno i prefissi della tabella T. Per l'inserimento di ogni prefisso viene utilizzata la procedura descritta nel paragrafo 4.3.1 la quale ha due importanti caratteristiche:

- mantiene sempre vere le proprietà sopra citate
- l'ordine con cui vengono inseriti i prefissi nel trie non cambia il risultato finale

La seconda caratteristica è importante dal punto di vista delle prestazioni, infatti, grazie ad essa non è necessario effettuare un ordinamento come nel caso delle quadruple, dove bisognava ordinare un insieme di $2|T|$ elementi.

5.2.3 Costruzione del vettore V_{RLE}

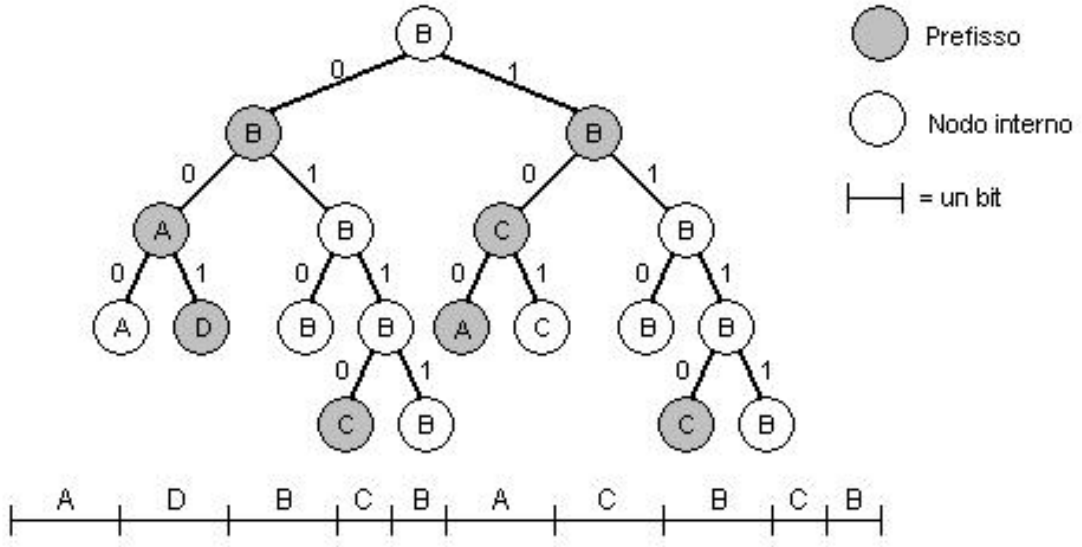
Ad ogni nodo N del trie, è possibile associare una stringa ottenuta tramite la concatenazione delle etichette riferite agli archi che è stato necessario attraversare per il suo raggiungimento a partire dalla radice. Tale stringa contiene la rappresentazione in binario del prefisso associato al nodo.

Per costruzione le foglie del trie rappresentano l'estremo sinistro di un intervallo.

Sia $|F|$ il numero di foglie del trie e sia F_i tale che $i \in [1, |F|]$ il prefisso associato alla i-esima foglia. Se la i-esima foglia si trova a profondità h_i sia F'_i il valore ottenuto concatenando m-h_i zeri a F_i.

La visita in profondità del trie garantisce che $\forall i, j \in [1, |F|]$ tale che $i < j$, $F'_i < F'_j$.

Esempio 5.2.2. *In figura è mostrato il trie ottenuto dopo aver inserito tutti i prefissi della tabella del capitolo 4 e la rappresentazione degli intervalli da esso generati.*



Dalle considerazioni appena fatte si ricava il seguente algoritmo per la costruzione del vettore V_{RLE}.

- Si effettua una visita in profondità del trie: per ogni foglia i tale che $1 \leq i < |F|$ si costruisce la coppia (l_i, d_i) in cui $l_i = F'_{i+1} - F'_i$ e d_i e il valore di hop assegnato alla i-esima foglia.

- Si aggiunge alle coppie create al punto precedente la coppia $(2^m - F'_{|F|},\ d_{|F|})$ dove $d_{|F|}$ è il valore di hop associato all'ultima foglia visitata.

Esempio 5.2.3. *Il risultato dell'algoritmo appena esposto per il trie dell'esempio precedente è mostrato nell'esempio 5.1.6.*

Il passaggio dalle coppie cosí costruite al vettore V_{RLE} è ovvio: la i-esima coppia costituisce l'i-esimo elemento di tale vettore.

5.3 Confronto delle prestazioni

Si procede adesso con un confronto tra le prestazioni delle due implementazioni del front end. In entrambe le realizzazioni viene presa in input una tabella di routing T a partire dalla quale viene costruito il vettore V_{RLE} ed il trie necessario per le operazioni di aggiornamento. Considerando il fatto che entrambe le realizzazioni del front end partendo dallo stesso input producono il medesimo output, il raffronto delle prestazioni viene fatto in base alla complessità computazionale ed al tempo medio di esecuzione.

5.3.1 Complessità della realizzazione con quadruple

La realizzazione del front end mediante l'impiego di quadruple può essere cosí riassunta:

1. Costruzione dell'insieme S delle quadruple
2. Sort delle quadruple
3. Scansione di S per l'inserimento delle quadruple supplementari
4. Scansione di S, come modificato nel punto precedente, per la costruzione del vettore V_{RLE}
5. Costruzione del trie necessario per l'update

Data una tabella di routing T contenente $|T|$ coppie, la costruzione dell'insieme delle quadruple S, viene realizzata in tempo $O(|T|)$ in quanto la generazione delle due quadruple riferite ad ogni coppia richiede tempo costante.

Il passo 2 della realizzazione con quadruple prevede l'ordinamento di $2|T|$ elementi. L'impiego del radix sort garantisce il completamento di questo passo in tempo $O(|T|)$.

Il passo 3 consiste nella scansione dei $2|T|$ elementi di S e nell'inserimento di un certo numero di nuove quadruple, dipendenti dalla particolare tabella di routing presa in input. Sia $|S|$ il numero di quadruple contenute in S dopo aver eseguito il passo 3, vale che $|S| \geq |2T|$ ed il numero di nuove quadruple inserite è $|S| - 2|T|$. Dal momento che, sia la scansione di S che l'inserimento di una nuova quadrupla vengono effettuate in tempo costante, il terzo passo del front end viene eseguito in tempo $O(|S|)$.

Il passo 4 prevede un'ulterore scansione del vettore S, cosí come modificato nel passo precedente, per creare il vettore V_{RLE}. Tale scansione viene effettuata in tempo $O(|S|)$.

I primi quattro passi del front end servono per realizzare il vettore V_{RLE}, l'ultimo passo è necessario alla per la creazione del trie impiegato dalla procedura di aggiornamento.

La complessità dell'ultimo passo viene discussa nel seguito del paragrafo perché esso corrisponde al primo passo del front end realizzato mediante trie.

La complessità del front end appena descritto per ottenere la generazione del vettore V_{RLE} è data dalla somma dello complessità dei quattro passi appena descritti, ovvero: $O(|T|) + O(|T|) + O(|S|) + O(|S|) = O(|S|)$.

5.3.2 Complessità della realizzazione con trie

La realizzazione del front end mediante trie può essere come di seguito schematizzata:

1. Creazione del trie mediante sequenza di inserimenti dei prefissi
2. Visita del trie per la costruzione del vettore V_{RLE}

La creazione del trie viene realizzata tramite una sequenza di invocazioni della chiamata alla funzione di inserimento descritta nel paragrafo 8.2.

Tale funzione, ripetuta per ogni rotta della tabella di routing T, esegue i quattro passi di seguito descritti:

1. Inserimento del prefisso
2. Inserimento dell'estremo destro
3. Inserimento dell'estremo sinistro
4. Visita del trie avente il prefisso come radice

I primi 3 passi consistono in una ricerca all'interno del trie. Sia m la lunghezza in bit degli indirizzi IP, la complessità di questi passi avviene nel caso pessimo in tempo $O(m)$.

L'ultimo passo prevede la visita del trie che abbia come radice il nodo N a cui corrisponde il prefisso che si sta inserendo. Se il prefisso è composto da $|P|$ bit tale nodo si trova a profondità $|P|$. La complessità della visita vale $O(2^{m-|P|})$ che è un valore costante.

L'intera procedura di inserimento viene quindi eseguita in tempo costante. Dovendo ripetere tale procedura $|T|$ volte, il primo passo del front end impiega quindi tempo $O(|T|)$.

Il secondo passo consiste nella visita in profondità del trie dalla quale si ottiene il vettore V_{RLE}. Tale trie ha $|S|$ foglie e profondità massima m. Una maggiorazione grossolana del numero di nodi complessivi del trie è $m|S|$ ammettendo che nessuno degli $|S|$ percorsi tra la radice ed una foglia abbia nodi in comune con un altro percorso. La complessità di questo passo è quindi $O(|S|)$.

La complessità dell'intero front end realizzata tramite trie è quindi dell'ordine di $O(|S|)$.

5.3.3 Confronto tra i front end

La complessità della realizzazione con quadruple del front end non varia neanche con l'aggiunta del passo di creazione del trie necessario alla procedura di update della tabella delle decisioni, in quanto questo viene effettuato in tempo $O(|T|)$.

Dal punto di vista della complessità computazionale entrambe le realizzazioni impiegano tempo lineare rispetto al numero di intervalli in cui partizionano lo spazio degli indirizzi IP.

Nel confronto dei tempi di esecuzione, si vede che la realizzazione con trie è piú efficiente di quella con quadruple. E' possibile scegliere di confrontare i tempi per la sola costruzione del vettore V_{RLE}, oppure aggiungere anche quelli per creare la struttura per l'update.

La realizzazione con trie ha prestazioni migliori in entrambi i casi: Ovviamente il divario di prestazioni aumenta se si aggiungono i tempi richiesti per la creazione della struttura per l'update, in quanto la realizzazione con trie del front end non necessita di ulteriori elaborazioni per la creazione di questa struttura.

Nel paragrafo 10.6 viene presentato il raffronto tra le previsioni dei due front end. In questa sede ci limitiamo a dire che nel caso in cui si richieda che anche la versione con quadruple del front end realizzi il trie necessario alle operazioni di

aggiornamento della tabella delle decisioni, la realizzazione mediante trie risulta essere il 37.35% piú veloce.

Il confronto tra le due realizzazioni del front end è a favore della versione che utilizza il trie anche nel caso in cui si decida di non fare costruire la struttura necessaria all'aggiornamento alla realizzazione con quadruple, infatti, in questo caso, questa risulta comunque piú lenta del 9.06%.

Capitolo 6

Il back end

Nel capitolo precedente sono state discusse due implementazioni per la realizzazione del front end, la prima realizzata tramite l'impiego di quadruple, la seconda, proposta in questa tesi, realizzata mediante l'impiego di un trie.

Il front end prende in input una tabella di routing T e restituisce il vettore V_{RLE} contenente la compressione RLE della tabella di routing F espansa con tutti i prefissi di lunghezza 32.

Viene discussa adesso la realizzazione del back end del CDG lookup e dell'SCDG lookup. In entrambi i casi viene preso in input il vettore V_{RLE} generato tramite il front end e vengono restituite: la tabella delle decisioni, il vettore delle colonne ed il vettore delle righe.

Dopo aver generato queste strutture viene eseguita una procedura, comune ad entrambi i back end, volta a verificare, ed eventualmente costruire il vettore dei bucket introdotto nel paragrafo 4.1.3.2.

L'esecuzione del back end deve essere realizzata in maniera quanto piú efficiente possibile in quanto essa influenza anche le prestazioni della procedura di aggiornamento. Infatti quest'ultima dopo aver effettuato l'aggiornamento del vettore V_{RLE} chiama il back end appropriato per la ricostruzione della tabella delle decisioni.

6.1 Il back end del CDG lookup

Si è già osservato nel paragrafo 4.1 che le dimensioni della tabella delle decisioni dipendono da un parametro k.

6.1.1 Costruzione dei cluster

Fissato k, il back end, a partire dal vettore V_{RLE}, costruisce 2^k cluster.

Viene creato un vettore C, chiamato *vettore dei cluster*, di 2^k elementi ognuno dei quali conterrà un cluster. Per ragioni di efficienza nell'implementazione il vettore C è un vettore di puntatori a delle strutture che contengono le informazioni del cluster.

Per costruire i cluster si esegue l'algoritmo seguente:

- $i, j = 0$
- a partire dalla (x=j)-esima coppia (l_x, d_x) contenuta nel vettore V_{RLE} se $\sum_{y=j}^{x} l_y \leq 2^k$ si aggiunge la coppia (l_x, d_x) all'i-esimo cluster e si incrementa x di 1
- se $\sum_{y=j}^{x} l_y = 2^k$ si aggiunge all'i-esimo cluster la coppia (l_x, d_x) ed a j si assegna il valore di $x+1$, si incrementa i di 1 e si inizia un nuovo cluster
- se $\sum_{y=j}^{x} l_y < 2^k$, ma l'inserimento della coppia (l_{x+1}, d_{x+1}) farebbe sì che $\sum_{y=j}^{x+1} l_y > 2^k$, si inserisce nel cluster la coppia $(2^k - \sum_{y=j}^{x} l_y, d_{x+1})$ e si modifica il valore di l_{x+1} in modo che valga $(\sum_{y=j}^{x+1} l_y) - 2^k$. Si assegna a j il valore x in maniera tale che la (x+1)-esima coppia venga riesaminata, si incrementa i di 1 e si inizia un nuovo cluster
- L'algoritmo termina quando $i = 2^k$ ovvero quando sono state esaminate tutte le coppie contenute nel vettore V_{RLE}.

6.1.2 Eliminazione dei cluster consecutivi ridondanti

Una volta creati i 2^k cluster C_i, si vogliono eliminare tutti i cluster ridondanti, ovvero tutti i cluster C_j tali che $\forall i, j \in [0, 2^k - 1]$ $C_i = C_j$ e $j > i$.

Per far ciò si costruisce il vettore *sort* di 2^k elementi in cui in posizione i si trova l'indice con cui accedere al vettore C per ottenere l'i-esimo cluster.

Ovviamente il vettore sort viene inizializzato nel modo seguente: $\forall i \in [0, 2^k - 1]$ $sort[i] = i$.

Una prima importante osservazione è che la costruzione esplicita di tutti i cluster è abbastanza onerosa dal punto di vista dell'impiego di memoria. Si osserva anche che l'inserimento di un prefisso P tale che $|P| < k$ dà luogo a cluster consecutivi uguali.

L'esecuzione dell'algoritmo descritto nel paragrafo precedente viene modificata nel seguente modo:

- Si crea il primo cluster
- Sia a la posizione nel vettore C dell'ultimo cluster inserito, inizialmente $a = 0$
- dopo aver creato il cluster C_i, con $i > a$ come spiegato nel'algoritmo del paragrafo precedente, se $C_i = C_a$ si elimina C_i e si assegna *sort[i]* $= a$, se $C_i \neq C_a$, si assegna ad a il valore i

In questa maniera il numero di elementi del vettore C realmente allocati risulta essere considerevolmente piú basso.

6.1.3 Ordinamento dei cluster

Si vuole ottenere una struttura identica a C in cui i cluster siano ordinati lessicograficamente. Dal punto di vista implementativo, C è un vettore di puntatori ai

cluster. Si crea un vettore γ di puntatori ai puntatori contenuti in C. In questo modo il vettore γ contiene soltanto i riferimenti ai cluster e non è necessario effettuarne una copia. Tramite la funzione di libreria *qsort* () vengono ordinati i puntatori di γ in maniera tale che questo contenga i riferimenti ai cluster in ordine lessicografico.

6.1.4 Rimozione dei cluster ridondanti

Sia $|C|$ il numero di cluster realmente inseriti in C. Si crea il vettore *subs* contenente $|C|$ elementi ed inizializzato nel seguente modo $\forall i \in [0, |C| - 1]$, $subs[i] = i$. Tale vettore conterrà le sostituzioni da effettuare nel vettore *sort* affinché esso mantenga l'ordine dei cluster dopo la rimozione di quelli ridondanti.

L'algoritmo di eliminazione dei cluster è il seguente:

- $i = 1, j = 0$
- se il cluster C_p riferito da γ_i è uguale al cluster C_q riferito da γ_j si cancella da C il cluster C_p, si tiene traccia della sostituzione tramite l'assegnamento $subs[p] = q$ e si incrementa i di 1.
- se i due cluster sono differenti si assegna a j il valore di i e si incrementa questo di 1.
- L'algoritmo termina quando sono stati eliminati tutti i cluster ridondanti, ovvero quando $i = |C|$.

A questo punto dal vettore C sono stati eliminati tutti i cluster ridondanti. E' adesso necessario tener traccia dei cambiamenti nel vettore *sort*. Per far questo si esegue la procedura seguente:

- $\forall j \in [0, |C| - 1]$ e $\forall i \in [0, 2^k - 1]$ se $sort[i] = j$ allora $sort[i] = subs[j]$

Si osserva immediatamente che la complessità di questa procedura è $0(2^k|C|)$. Nell'implementazione reale si è riusciti a scrivere una procedura che esegue le stesse operazioni ed avente complessità $0(2^k)$.

6.1.5 Costruzione della metrica

Il vettore M contenente le lunghezze dei run date dalla compressione RLE delle righe della tabella delle decisioni si costruisce tramite il seguente algoritmo:

- Si crea un vettore B di 2^k bit inizializzati a 0
- $\forall i \in [0, |C| - 1]$ sia C_i l'i-esimo cluster, sia n_i il numero di coppie in esso contenute
- $\forall j \in [1, n_i]$ sia $b_j = \sum_{q=1}^{j} l_q$ si settano ad 1 i bit nelle posizioni $b_j - 1$

Costruito il vettore B, da esso si ricava M nel modo seguente:

- $|M| = \sum_{i=0}^{2^k-1} B_i$
- Sia i tale che $B[i] = 1$ e $\forall j \in [0, i-1]$ $b[j] = 0$ allora $M[0] = i + 1$
- $\forall j \in [i+1, |M| - 1]$ se $B[j] = 1$ sia $m = \sum_{a=0}^{j} B_a$ e $q < j$ tale che $B[q] = 1$ e $\nexists p$ tale che $q < p < j$ per cui valga $B[p] = 1$, allora $M[m-1] = j - q$.

La costruzione del vettore delle colonne a partire dalla metrica viene descritta nel paragrafo 6.3.

6.1.6 Costruzione della tabella delle decisioni

La tabella delle decisioni viene allocata in un unico vettore W di α_k x β_k elementi dove $\alpha_k = |C|$ e $\beta_k = |M|$.

Per ogni cluster viene generata una riga R della tabella delle decisioni per mezzo del seguente algoritmo:

- Si inizializza $j = 0$
- $\forall i \in [0, \beta_k - 1]$ se per la coppia $(l_j, d_j) \in V_{RLE}$ vale $l_j > M[i]$, si decrementa l_j del valore $M[i]$ e si assegna $R[i] = d_j$.
- se, invece, vale $l_j = M[i]$ si assegna $R[i] = d_j$ e si incrementa j di 1.
- per costruzione del vettore M non può mai esistere la condizione $l_j < M[i]$.

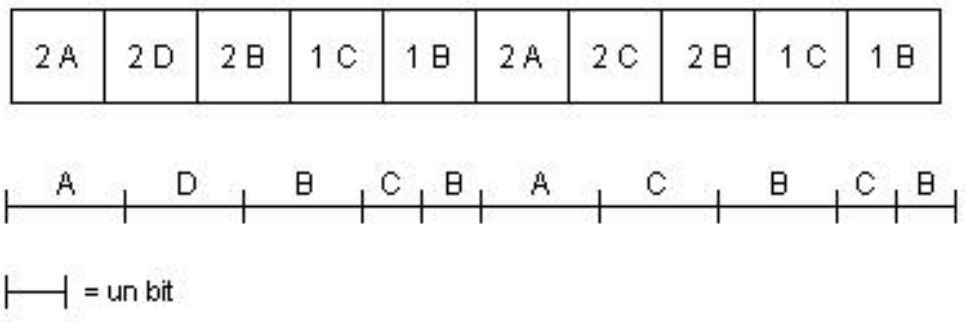
Il primo elemento della riga corrispondente al cluster C_i viene memorizzato in W a partire dall'offset $i * \beta_k$.

A questo punto non resta da fare altro che costruire il vettore delle righe, *row*. Questo viene realizzato per ragioni di efficienza tramite un vettore di puntatori. La realizzazione è molto semplice, si procede nel seguente modo:

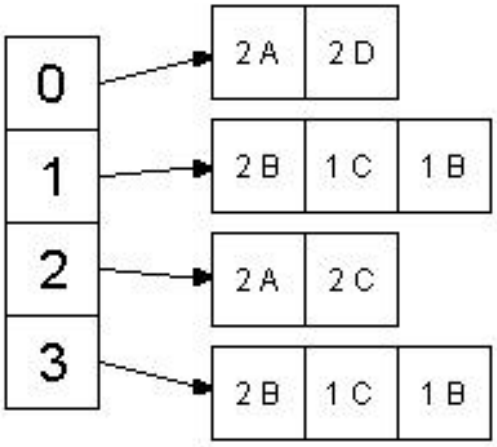
- $\forall i \in [0, 2^k - 1]$ $row[i] = W + (sort[i] * \beta_k)$

6.1.7 Esempio di CDG back end

Si supponga di aver costruito il vettore V_{RLE} associato alla tabella dell'esempio del capitolo 4. Il vettore V_{RLE} è come mostrato in figura.

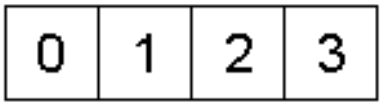


Il primo passo consiste nel creare, a partire dal vettore V_{RLE} i cluster C_i ed il vettore C. Nella figura seguente vengono mostrati i quattro cluster in cui è stato diviso il vettore V_{RLE} del nostro esempio ed il vettore C contenente i riferimenti ad essi.



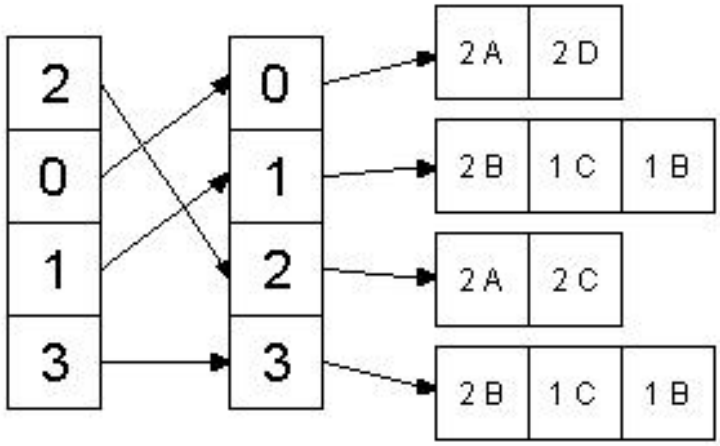
Contestualmente alla loro creazione viene controllato se cluster consecutivi sono uguali, in caso affermativo si elimina una copia. Nell'esempio si nota che questa cosa non si verifica. In realtà si è già discusso del fatto che questa, invece, sia una situazione molto frequente dovuta all'inserimento di prefissi piú corti di k bit.

Assieme al vettore C viene creato il vettore *sort*, inizializzato come in figura

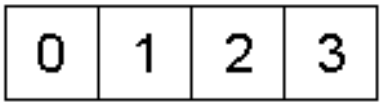


Si è appena detto che in questo esempio non ci sono cluster consecutivi ridondanti, quindi dopo questa verifica il vettore *sort* non subisce modifiche.

A questo punto viene creato il vettore γ che contiene i puntatori al vettore C in maniera da poter accedere ai cluster in ordine lessicografico. Nella figura seguente viene mostrato il vettore γ

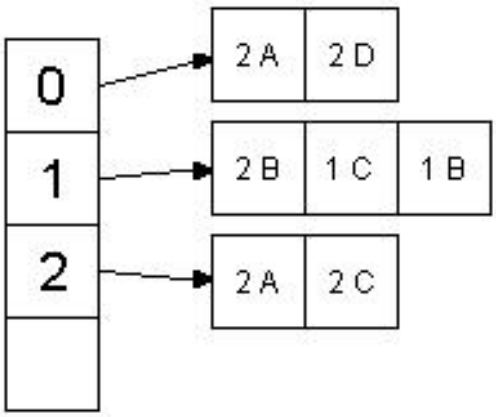


Prima di eliminare i cluster ridondanti viene creato il vettore delle sostituzioni *subs* inizializzato come in figura

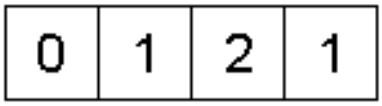


Si nota immediatamente che il fatto di non aver trovato cluster consecutivi ridondanti fa in maniera tale che il vettore subs sia identico al vettore *sort*. Questo è dovuto alle ridotte dimensioni di questo esempio, nei casi reali, non solo questi due vettori sono differenti, inoltre subs risulta essere piú piccolo di un ordine di grandezza.

Nella figura seguente viene mostrato il vettore C dopo aver eliminato il cluster ridondante, si osservi che il puntatore al cluster eliminato non contiene piú alcun riferimento.

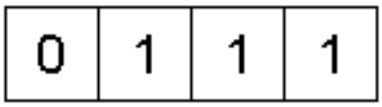


Il vettore *sort* viene modificato in maniera da tener traccia delle modifiche apportate a C e si trasforma nel seguente modo.



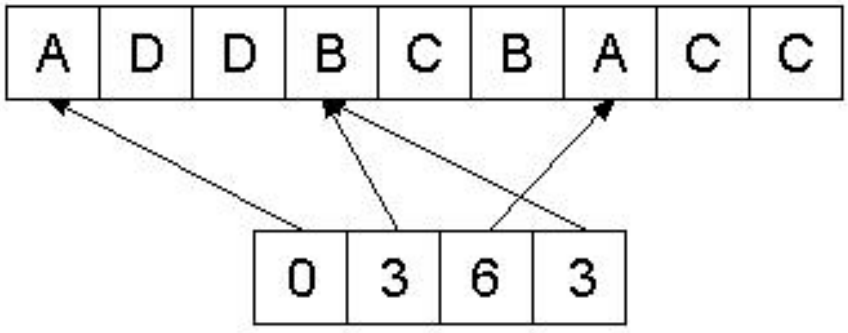
A questo punto viene creato il vettore M contenente la metrica. Per costruire M si realizza un vettore di bit B di 2^{m-k} elementi inizializzati a 0. Dalla scansione del primo cluster si ha che il secondo ed il quarto bit vanno settati ad 1, dalla scansione del secondo cluster si ha che il secondo, il terzo ed il quarto bit vanno settati ad 1. Infine la scansione dell'ultimo cluster richiede che il secondo e l'ultimo bit siano settati ad 1. Ovviamente non è necessario verificare il valore di un bit prima di modificarne il valore. L'operazione di impostare il valore di un bit ad 1 equivale a "marcare" una posizione del vettore B.

In figura viene mostrato il vettore B.



A partire dal vettore B si ottiene la metrica mostrata nella figura seguente.

Infine è possibile costruire il vettore delle righe e la tabella delle decisioni. Quest'ultima viene memorizzata in un vettore unico per ragioni di efficienza. Nella figura seguente vengono mostrate queste due strutture.



6.2 Il back end dell'SCDG lookup

La struttura SCDG è una variante di quella CDG che si ripromette di costruire una tabella delle decisioni di dimensioni ridotte. Alcuni dei passi dell'algoritmo di costruzione di tale struttura sono molto simili a quelli descritti nei paragrafi precedenti. Per questa ragione non viene descritta l'intera procedura nel dettaglio, ma vengono mostrate soltanto le differenze ed i passi supplementari per la realizzazione della tabella delle decisioni.

6.2.1 Realizzazione delle tabelle split

Sia m la lunghezza degli indirizzi IP ed $h = m - k$ dove k è il valore scelto per la costante K_SIZE.

A livello logico il primo passo per la realizzazione della struttura SCDG consiste nel dividere in due la tabella delle decisioni CDG in maniera tale che la prima parte contenga le colonne accedute dai primi 2^{h-1} elementi del vettore delle colonne e la seconda parte contiene le colonne riferite dai restanti 2^{h-1} elementi. Il vettore delle righe viene sdoppiato e si fa in maniera tale che la copia punti alle righe della seconda metà della tabella delle decisioni. Il vettore delle colonne viene spezzato esattamente a metà. In questo modo si sono ottenute due strutture CDG dove K_SIZE vale k ed $h = m - k - 1$. La prima struttura si chiama *sinistra* e la seconda *destra.* In seguito ad ogni oggetto riferito alla struttura sinistra viene associato il pedice s, equivalentemente ad ogni oggetto riferito alla struttura destra viene associato il pedice d.

Per ottenere queste due strutture si procede in modo seguente: si creano due vettori dei cluster C_s e C_d equivalenti a C. Si generano i cluster cosí come mostrato per il CDG lookup, questa volta si vuole che per il cluster C_i valga che $\sum_j l_j = 2^{k-1}$. Se $i = 2n$ tale che $n \in \mathbb{N}$, C_i diventa il cluster $C_{s,n}$ se invece se $i = 2n + 1$ il cluster C_i diviene $C_{d,n}$.

Osservazione 6.2.1. *Dopo aver riunito la tabella sinistra e destra non si ottiene una tabella come se si fosse scelto K_SIZE pari a $k + 1$. Infatti in quest'ultimo caso, i cluster sarebbero stati ordinati nel seguente modo:*

- *Sia $|C|$ il numero totale di cluster,* $\forall i \in [0, \frac{|C|}{2} - 1]$ $C_i = C_{s,i}$
- $\forall i \in [\frac{|C|}{2}, |C|]$ $C_i = C_{d,i-\frac{|C|}{2}}$

E' importante notare che con questo ordinamento dei cluster, la compressione RLE delle righe delle tabelle sinistra e destra, e quindi della tabella generata dalla loro unione, sarebbe differente rispetto all'ordinamento scelto per l'SCDG.

Ottenuti C_s e C_d le tabelle sinistra e destra si ottengono seguendo per ognuno dei due vettori i passi del CDG lookup descritti in precedenza.

6.2.2 Accoppiamento delle colonne

Sia $|M_s|$ il numero di colonne della tabella sinistra ed $|M_d|$ il numero di quelle nella tabella destra. $|M_s|$ ed $|M_d|$ sono anche le dimensioni dei vettori della metrica associati alle tabelle. Se $|M_s| < |M_d|$ si aggiungono $|M_d| - |M_s|$ colonne contenenti il carattere speciale *don't care* alla tabella sinistra, equivalentemente se $|M_s| > |M_d|$ si aggiungono $|M_s| - |M_d|$ colonne alla tabella destra.

In questo modo entrambe le tabelle avranno $\beta_k = \max(|M_s|, |M_d|)$ colonne. Siano S_s ed S_d insiemi tali che $S_{s,i}$ sia la i-esima colonna della tabella sinistra e $S_{d,i}$ sia la i-esima colonna della tabella destra.

Si costruisce un grafo bipartito $G = (O \cup D, A)$ dove $O = S_s$, $D = S_d$ e V è un insieme di cardinalità β_k^2 in cui $\forall i, j \in [1, \beta_k]$ $V_{i,j}$ rappresenta il vertice che collega il nodo O_i con D_j. Al vertice $V_{i,j}$ viene associato un peso dato dal risultato della funzione $f(i, j)$.

Nel corso della sperimentazione si è cercata la funzione $f(i, j)$ che renda minori le dimensioni della tabella SCDG. Nel paragrafo 4.1.2.2 sono state spiegate le difficoltà connesse a questo problema, in questa sezione ci limitiamo a discutere i dettagli implementativi.

Il grafo G viene rappresentato da una matrice quadrata di ordine β_k in cui l'elemento nella riga i e colonna j vale $f(i, j)$.

Tramite questa matrice viene calcolato l'accoppiamento di costo minimo. Tale accoppiamento può essere visto come un problema di flusso di costo minimo seguendo le trasformazioni descritte in [16].

Sia $ord()$ l'ordinamento tale che se il nodo O_i è stato accoppiato con D_j allora $ord(i) = j$. Si permutano le colonne S_d in maniera tale che $S_{d,i} = S_{d,ord(i)}$.

Si noti che, avendo sperimentalmente osservato che, l'ordinamento con il quale si ottiene una struttura SCDG di dimensioni minori è quello per cui vale $ord(i) = i$ quanto descritto in questo paragrafo non ha effetto sulla creazione della struttura SCDG e può essere ignorato.

6.2.3 Riunione delle tabelle

A partire dalle tabelle destra e sinistra si vuole ottenere un'unica tabella. Si procede nel modo seguente: $\forall i \in [1, \beta_k]$ si crea S_i come la concatenazione di $S_{d,i}$ in coda ad $S_{s,i}$.

Si ottiene cosí una tabella S dove la i-esima colonna è S_i. Alla tabella destra era associato un vettore delle righe row_d come alla sinistra era associato row_s. Si crea il vettore row come la concatenazione di row_d a row_s.

Nella realizzazione pratica gli elementi, sia della tabella sinistra che della destra, sono memorizzati per righe; row_s e row_d sono dei vettori di puntatori a tali righe, per questa ragione quanto appena descritto si implementa semplicemente ridimensionando row_s e copiando i puntatori di row_d in coda a quelli di row_s.

6.2.4 Eliminazione delle righe ridondanti

Dopo aver riunito la tabella sinistra e quella destra si ottiene una versione provvisoria della tabella delle decisioni in cui è probabile vi siano delle righe uguali. Siano R_i ed R_j due righe della tabella delle decisioni, se $i < j$ e $R_i = R_j$, la riga R_j viene eliminata e tutti i riferimenti ad essa nel vettore delle righe vengono modificati in maniera tale da far riferimento ad R_i.

Non viene descritto in dettaglio questo algoritmo perché simile a quello descritto nei paragrafi 6.1.3 e 6.1.4.

6.2.5 Compattamento della struttura

Anche nel caso della struttura SCDG per l'intera tabella delle decisioni viene utilizzato un unico vettore W in cui memorizzare l'intera struttura. Se, dopo l'eliminazione delle righe ridondanti, la tabella delle decisioni contiene α_k righe, il vettore W è formato da $\alpha_k * \beta_k$ elementi. Le righe della tabella delle decisioni vengono copiate consecutivamente dentro W in maniera che il primo elemento della i-esima riga sia memorizzato a partire dalla posizione $(i-1) * \beta_k$.

Naturalmente anche i puntatori contenuti nel vettore delle righe devono essere modificati in maniera tale che i puntatori riferiti alla riga i puntino all'indirizzo $W + ((i-1) * \beta_k)$.

6.2.6 Shuffle del vettore delle righe

A causa della divisione dei cluster nei vettori C_s e C_d accade che i riferimenti nel vettore delle righe cosí come è stato realizzato non risultino in ordine. Infatti tale divisione fa in maniera tale che gli indirizzi il cui (k+1)-esimo bit vale 0 appartengono ai cluster memorizzati in C_s, gli indirizzi in cui lo stesso bit vale 1 sono memorizzati in C_d. In questo modo il bit $k+1$ dell'indirizzo x di cui si effettua il lookup dovrebbe essere utilizzato per scegliere a quale metà del vettore accedere con i primi k bit di x.

Esistono due metodi per risolvere questo problema:

- L'indice con cui accedere al vettore delle righe vale $first(k, x)$ se il bit $k+1$ di x vale 0 oppure $first(k, x) + 2^{k+1}$ se vale 1

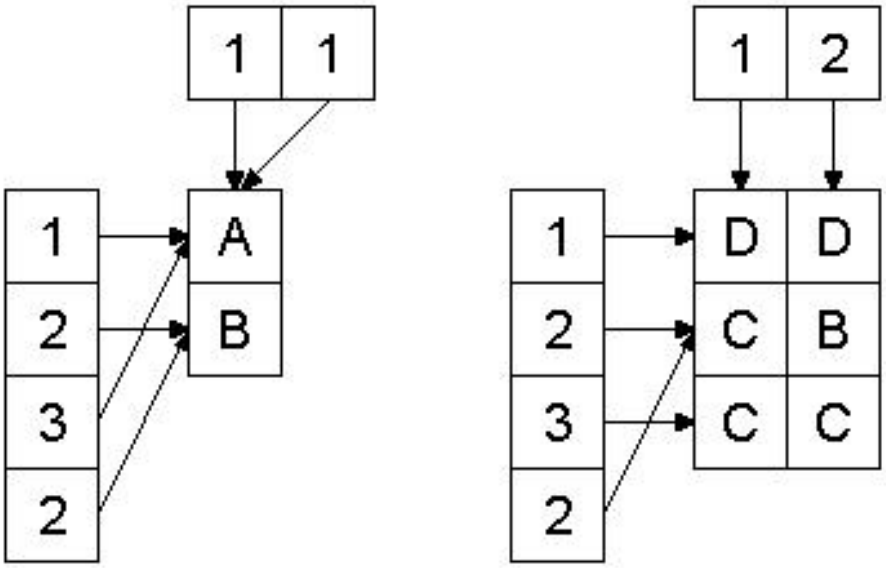
- Si effettua uno *shuffle*[11] degli elementi del vettore delle righe.

Per ragioni di efficienza in fase di lookup si preferisce la seconda soluzione. Lo shuffle viene effettuato nel seguente modo:

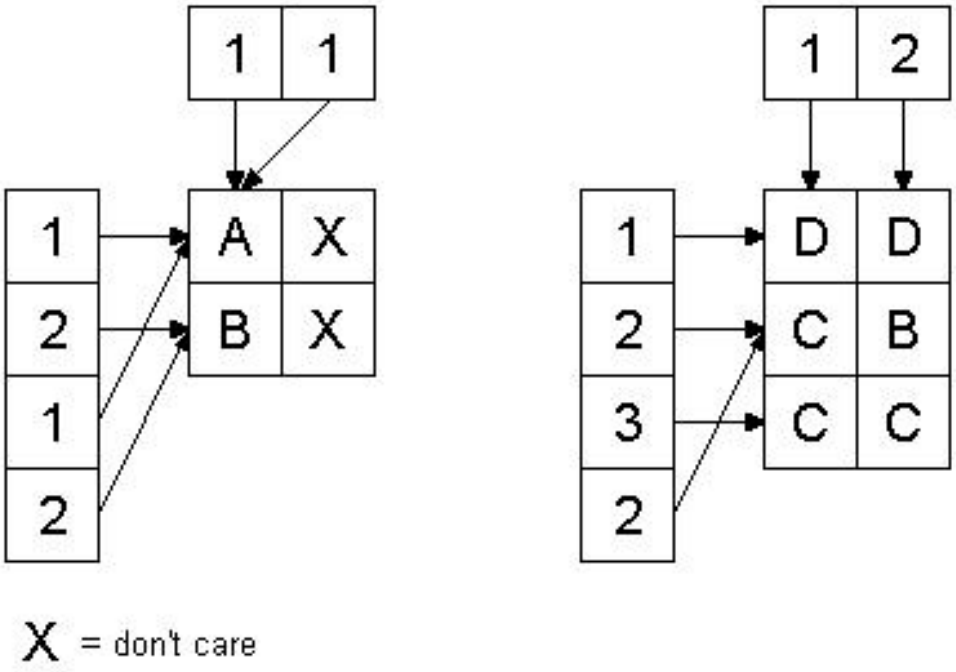
- $\forall i \in [0, 2^k - 1]\ row[2i] = row[i]$
- $\forall i \in [2^k, 2^{k+1} - 1]\ row[2(i - 2^k) + 1] = row[i]$

6.2.7 Esempio di SCDG back end

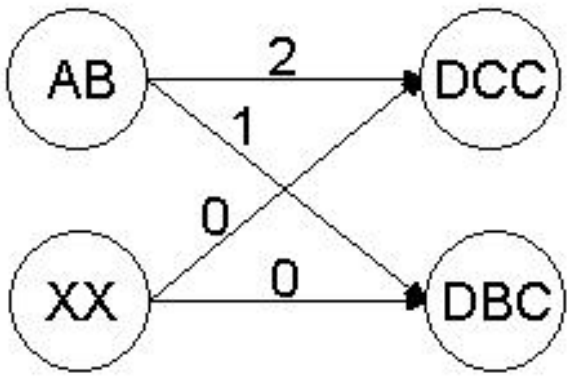
Si presenta in questo paragrafo un esempio di costruzione della struttura SCDG a partire dalla tabella dell'esempio del capitolo 4. Anche in questo caso il back end prende in input il vettore V_{RLE}. A partire da questo vengono create le tabelle *sinistra* e *destra*. Per la loro creazione si procede eseguendo l'algoritmo di creazione della struttura CDG a partire dai vettori dei cluster C_s e C_d. Si omettono in questa sede i dettagli perché simili a quelli descritti nella sezione 6.1. Nella figura seguente vengono mostrate le tabelle sinistra e destra. In questo esempio $m = 4$, $k = 2$ e $h = 2$. Si osservi che le tabelle in figura sono due strutture CDG in cui $k = 2$, ma $h = 1$.



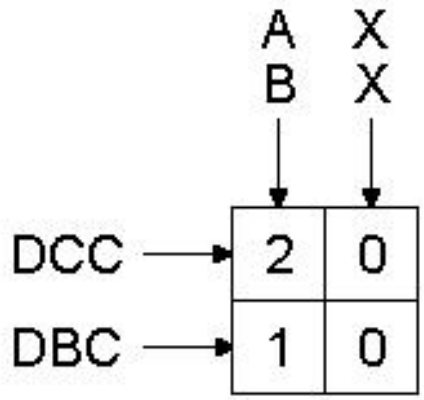
Dalla situazione appena mostrata si aggiungono alla tabella con meno colonne, la sinistra, tante colonne in maniera tale da far sí che esse risultino in numero pari all'altra. Le colonne inserite contengono al loro interno il valore speciale *don't care*. Si osservino le seguenti cose: non esistono riferimenti alle colonne aggiunte alle quali quindi non è possibile accedere, inoltre il numero di colonne ora ottenuto equivale al valore di β_k.



A partire dai set di colonne si crea il grafo bipartito mostrato in figura. In questo esempio si è scelto come funzione per assegnare un peso agli archi il numero di *missmatch* rilevati fra gli elementi nella stessa posizione delle stringhe associate alle colonne. Si osservi anche che il valore *don't care* ha un match con qualunque valore.

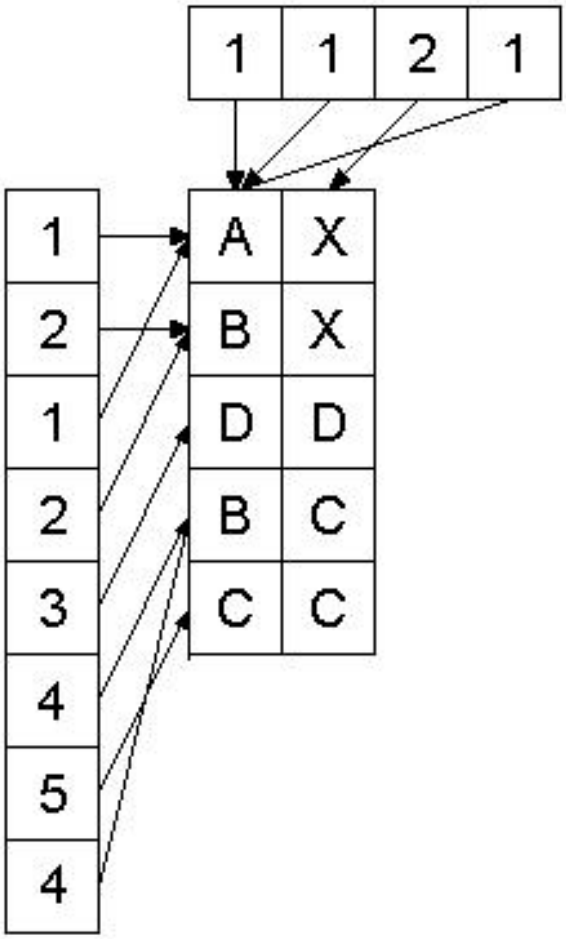


Il grafo mostrato in figura viene memorizzato in una matrice quadrata di ordine β_k, in questo caso 2. L'elemento nella riga i e colonna j contiene il valore del peso associato all'arco che va dal nodo O_i al nodo D_j.

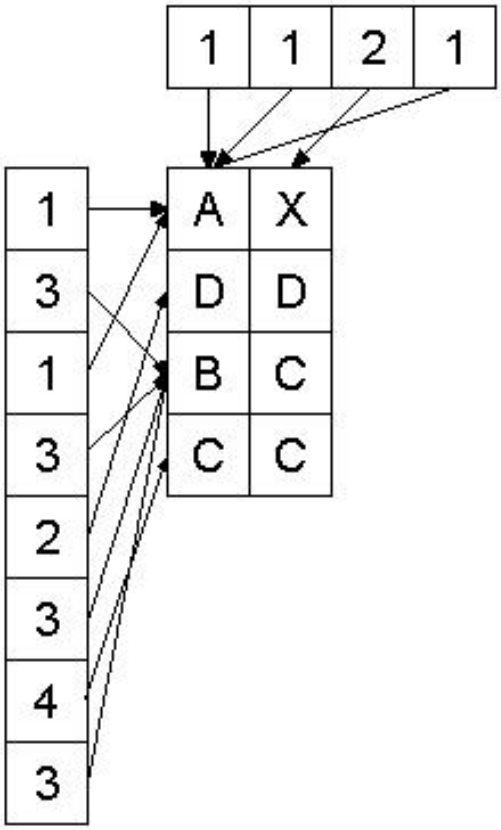


Gli archi da cui si ottiene l'accoppiamento di costo minimo sono: $1 \rightarrow 2$ e $2 \rightarrow 1$ da cui si ottiene *ord(1) = 2* e *ord(2) = 1*.

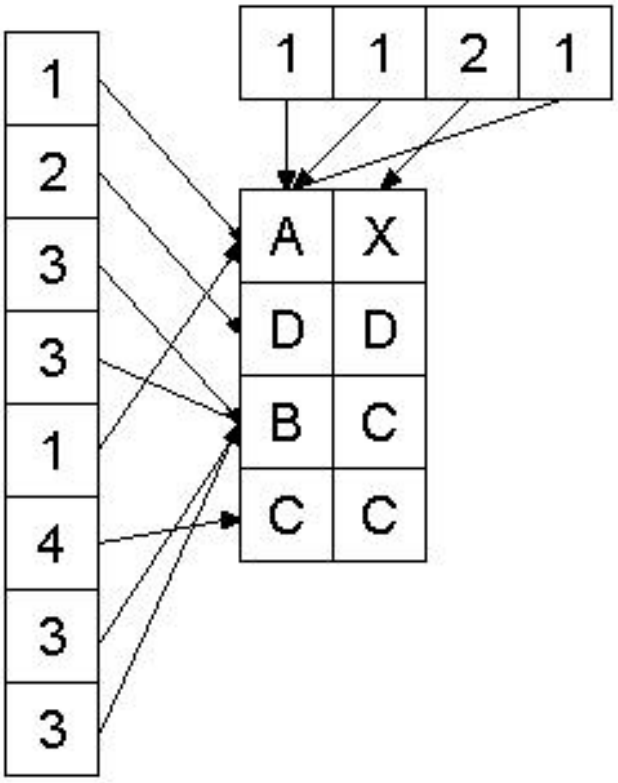
Nella figura seguente si mostra la tabella delle decisioni dopo aver riunito le tabelle sinistra e destra. Come si può osservare la seconda e la quarta riga sono uguali a meno degli elementi contenenti il valore *don't care*.



Nella figura seguente si vede la tabella delle decisioni in cui è stata tolta la riga ridondante. Ovviamente la riga da eliminare è quella contenente il valore *don't care*.



L'ultima immagine mostra la struttura SCDG dopo aver effettuato lo shuffle del vettore delle righe.



6.3 Costruzione del vettore delle colonne

Il vettore delle colonne di lunghezza 2^h dell'SCDG lookup può essere visto come la concatenazione di due vettori delle colonne CDG di lunghezza 2^{h-1}. Per questa ragione è stato scelto di impiegare la stessa procedura per la creazione di entrambi.

Nel caso dell'SCDG, tale procedura viene invocata due volte e le viene passata la prima o la seconda metà del vettore delle colonne. Gli elementi della seconda metà del vettore delle colonne devono essere permutati in maniera da tener traccia dell'ordinamento delle colonne della tabella destra, ovvero, devono essere permutati in maniera tale che tutti i riferimenti ad una generica colonna i si trasformino in riferimenti ad $ord(i)$.

La procedura per la valutazione, ed eventuale creazione, del vettore dei bucket

tratta indifferentemente sia il vettore delle colonne CDG che quello SCDG.

A partire dal vettore M, chiamato *metrica*, contenente le lunghezze dei run della compressione RLE delle righe, il vettore delle colonne viene costruito nel seguente modo:

- Sia j il numero di elementi già inserito nel vettore delle colonne, *col*. Inizialmente j vale 0.
- $\forall i \in [0, |M|)$, si inserisce il valore i per $M[i]$ volte nel vettore delle colonne a partire dalla posizione j. Si incrementa j di $M[i]$.

Come si può immediatamente notare, per costruzione il vettore delle colonne contiene lunghe sequenze di valori ripetuti, ordinati in maniera tale che $\forall i \in [0, 2^h)$ si ha: $|col[i+1] - col[i]| \leq 1$.

Il valore minimo contenuto nel vettore delle colonne è 0, il valore massimo è $|M| - 1$. Si osservi che $|M| = \beta_k$.

Se $|M| < 256$ è possibile utilizzare per memorizzare i valori contenuti nel vettore delle colonne il tipo **unsigned char** al posto del tipo **unsigned short int**. Si osservi che per $h = 16$ non è possibile che sia richiesto un tipo piú grande dell **unsigned short int**, in quanto nel caso pessimo $|M| = 2^{16}$.

Nel caso dell'SCDG al posto del vettore M sono presenti due vettori M_1 ed M_2 riferiti rispettivamente alla prima ed alla seconda metà del vettore delle colonne. In questo caso, la condizione per utilizzare il tipo **unsigned char** è che sia M_1 che M_2 valgano meno di 256.

E' importante notare che per costruzione della tabella SCDG, M_1 ed M_2 valgono circa $M/2$; per questa ragione è spesso possibile usare per le tabelle SCDG il tipo **unsigned char** quando per le tabelle CDG è necessario usare il tipo **unsigned short int**. Nel paragrafo 10.5 viene presentata una statistica sul tipo impiegato

per la memorizzazione degli elementi del vettore delle colonne con riferimento alle tabelle impiegate per la sperimentazione condotta in questa tesi.

6.3.1 Costruzione del vettore dei bucket

Come già detto, il risultato principale presentato in questa tesi e la realizzazione di un vettore delle colonne alternativo, chiamato vettore dei bucket, di dimensioni piú compatte capace di essere memorizzato nella cache L1 e di rendere la procedura di lookup piú performante.

La costruzione del vettore dei bucket non dipende dalla scelta di impiegare il CDG lookup o lo SCDG lookup, essa si basa semplicemente sulla disposizione degli elementi del vettore delle colonne.

Come primo passo viene calcolata la dimensione dei bucket. Per ragioni di efficienza nell'operazione di lookup, tale dimensione deve essere una potenza di due, in maniera tale da poter accedere al vettore dei bucket con un numero intero di bit.

Sia m la media degli elementi del vettore M, per k intero tale che $2^k \leq m < 2^{k+1}$, sia $b = 2^k$, allora si sceglie come dimensione dei bucket il valore b. Un istante di riflessione mostra che se il vettore delle colonne contiene 2^h elementi, allora $m = 2^h/|M|$.

Il numero totale di bucket sarà $|B| = 2^h/b$.

Si partizioni ora il vettore delle colonne costruito nel paragrafo precedente in blocchi B_i di b elementi. Per $i \in [0, |B|)$ ogni B_i è un bucket.

Definizione 6.3.1. *Se tutti gli elementi di un bucket hanno lo stesso valore si dice che questo non contiene un* fault

Il vettore dei bucket può essere visto come due vettori concatenati, chiamati segmento dei riferimenti e segmento dei bucket, il primo contenente $|B|$ elementi ed

il secondo contenente Fb elementi, dove F indica la somma dei bucket contenenti un fault.

Il vettore B conterrà in tutto $|B| + Fb$ elementi.

Si osservi che in realtà non è necessario costruire direttamente il vettore delle colonne per realizzare il vettore dei bucket, tutte le operazioni presentate in questo paragrafo sono realizzabili mediante una trattazione opportuna del vettore M.

Il vettore dei bucket, B,viene costruito nel seguente modo:

- se B_i non contiene fault, allora $B[i]$ contiene il valore di un qualunque elemento di B_i.
- se B_i contiene il j-esimo fault, $B[i] = -(j-1)b$ e B_i viene interamente copiato in B a partire dalla posizione $(|B| + (j-1)b)$.

Il valore negativo serve per distinguere se il contenuto della cella esaminata è il valore contenuto dentro un bucket o se è un riferimento ad un altra porzione di B.

Sarebbe stato preferibile, anche se cambiato di segno, inserire il riferimento assoluto all'offset di B in cui sono stati copiati i bucket contenenti fault. Questo avrebbe evitato di dover sommare il valore $|B|$ in fase di lookup ogni volta che si presenti la necessità di reperire un bucket dal segmento dei bucket. Questa scelta risulta impraticabile perché sarebbe causa di un overflow nei casi sfortunati in cui il vettore B avesse superato i 32Kb. Per ovviare a questo si dovrebbe utilizzare un tipo piú grande dello short int con la conseguente occupazione di una quantità inaccettabile di memoria.

6.3.2 Considerazioni sulla scelta del tipo di vettore delle colonne

Il vettore delle colonne, come si è già detto, contiene $|M|$ elementi nel caso CDG ed $M_1 + M_2$ elementi nel caso SCDG. A seconda del valore di $|M|$, ovvero di $|M_1|$ ed $|M_2|$, è necessario memorizzare gli elementi in un tipo differente. Per ottenere la dimensione in byte del vettore delle colonne bisogna moltiplicare il numero di elementi di tale vettore per la dimensione, in byte, del tipo impiegato per la loro memorizzazione.

Ovviamente, a partire dalla metrica è possibile stabilire le dimensioni in byte del vettore delle colonne prima della costruzione esplicita.

Si è discussa anche la costruzione del vettore dei bucket; in questo caso, come per il vettore delle colonne, è possibile stabilirne la dimensione prima di costruirlo esplicitamente. Il vettore che verrà costruito realmente è quello le cui dimensioni sono minori.

Questa scelta è dettata dal fatto che piú piccolo è il vettore delle colonne, maggiore è la probabilità di trovare i suoi elementi nella cache L1 al momento del lookup. Un numero alto di fault nel vettore dei bucket, però, può portare ad un incremento dei tempi di lookup come discusso nel paragrafo 7.2, per questa ragione nella scelta del tipo di vettore da costruire questo dato può risultare significativo. Nel paragrafo 10.5 viene mostrata una statistica sulle dimensioni del vettore dei bucket e di quello delle colonne in base ai dati in possesso per la sperimentazione di questa tesi.

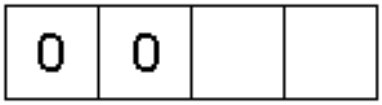
6.3.3 Esempio di costruzione di vettore delle colonne

Si riprende l'esempio del capitolo 4. Si mostra la costruzione del vettore delle colonne e del vettore dei bucket per per la tabella CDG. Si osservi che nel caso dell'SCDG

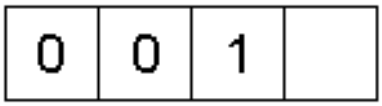
lookup si sarebbe ottenuto un risultato analogo.

Nella figura seguente viene riportato il vettore M contenente le lunghezze dei run della compressione RLE di ogni riga della tabella delle decisioni.

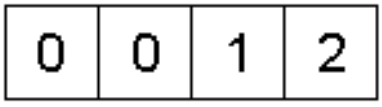
Il primo elemento del vettore M contiene il valore 2, si ricorda che gli indici dei vettori partono da 0. Si inseriscono due zeri in testa al vettore delle colonne.



Il successivo elemento del vettore M vale 1, per questa ragione si inserisce un 1 a partire dalla prima posizione ancora libera del vettore delle colonne.



Per finire, l'ultimo elemento, in posizione 2, vale 1. Si inserisce nella successiva posizione ancora libera del vettore delle colonne il valore 2. Di seguito si vede il vettore delle colonne completo.



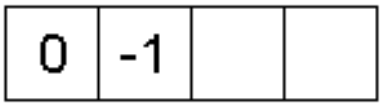
La media degli elementi del vettore M vale 4/3, infatti la somma degli elementi del vettore M vale 4 e |M| vale 3. Come si è osservato nel paragrafo 6.3.1

$\sum_{i=0}^{|M|-1} M_i = 2^h$ quindi la media vale $2^2/3$. Si osservi che $2^1 \leq 4/3 < 2^2$, ragion per cui i bucket conterranno due elementi. Nella figura seguente vengono mostrati i due bucket. Si vede immediatamente che nel primo gli elementi sono tutti uguali, mentre nel secondo gli elementi sono differenti quindi vi è un fault.

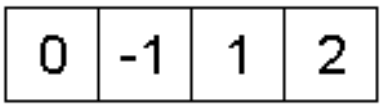
Con i dati del nostro esempio vi sono $|B| = 2$ bucket, la cui dimensione sarà $b = 2$ elementi. Il segmento dei riferimenti contiene $|B|$ elementi. Avendo trovato un solo bucket contenente un fault F vale 1 ed il segmento dei bucket contiene $Fb = 2$ elementi.

Il primo bucket non contiene fault e gli elementi al suo interno valgono 0, al primo elemento del segmento dei riferimenti viene assegnato 0

Il secondo bucket contiene il fault numero 1, il secondo elemento del segmento dei riferimenti contiene il valore 1 cambiato di segno.



A questo punto, il segmento dei bucket si ottiene tramite la concatenazione di tutti i bucket contenenti dei fault. Nel nostro esempio soltanto il secondo bucket. In figura viene mostrato il vettore dei bucket cosí ottenuto.



6.4 Confronto delle prestazioni

Al contrario di quanto avveniva per il front end, partendo dallo stesso vettore V_{RLE} i due back end presentati costruiscono strutture differenti. Per quanto sia importanta la realizzazione efficiente del back end, in quanto influenza sia le prestazioni della costruzione della struttura per il lookup, sia quelle della procedura di update, un confronto tra i tempi dei due back end non ha senso.

Basta un istante di riflessione per rendersi conto che il back end SCDG è intrinsecamente piú lento di quello CDG. La costruzione della tabella destra e di quella sinistra equivalgono all'esecuzione di due istanze del back end CDG. Il fatto di poter costruire le due strutture contemporaneamente ed il fatto di dover trattare cluster contenenti mediamente la metà degli elementi di quelli della struttura CDG fa si che il tempo medio in cui si ottengono la tabella sinistra e quella destra sia leggermente superiore rispetto all'esecuzione del back end CDG.

Il tempo necessario per il completamento del back end SCDG dipende fortemente dall'euristica usata per scegliere l'ordinamento delle colonne della tabella destra prima di unirle con quelle della tabella sinistra. Se si considera che: le colonne delle due tabelle sono β_k quindi il grafo bipartito ha β_k^2 archi e che la funzione per il calcolo del peso impiega come minimo tempo lineare rispetto al numero di righe delle tabelle, la complessità di questo passo vale $O(\min(\alpha_{k,s}, \alpha_{k,d}) * \beta_h^2)$. Dai dati sperimentali si è osservato che la funzione di ordinamento che ha avuto le prestazioni migliori risulta essere la funzione identità. Questo vuol dire che le colonne della tabella destra non vengono permutate prima di essere unite a quelle della tabella sinistra. Per questa ragione la scelta di usare la funzione identità equivale a non costruire il grafo bipartito ma creare direttamente un ordinamento in cui vale che per il generico i-esimo elemento $ord(i) = i$.

6.4.1 Occupazione di memoria

La realizzazione del back end SCDG necessita di raddoppiare le dimensioni del vettore delle righe rispetto all'analoga realizzazione CDG. Questo è dovuto al fatto di dover concatenare il vettore delle righe della tabella destra a quello della tabella sinistra e che entrambi i vettori hanno le stesse dimensioni di quelli realizzati per la struttura CDG. In altre parole, a fronte di un decremento variabile delle dimensioni della tabella delle decisioni SCDG, bisogna aggiungere alle dimensioni totali della struttura un valore costante dovuto al raddoppio degli elementi del vettore delle righe. All'aumento delle righe corrisponde però una diminuzione del numero di colonne. Dai dati sperimentali è visto che nel 75.29% dei casi si possono dimezzare le dimensioni in byte del vettore delle colonne in quanto è possibile usare un tipo piú piccolo per memorizzare ogni elemento. Nel caso di questa sperimentazione in cui si è scelto $K\_SIZE = 16$ e per ragioni di efficienza si vuole che gli elementi del vettore delle righe siano puntatori, al vettore delle righe vanno aggiunti $4 * 2^{16} = 262144$ byte. Il vettore delle colonne nel 75.29% dei casi è formato da elementi di un byte occupando cosí $1 * 2^{16} = 65536$ byte, nel restante 24.71% dei casi necessitano due byte, come per la struttura CDG, per memorizzare ogni elemento occupando $2 * 2^{16} = 131072$ byte. La media della dimensione del vettore delle colonne vale quindi 81729 byte per cui al totale dell'intera struttura vanno sottratti 49343 byte.

Nel caso le tabelle CDG risultino di dimensioni ridotte, il peso della memoria in piú necessaria per la struttura SCDG è molto rilevante. Con la crescita di internet e la conseguente crescita delle dimensioni delle tabelle di routing anche le dimensioni delle tabelle CDG sono destinate ad aumentare facendo si che tale peso tenda a diminuire.

Nel capitolo 10 viene fatto un confronto tra le dimensioni della struttura CDG e quelle della struttura SCDG. La seconda risparmia complessivamente il 12.78% di

spazio. Dal confronto tra la tabella delle decisioni CDG e quella SCDG si vede che la seconda è piú piccola mediamente del 35.45%.

Capitolo 7

La procedura di lookup

Come è stato già spiegato nel capitolo 1, il lookup rappresenta il principale collo di bottiglia nella realizzazione di router in grado di smistare un traffico nell'ordine del Gigabit al secondo. Per questa ragione gli algoritmi per l'IP address lookup devono avere delle procedure di lookup quanto piú efficienti possibile anche a discapito di altri aspetti secondari.

In questo capitolo viene presentata la procedura per il CDG lookup che non necessita di alcuna elaborazione ed utilizza soltanto tre accessi alla struttura dati.

In questa tesi viene presentata anche una procedura per il lookup che, pur richiedendo un minimo di elaborazione, riesce a memorizzare il vettore delle colonne nella cache di primo livello ottenendo un buon incremento delle prestazioni.

Entrambe le procedure di lookup sono tali da poter essere impiegate sia con la struttura CDG che con quella SCDG.

7.1 La procedura di lookup

Nell'implementazione pratica, la tabella delle decisioni viene allocata in un unico segmento di memoria in cui le righe vengono memorizzate consecutivamente. Sia W

il vettore in cui è stata memorizzata la tabella delle decisioni, se ogni riga contiene β_k elementi, l'offset alla i-esima riga non sarà altro che il valore $W + i\beta_k$.

Sia $0 \leq i < 2^k$ e $0 \leq j < \beta_k$ se all'i-esimo elemento del vettore delle righe corrisponde la j-esima riga della tabella delle decisioni, si ha che $row[i] = W + (j * \beta_k)$.

il vettore *col* rappresenta il vettore delle colonne. Si osservi che a seconda del valore di β_k per la memorizzazione del vettore delle colonne può essere utilizzato il tipo **unsigned short int**, che memorizza interi nell'intervallo [0, 65536], o il tipo **unsigned char**, che memorizza interi nell'intervallo [0, 256].

La scelta del tipo appropriato è importante dal punto di vista dell'occupazione di memoria del vettore delle colonne e viene effettuata dinamicamente dalla procedura descritta nel paragrafo 6.3.

E' stato necessario realizzare due procedure di lookup che differiscono soltanto per il tipo del vettore *col*, la scelta di dichiarare il vettore *col* di tipo **void** ed effettuare un cast a seconda del valore di β_k non risulta percorribile perché questo introdurrebbe un costrutto condizionale ripetuto per ogni lookup.

Quando vengono caricati, gli indirizzi IP di cui è richiesto il lookup, vengono spezzati in maniera tale da conservare i primi k bit in un registro chiamato **first_pair** e gli ultimi h bit in un altro chiamato **last_pair**.

Su indirizzi IP di m bit, per effettuare il lookup con la struttura CDG si sceglie $k = K\_SIZE$ e $h = m - k$, per il lookup su SCDG si sceglie $k = K\_SIZE + 1$ e $h = m - K\_SIZE$.

Esempio 7.1.1. *Sia X un indirizzo IP di cui si vuole effettuare il lookup, se $m = 32$ e $k = K\_SIZE$ si ha, nel caso di CDG lookup:*

$first\_pair = first(k, X)$

$last\_pair = last(32 - k, X)$

Nel caso SCDG, mantenendo gli stessi valori di m e k:

$first\_pair = first(k+1, X)$

$last\_pair = last(32-k, X)$

La procedura di lookup può essere espressa ad alto livello tramite un'unica istruzione:

$$lookup = row[first\_pair][col[last\_pair]];$$

Risulta evidente da un'analisi del codice che la procedura di lookup non esegue alcuna elaborazione, essa si limita ad eseguire tre accessi alla memoria e precisamente, un accesso diretto al vettore row in cui trova un puntatore alla riga della tabella delle decisioni memorizzata in W, un accesso diretto al vettore col in cui trova l'offset all'interno della riga ed un accesso indiretto a W in cui si ottiene il valore di hop associato all'indirizzo di cui si era richiesto il lookup.

7.2 Bucket lookup

La procedura di bucket lookup è leggermente piú complessa di quella appena descritta. La differenza tra le due risiede in come viene trovato l'offset all'interno della riga della tabella delle decisioni.

Si ricorda brevemente che il vettore dei bucket B contiene $|B| + Fb$ elementi dove: F rappresenta il numero di bucket contenenti un fault, $b = 2^m$ tale che $2^m \leq 2^h/\beta_k < 2^{m+1}$ è il numero di elementi contenuti in un bucket e $|B| = 2^h/b$ è il numero complessivo di bucket.

Quando viene creato, al vettore dei bucket vengono associate le seguenti costanti necessarie per il lookup:

- $offset = |B| - b$ contenente l'offset, all'interno del vettore bucket, in cui inizia il segmento dei bucket.

- $shift = \log_2 b$ contenente il numero di bit di cui è necessario effettuare gli shift per le operazioni di allineamento.

- $size = b$ contenente il numero di elementi di cui è composto un bucket.

La procedura di bucket lookup necessita dell'ausilio della variabile *last*, questa non sarebbe indispensabile, ma il suo impiego evita di ricalcolare un valore che viene utilizzato due volte. La variabile last viene dichiarata di tipo **register** per indicare al compilatore che essa deve essere conservata in uno dei registri interni del processore e non in cache. In questo modo l'accesso alla variabile *last* viene fatto, se possibile, in parallelo con altre istruzioni e non comporta nessun overhead.

il vettore B viene indirizzato tramite gli $h-shift$ bit piú significativi di $last\_pair$, per far ciò è necessario effettuare uno shift dei bit contenuti in $last\_pair$.

Il secondo passo della procedura di lookup consiste nel test del valore di *last*, se questo è positivo significa che $last = col[last\_pair]$ e quindi viene utilizzato l'algoritmo di lookup descritto nel paragrafo precedente. Per costruzione del vettore dei bucket *last* contiene un numero positivo se nel bucket corrispondente non si sono verificato dei fault, in caso contrario, *last* contiene un valore negativo che fa riferimento alla porzione del vettore dei bucket in cui è memorizzato il bucket desiderato. Dai dati sperimentali si è osservato come il numero di bucket contenenti un fault sia abbastanza basso e quindi la probabilità che last sia maggiore di zero è molto alta. Il sistema di *branch prediction* dei processori tenderà quindi a predire $last \geq 0$, in questo modo nella maggior parte dei casi il costrutto condizionale della procedura di lookup non avrà alcun peso nelle prestazioni di lookup.

Nel caso in cui *last* risulti essere un numero negativo bisogna effettuare diversi calcoli oltre ad un quarto accesso alla memoria. Il valore di *last* cambiato di segno e moltiplicato per 2^{shift} a cui venga sommata la costante $offset$ rappresenta l'off-

set all'interno del vettore dei bucket in cui si trova la copia del bucket desiderato, all'interno di tale bucket si accede tramite gli ultimi $shift$ bit di $last\_pair$.

In realtà i calcoli necessari per accedere alla giusta posizione del vettore bucket potrebbero essere effettuati in maniera piú efficiente tramite delle ipotesi su come vengono memorizzati i dati all'interno del processore, ma questa scelta non è stata presa in considerazione perché in conflitto con quella di ottenere del codice portabile su qualunque architettura.

Di seguito viene riportato il codice della procedura di bucket lookup.

```
register short int last = B[last_pair ≫ shift];
if (last ≥ 0)
  lookup = row[first_pair][last];
else
  lookup = row[first_pair][B[offset +((-last) ≪ shift) + (last_pair % size)]];
```

Si osservi che, nella prima istruzione, accedere a $B[last\_pair \gg shift]$ equivale ad accedere al segmento de riferimenti. Per ogni lookup viene effettuato un accesso a questo segmento, per questa ragione, a regime, se le sue dimensioni lo consentono, da quando viene caricato in cache L1, non viene piú rimosso.

La dimensione del segmento dei riferimenti dipende dalla distribuzione che avrebbero avuto gli elementi del vettore delle colonne. La media delle dimensioni assunte dai segmenti dei riferimenti delle tabelle impiegate per la sperimentazione illustrata nel capitolo 10 è di 764 byte. Si consideri che il vettore dei bucket viene memorizzato in un tipo **short int** il quale occupa 2 byte, quindi sperimentalmente $|B| = 382$.

Nell'ipotesi in cui le richieste di lookup vengano fatte su indirizzi casuali, la probabilità che $last$ sia minore di zero, ovvero che sia necessario accedere al segmento dei bucket, è una variabile casuale $X = \frac{F}{|B|}$. Sia i la i-esima tabella di routing

esaminata nella sperimentazione descritta nel capitolo 10, la speranza matematica della distribuzione di probabilità della variabile aleatoria $X_i = \frac{F_i}{|B_i|}$ vale 0.1814.

7.3 Esempi di lookup

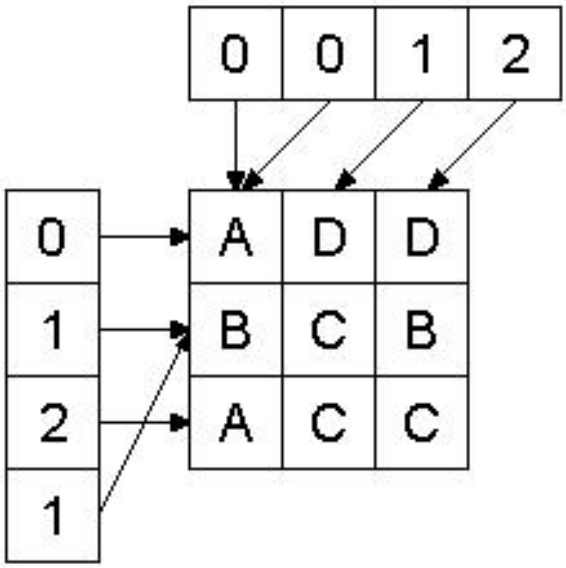
In questa sezione si mostrano due esempi di lookup effettuati sia impiegando la procedura standard che il bucket lookup. Il primo lookup con la procedura standard viene effettuato sulla struttura CDG, il secondo sulla SCDG in modo da poter fare alcune considerazioni e raffronti. Per entrambi i lookup effettuati col la procedura bucket si è scelto di impiegare per maggior chiarezza soltanto la struttura CDG. Si noti che l'impiego della struttura SCDG o CDG non influenza la procedura di bucket lookup.

Negli esempi che seguono si è scelto di trattare i vettori al pari del linguaggio C, la tabella delle decisioni non viene rappresentata tramite il vettore W, ma come tabella per ragioni di chiarezza. Si ricorda che negli esempi seguenti gli indirizzi IP vengono espressi come numeri binari e sono lunghi $m = 4$ bit e che la costante $k = 2$.

Negli esempi seguenti accadrà spesso che vengano usati numeri sia in base due che in base dieci a seconda delle necessità. La base viene indicata dal pedice seguente il numero, ove non indicato i numeri si intendono espressi in base dieci.

7.3.1 Esempio di lookup su struttura CDG

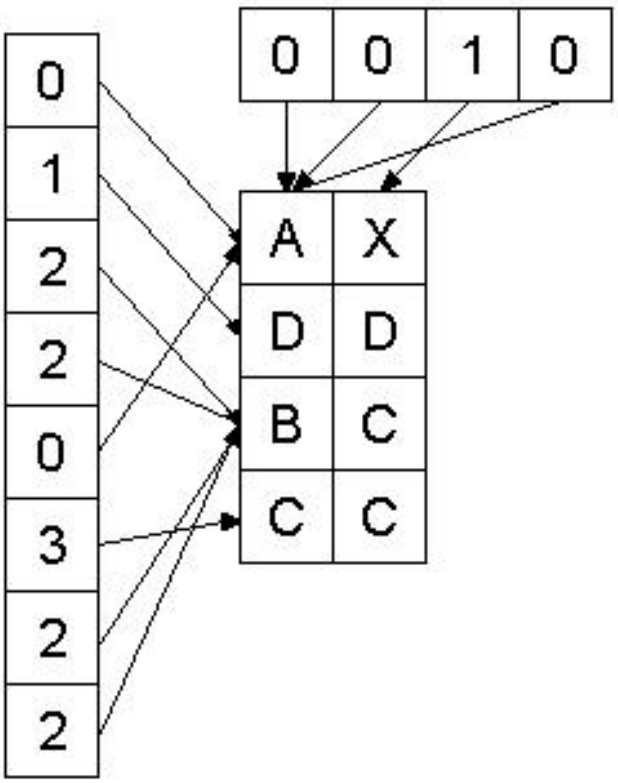
Si immagini di voler effettuare il lookup dell'indirizzo IP 1110_2, avendo indirizzi di 4 bit ed essendo $k = 2$, vale che: $first(2, 1110_2) = 11_2$ e $last(2, 1110_2) = 10_2$.



In figura si vede che $row[11_2]$ punta alla seconda riga della tabella delle decisioni, $col[10_2]$ contiene l'offset all'interno di tale riga. La procedura di lookup restituisce quindi il next hop C.

7.3.2 Esempio di lookup su struttura SCDG

In questo esempio si vuole effettuare il lookup dell'indirizzo 0001_2. In questo caso, considerando $k = K\_SIZE = 2$ la chiamata alla funzione $first()$ deve essere effettuata passandole come primo parametro $k+1$, quindi sarà $first(3, 0001_2) = 000_2$ e $last(2, 0001_2) = 01_2$.



Dalla figura si nota che $row[000_2]$ seleziona la prima riga della tabella delle decisioni, l'offset all'interno di tale riga è contenuta in $col[01_2]$. La procedura di lookup restituisce cosí il valore A.

E' importante sottolineare che nella tabella delle decisioni è presente un simbolo *don't care*, esattamente il secondo elemento della prima riga. A tale elemento non è possibile accedere. Infatti, la prima riga viene puntata soltanto da $row[000_2]$ e $row[100_2]$. In entrambi i casi il bit meno significativo dell'indice del vettore delle righe vale 0. Tale bit è anche il piú significativo con il quale si accede al vettore delle colonne. In altre parole nel caso in cui si acceda alla prima riga della tabella delle decisioni, si utilizza la prima metà del vettore delle colonne per selezionare l'offset. Dalla figura si nota immediatamente che nella prima metà del vettore delle colonne non ci sono riferimenti alla posizione contenente il carattere *don't care* rendendo impossibile l'accesso ad esso cosí come desiderato.

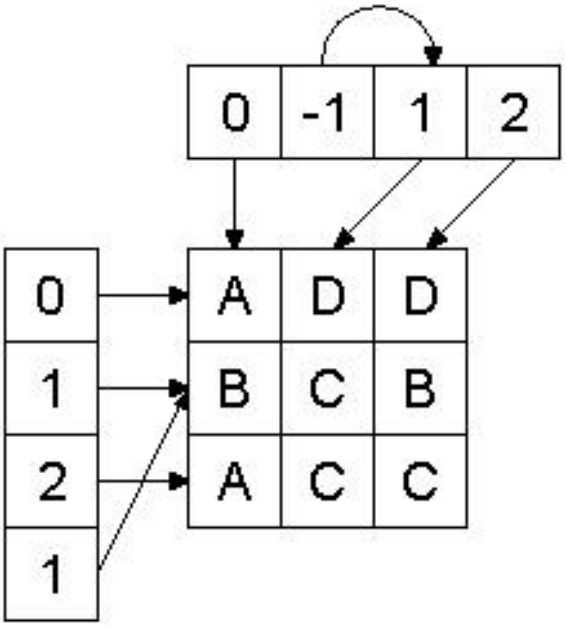
7.3.3 Esempi di bucket lookup

In questo paragrafo si effettuano i lookup agli indirizzi 0001_2 e 1110_2 discussi nei paragrafi precedenti mediante la procedura di bucket lookup e l'impiego del vettore dei bucket.

Dalla costruzione dei bucket risulta che sono presenti $|B| = 2$ bucket contenenti ognuno $b = 2$ elementi. Il primo bucket non contiene un fault, mentre il secondo si.

La costante $offset$ vale quindi 0, la costante $shift$ vale 1 e la costante $size$ vale 2.

Si osservi che il vettore dei bucket in figura contiene quattro elementi, i primi due rappresentano il segmento dei riferimenti, gli ultimi due il segmento dei bucket.



Nell'effettuare il lookup dell'indirizzo 0001_2 si ha: $first\_pair = 0_2$ e $last\_pair = 01_2$. La variabile last prende il valore $B[last\_pair \gg shift]$. Si osservi che $last\_pair \gg shift$ è composto da un solo bit con il quale, quindi si può accedere soltanto ai primi due elementi del vettore dei bucket, ovvero al segmento dei riferimenti. Il test di $last \geq 0$, da esito positivo quindi per questo lookup viene

impiegata la procedura standard di lookup in cui $col[last\_pair] = last$. Si seleziona quindi la riga puntata da $row[00_2]$, all'interno di questa si sceglie l'elemento in posizione last. Da verifica diretta si vede che la procedura di lookup restituisce il valore A come discusso nell'esempio del paragrafo precedente.

Si prova adesso ad effettuare il lookup dell'indirizzo 1110_2. In questo caso si ha che $first\_pair = 11_2$ e $last\_pair = 10_2$. In questo caso $last\_pair \gg shift = 1$ quindi viene assegnato a last il valore di $B[1] = -1$. Questa volta il test di $last \geq 0$ fallisce quindi deve essere fatto un accesso al segmento dei bucket. Il bucket al quale si vuole accedere è il -last-esimo bucket copiato nel segmento dei bucket. All'interno di tale bucket si cerca l'elemento la cui posizione è data dall'ultimo bit di $last\_pair$. In definitiva si vuole accedere alla posizione $offset + ((-last) \ll shift) + (last\_pair\%size) = 0 + ((--1) \ll 1) + 10_2\%2 = 3$ del vettore dei bucket. In $B[3]$ è contenuto il valore $col[last\_pair]$ per la selezione dell'offset in cui si trova il next hop richiesto. La procedura di lookup seleziona quindi la riga data da $row[11_2]$ ed all'interno di questa l'elemento calcolato come appena discusso. Il valore restituito è C cosí come desiderato.

Capitolo 8

Aggiornamento della tabella delle decisioni

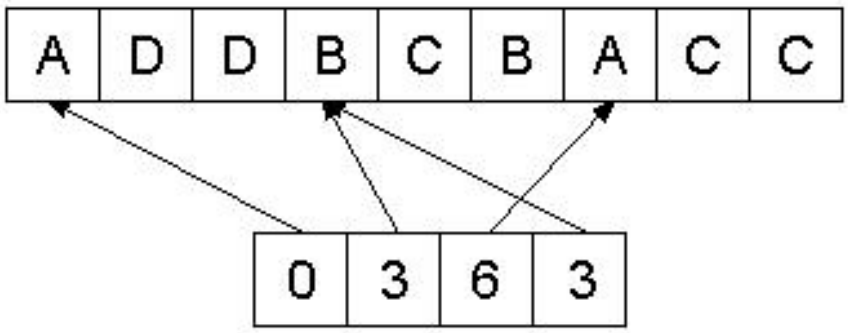
Si è già discusso nel capitolo 4 della necessità di ricostruire la tabella delle decisioni a partire dalla tabella di routing ad ogni aggiornamento. In questa tesi viene proposto un algoritmo di aggiornamento della tabella delle decisioni che risolve parzialmente il problema grazie all'impiego del vettore V_{RLE} In questo capitolo si descrivono le problematiche che sono sorte nella progettazione di tale algoritmo di aggiornamento.

8.1 Problemi connessi all'aggiornamento della tabella delle decisioni

Si è visto nei capitoli precedemto che le dimensioni della tabella delle decisioni vengono calcolate in base agli effetti della compressione RLE sulle righe ed al numero di cluster ridondanti che si riesce ad eliminare. Per questa ragione non è pensabile allocare staticamente una matrice che contenga la tabella delle decisioni, in quanto, tale matrice potrebbe essere di dimensioni eccessive oppure insufficenti.

Una via percorribile sarebbe quella di allocare dinamicamente le righe della tabella delle decisioni, questa scelta, però, risulta inadeguata per ragioni di efficienza. Il metodo migliore è quello di allocare un vettore unico, chiamato W, che contenga le righe della tabella disposte l'una di seguito all'altra e inserire nel vettore delle righe gli offset all'interno di tale vettore.

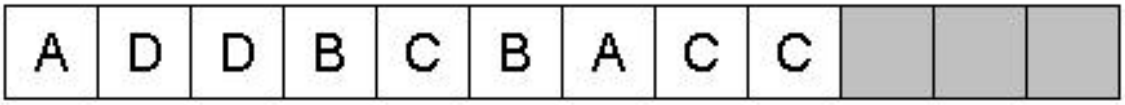
Esempio 8.1.1. *In figura, con riferimento alla tabella di routing del capitolo 4, si mostra come viene allocato il vettore contenente la tabella delle decisioni e come si presenta il vettore delle righe ad esso riferito.*



L'inserimento di una nuova coppia nella tabella di routing T, equivale all'inserimento di un nuovo intervallo nello spazio di indirizzamento e conseguentemente alla modifica del vettore V_{RLE}. Conseguentemente ciò può comportare la necessità di inserire un nuovo cluster o di modificare la compressione RLE delle righe; in sostanza può modificare le dimensioni della tabella delle decisioni.

L'inserimento di una nuova riga può essere effettuato ampliando le dimensioni del vettore contenente la tabella delle decisioni ed inserendo la nuova riga in coda.

Esempio 8.1.2. *Con riferimento all'esempio precedente si mostra come viene trasformato il vettore W per l'inserimento di una riga. In figura, i nuovi elementi sono stati evidenziati in grigio.*



L'inserzione di una riga non crea problemi apparenti, infatti non è necessario spostare gli elementi di W.

L'inserzione di una colonna, invece, genera il problema di dover spostare gli elementi all'interno di W, infatti bisogna lasciare uno spazio vuoto al termine di ogni sequenza che rappresenta una riga. Per far questo, tutti gli elementi alla destra di ogni riga, si devono spostare di una posizione verso destra.

Esempio 8.1.3. *Viene mostrato in figura come si trasforma il vettore W se si volesse inserire una nuova colonna nella tabella delle decisioni.*

Per ridimensionare un vettore creato dinamicamente, il linguaggio C dispone della funzione di libreria *realloc*(), tale funzione procede nel seguente modo: se c'è spazio libero sufficiente contiguo alla struttura di cui si vogliono cambiare le dimensioni, lo rende disponibile in fondo a tale struttura, altrimenti viene allocato un nuovo segmento di memoria delle dimensioni appropriate ed in esso vengono copiate le informazioni contenute nella struttura che si voleva espandere.

Nel caso vi sia spazio sufficiente, l'inserimento di una nuova riga non necessita quindi di effettuare lo spostamento degli elementi. Nel caso lo spazio non sia abbastanza, la *realloc*() deve invece ricopiare l'intero vettore W.

Nel caso si inserisca una colonna nella tabella delle decisioni, le cose vanno peggio;

infatti, oltre a dover effettuare uno spostamento degli elementi di W per inserire gli spazi, è possibile che anche $realloc()$ debba effettuare una copia di W.

Se si considera che il vettore delle righe, per ragioni di efficienza, è costituito da puntatori a segmenti di W, sia lo spostamento degli elementi di W che la loro copia, hanno la stessa complessità computazionale di un passo di ricostruzione della tabella.

Dai discorsi appena fatti risulta evidente che un passo di ricostruzione della tabella è intrinsecamente non eliminabile.

In questa tesi viene proposto un algoritmo per l'aggiornamento della tabella delle decisioni che si pone i seguenti obiettivi:

- essere indifferentemente utilizzabile sia per la tabella CDG che per quella SCDG
- evitare di dover ricostruire la tabella delle decisioni a partire da T, ma mantenendo aggiornato V_{RLE}
- essere implementabile in maniera efficiente.

8.2 La procedura di insert

La procedura di inserimento di una coppia (P, hop_P) tale che $P \neq \epsilon$ effettua quattro passi:

- Inserimento del prefisso nel trie
- Inserimento dell'estremo destro
- Visita del trie
- Inserimento dell'estremo sinistro

8.2.1 Inserimento del prefisso

Il primo passo della procedura di inserimento consiste nella creazione del percorso tra la radice ed il nodo N associato al prefisso P. Se tale percorso non esiste vuol dire che nel trie non esiste un prefisso di cui P sia a sua volta prefisso. Questo vuol dire anche che il nodo P rappresenta una rete aggregata per la quale non esistono eccezioni e che il nodo N è un nodo foglia.

Sia N_0 la radice del trie, P il prefisso che si vuole inserire e $|P|$ il numero di bit da cui è composto.

Per $0 \leq i < |P|$, si percorre il cammino in cui N_{i+1} è il figlio sinistro di N_i se l'i-esimo bit di P vale 0, il figlio destro se vale 1. Si supponga che, per $i = k$, il nodo N_{i+1} non esista, a partire da N_k bisogna inserire nel trie i nodi che servono a completare il cammino fino ad N.

Per $k \leq i < |P|$ se l'(i+1)-esimo bit di P vale 0 ad N_i si aggiunge un figlio sinistro etichettato con il valore di hop di N_i, se vale 1 si crea un figlio destro etichettato nel medesimo modo. Tale figlio viene indicato con N_{i+1}.

Il nodo $N_{|P|} = N$ viene marcato come prefisso e gli viene assegnato il valore h_P.

Si noti che, se N_k non era un nodo foglia prima dell'inserimento del cammino fra la radice ed N, ogni nodo che era foglia prima dell'inserimento di tale cammino resterà foglia anche dopo. Altrimenti N_k diventerà un nodo interno.

8.2.2 Inserimento dell'estremo destro

Sia m il numero di bit di cui è composto un indirizzo IP A partire da P si calcola il valore $P_h + 1$ dove P_h è come in 2.7.1, ovvero P a cui sono stati aggiunti $m - |P|$ bit uguali ad 1 in coda. Si percorre il cammino suggerito dai bit di $P_h + 1$, a partire da quello piú significativo, finche possibile. Si supponga di essersi fermati al nodo M a profondità k. Se il (k+1)-esimo bit di $P_h + 1$ vale 0 si aggiunge ad M un

figlio sinistro se vale 1 un figlio destro. Per costruzione questo figlio deve essere una foglia, altrimenti sarebbe stato possibile andare avanti nel cammino. Al campo hop del nodo cosí creato si da il valore dello stesso campo di M.

Sia $N_{|P|-1}$ il nodo padre di N, se N è figlio sinistro di $N_{|P|-1}$, il nodo M non sarà altro che $N_{|P|-1}$. Ovviamente il (k+1)-esimo bit di $P_h + 1$ valrà 1, altrimenti N non sarebbe stato un figlio sinistro.

Nel caso l'estremo destro dell'intervallo corrispondente a P non sia sovrapposto all'estremo destro, o non sia contiguo ad un estremo sinistro, di un intervallo già inserito, tra l'estremo destro di P e l'estremo successivo, sia esso destro o sinistro, si crea un nuovo intervallo. Del nuovo intervallo che si è venuto a formare deve essere tenuta traccia nel trie; la procedura appena descritta si occupa di fare questo. Esiste una condizione sufficiente per stabilire che questa situazione non si sia verificata e quindi che permette di saltare il passo appena descritto. Se N_k non è era un nodo foglia prima di inserire il cammino riferito a P, la procedura descritta in questo paragrafo può non essere eseguita.

8.2.3 Visita del trie

L'inserimento del cammino dalla radice ad N crea all'interno del trie un riferimento all'intervallo $[P_l,P_h]$ dove P_l e P_h sono costruiti come in 2.7.1, ovvero: P_l è P a cui sono stati aggiunti $m - |P|$ bit uguali a 0 in coda, P_h è P a cui sono stati aggiunti $m - |P|$ bit uguali ad 1 in coda.

Si possono verificare due casi:

- nel trie è presente un sottointervallo di $[P_l,P_h]$
- l'intervallo $[P_l,P_h]$ non contiene sottointervalli al suo interno.

Nel primo caso il nodo N è la radice di un trie a cui è associato l'intervallo $[P_l,P_h]$.

Tutti i nodi marcati come prefisso all'interno di questo trie rappresentano i riferimenti ai sottointervalli di $[P_l,P_h]$. Per associare ai frammenti dell'intervallo $[P_l,P_h]$ il giusto valore di hop si si effettua una visita del trie che ha N come radice sostituendo il valore hop di ogni nodo con hop_P, se si incontra un nodo marcato come prefisso si ignora l'intero sottoalbero che ha quest'ultimo come radice.

Nel secondo caso, invece, N è un nodo foglia quindi il valore corretto di hop è già stato assegnato dal passo descritto in 8.2.1.

8.2.4 Inserimento dell'estremo sinistro

Anche questo passo, come quello descritto in 8.2.2, va eseguito soltanto se N_k era un nodo foglia prima di inserire il cammino riferito a P. Nel caso l'estremo sinistro dell'intervallo non fosse sovrapposto a nessun altro estremo sinistro, o nel caso in cui non fosse contiguo ad un estremo destro di un intervallo preesistente, si crea un intervallo tra l'estremo precedente l'estremo sinistro dell'intervallo riferito a P e quest'ultimo. Di questo intervallo si tiene traccia nel trie mediante la procedura descritta di seguito

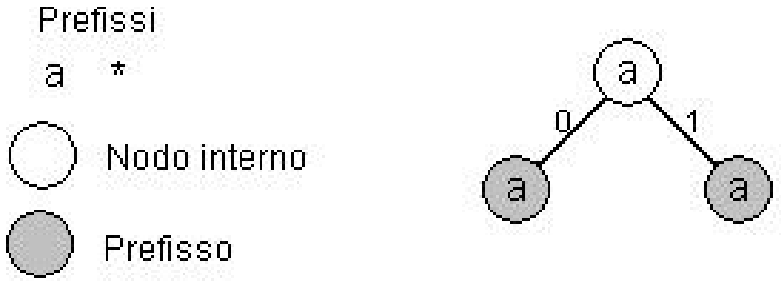
Sia A il nodo marcato come prefisso piú diretto predecessore di N. Seguendo il cammino tra A compreso, ed N escluso, si cerca, se esiste, il primo nodo M per cui valga la seguente proprietà:

Proprietà 2. *Si percorra a partire da M il cammino ottenuto scegliendo sempre il figlio sinistro, il nodo I, a partire dal quale non è piú possibile andare avanti, non è una foglia.*

Si aggiunge ad I un figlio sinistro che erediti da I il valore di hop.

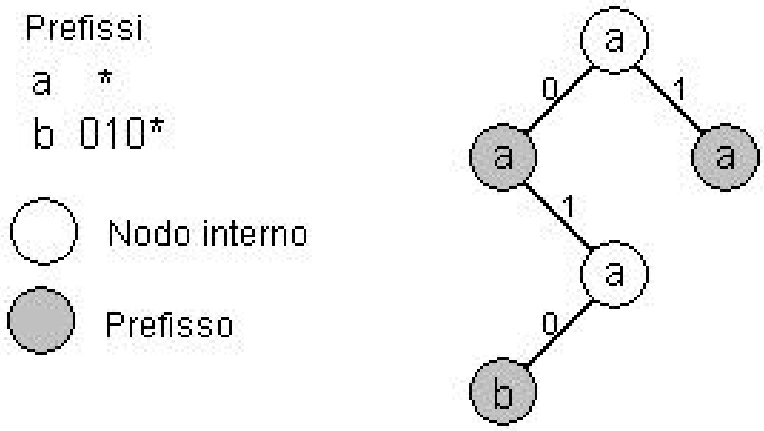
8.2.5 Esempio di inserimento di un prefisso

Si immagini il trie inizializzato con la sola coppia (ϵ, h_ϵ) dove $h_\epsilon = a$, esso contiene la radice etichettata con a ed i due figli della radice, anch'essi etichettati con a.

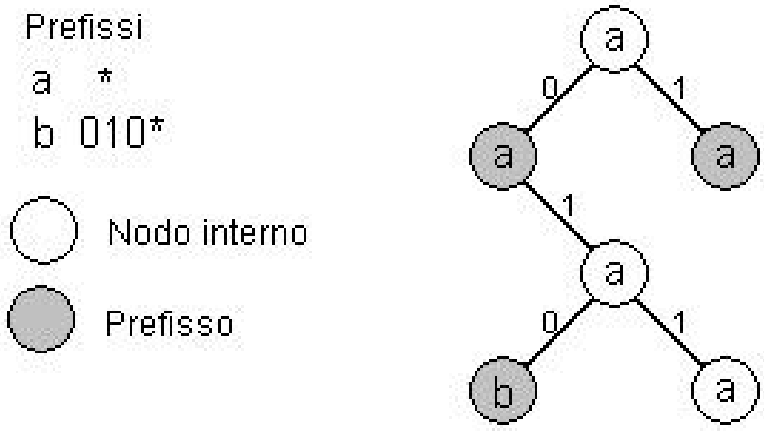


Il primo passo dell'inserimento della coppia $(010*, b)$ consiste nella creazione del percorso tra la radice ed il nodo associato al prefisso 010. Essendo il prefisso inserito di tre bit, il nodo N associato si troverà a profondità 3.

Partendo dalla radice si segue il percorso verso N, dal momento che il primo bit del prefisso che si vuole inserire vale 0, si sceglie il figlio sinistro. A questo punto ci si rende conto che non è piú possibile proseguire e quindi si devono inserire degli altri nodi per costruire il cammino fino ad N. Con la notazione dei paragrafi precedenti il figlio sinistro della radice è il nodo N_k e $k = 1$. A N_k viene aggiunto un figlio destro perché il secondo bit del prefisso da inserire vale 1, A questo nodo si aggiunge un figlio sinistro perché l'ultimo bit del prefisso vale 0. A questo punto l'ultimo nodo inserito è N e viene marcato come prefisso.

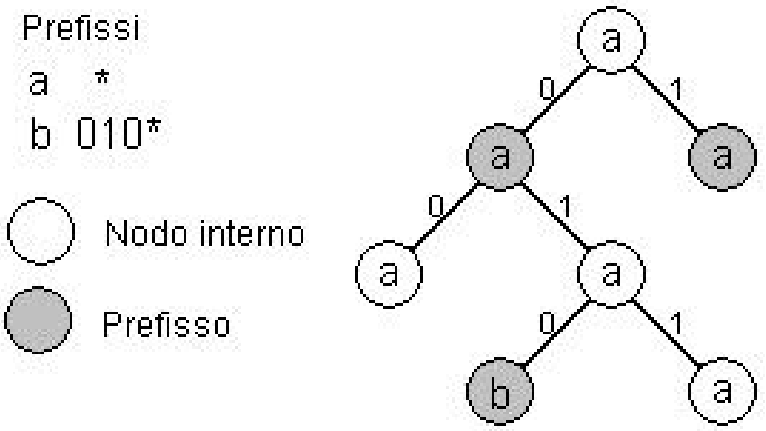


Prima dell'inserimento del percorso relativo a 010, N_k era una foglia quindi è necessario eseguire il passo 2 che ha l'effetto mostrato in figura. In questo esempio $P = 010$ per cui $P_h = 0101$. Si osservi anche che: essendo N un figlio sinistro, il passo 2 consiste nell'aggiunta del figlio destro al padre di N.



Il terzo passo non viene effettuato in quanto N è un nodo foglia.

Il quarto passo consiste nella ricostruzione dell'intervallo precedente quello riferito a P. Il trie si modifica così come mostrato in figura.



Dalla figura si osserva che il trie ha partizionato lo spazio di indirizzamento, supponendo indirizzi lunghi 4 bit, nei seguenti intervalli [0000, 0011], [0100, 0101], [0110, 0111], [1000, 1111] cosí come richiesto.

8.3 La procedura di delete

Si supponga di voler eliminare dalla tabella di routing T la coppia (P, hop_P), la procedura di delete procede seguendo tre passi:

- Ricerca del nodo corrispondente al prefisso da eliminare
- Rimozione del sottoalbero ridondante
- Riassegnazione delle etichette

Nel primo passo viene fatta una ricerca all'interno del trie, guidata dai bit di P, del nodo N corrispondente al prefisso che si vuole eliminare. Se il percorso non esiste, o il nodo non risulta marcato come prefisso, la procedura di delete termina segnalando un errore.

Nel caso il nodo venga trovato, in esso viene resettato il flag che lo marcava come prefisso. Questa operazione, di fatto rimuove il prefisso P dal trie.

Il secondo passo cerca sottointervalli contigui, all'interno dell'intervallo $[P_l,\ P_h]$ che facciano riferimento allo stesso valore di hop; in caso vengano trovati di questi intervalli, vengono uniti in un unico intervallo.

Il terzo passo è l'analogo di quello descritto in 8.2.3 ed ha lo scopo di propagare a tutti i sottointervalli di $[P_l,\ P_h]$ il corretto valore di hop dopo l'eliminazione di P.

8.3.1 Ricerca del nodo corrispondente al prefisso da eliminare

Il primo passo della procedura di eliminazione della coppia (P, hop_P) consiste nella ricerca ed eliminazione dal trie del riferimento a P.

Si percorre il cammino del trie guidato dai bit di P, nel caso ci si trovi in condizione di non poter scegliere il percorso indicato dal bit di P che si sta esaminando, la procedura di eliminazione termina con un fallimento. Questa situazione è dovuta al fatto di star cercando di eliminare un prefisso inesistente.

Sia N il nodo a cui si giunge alla fine della ricerca, se N non è marcato come prefisso, vuol dire che esiste nel trie un prefisso di cui P è prefisso, ma non esiste in T la coppia (P, hop_P). Anche in questo caso la procedura di eliminazione termina con un fallimento.

Nel caso in cui, invece, il nodo N sia marcato come prefisso si resetta il flag che indica tale marcatura. Questa operazione corrisponde all'eliminazione del riferimento al prefisso P nel trie. Al nodo N viene assegnato il valore di hop del suo nodo padre.

8.3.2 Rimozione del sottoalbero ridondante

Dopo l'eliminazione di un prefisso può accadere che dei sottointervalli contigui inclusi in $[P_l,\ P_h]$ facciano riferimento allo stesso valore di hop. Questo è dovuto al fatto

che il passo descritto nel paragrafo precedente elimina i riferimento a P, ma non modifica gli intervalli in cui è partizionato lo spazio degli indirizzi.

Si effettua una visita in profondità del sottoalbero che ha N come radice, nel caso si trovi un nodo X avente 2 figli i quali sono foglie, nessuno dei due è marcato come prefisso e riferiscono lo stesso valore di hop, questi rappresentano due intervalli contigui con lo stesso valore di hop. Tali intervalli possono essere eliminati. Il nodo X sarà adesso una foglia e ad esso viene riferito l'intervallo composto dall'unione degli intervalli originariamente riferiti ai figli.

La visita in profondità del sottoalberi riferito ad N, garantisce che questa operazione venga fatta ricorsivamente ed unisca tutti gli intervalli contigui.

Si noti, inoltre, che il passo descritto da questo paragrafo non sarebbe necessario ai fini della cancellazione di un prefisso, esso serve per eliminare il fenomeno della frammentazione degli intervalli descritto nel paragrafo 4.3.2. Va inoltre osservato, che la frammentazione degli intervalli, oltre a non introdurre errori non influenza la struttura della tabella delle decisioni.

8.3.3 Riassegnazione delle etichette

Il passo descritto in questo paragrafo ha lo stesso senso di quello descritto nel paragrafo 8.2.3. Si effettua una visita in profondità del sottoalbero che abbia N come radice e si etichettano tutti in nodi incontrati con il valore di hop del nodo padre di N. Nel caso si incontri un nodo marcato come prefisso, si ignora tutto il sottoalbero ad esso riferito.

E' molto importante notare che la ridefinizione delle etichette proposta in questo paragrafo non ha effetti sul passo descritto dal paragrafo precedente. Questo vuol dire che non è importante in quale ordine essi vengano eseguiti.

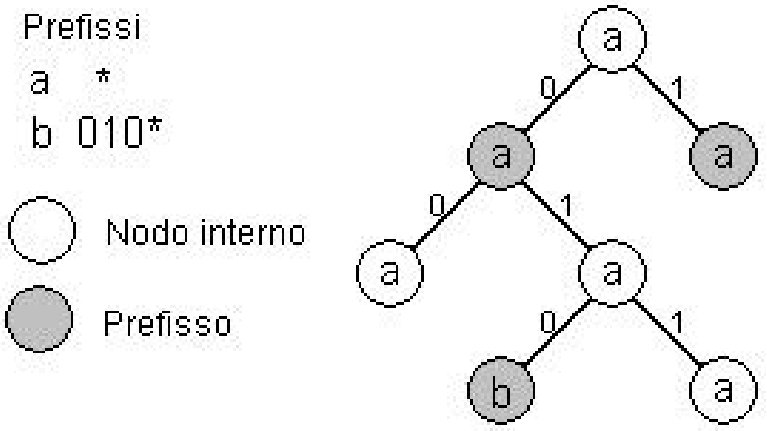
Nell'implementazione pratica si è scelto di fondere insieme questi due passi evi-

tando cosí di ripetere inutilmente la visita in profondità. Si procede nel modo seguente: viene eseguito il passo indicato nel paragrafo precedente, se i figli di X soddisfano le condizioni per l'eliminazione vengono rimossi, altrimenti vengono etichettati con il valore del nodo padre di N.

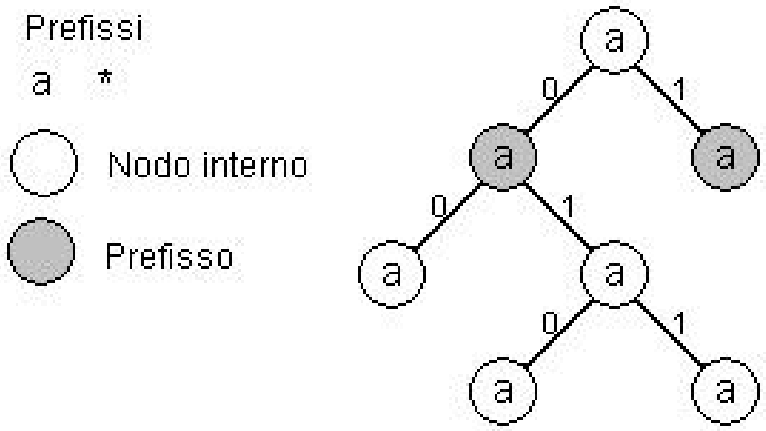
8.3.4 Esempio di cancellazione di un prefisso

Si immagini di trovarsi nella situazione descritta nell'esempio del paragrafo 8.2.5 e si decida adesso di eliminare il prefisso $P = 010*$.

In figura si mostro lo stato del trie prima del'esecuzione della procedura di cancellazione. Come si può subito osservare il nodo associato a P è marcato come prefisso.

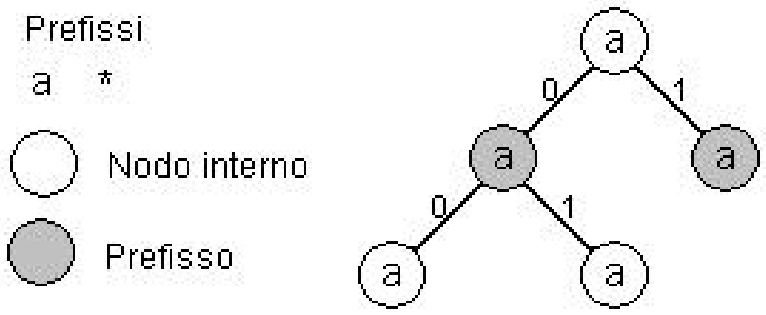


Il primo passo della procedura di cancellazione consiste nel percorrere il cammino tra la radice ed il nodo N che rappresenta il prefisso P. Tale nodo deve non essere piú marcato come prefisso, cosí come mostrato dalla figura seguente e ad esso viene associato il valore di hop del padre. Se, come nel caso in figura il nodo N è una foglia non è necessario procedere alla visita del sottoalbero che ha N come radice, in quanto tale sottoalbero è costituito solo da N.

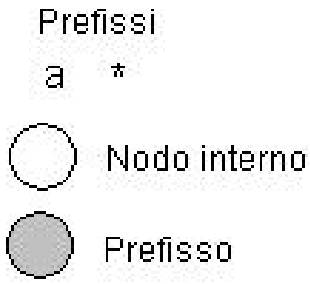


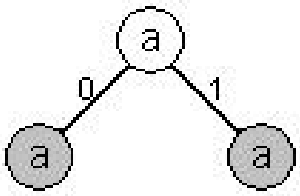
La procedura di cancellazione del prefisso P potrebbe adesso terminare, infatti, nel trie della figura precedente non è piú presente alcun riferimento a P. Quello che si nota immediatamente, invece, è che non sono stati modificati gli intervalli in cui era partizionato lo spazio di indirizzamento. In questo modo si creano degli intervalli contigui che riferiscono allo stesso valore di next hop. Per ovviare a questo problema si procede ad una visita in profondità cosí come spiegato nel paragrafo precedente.

Nelle due figure seguenti, vengono mostrati i passi di ricongiungimento degli intervalli contigui con lo stesso valore di next hop.



Come si può subito osservare la figura seguente è uguale allo stato iniziale del trie nel paragrafo 8.2.5 prima di inserire il prefisso $010*$.





8.4 Ricostruzione della tabella

Dopo aver inserito o eliminato un prefisso dal trie, se si vuole rendere effettiva la modifica anche sulla tabelle delle decisioni, è necessario effettuare un passo di ricostruzione di tale tabella.

Per prima cosa va ricostruito il vettore V_{RLE}, dopodicchè va chiamato il back end appropriato a seconda che si voglia costruire la tabella CDG o SCDG.

8.4.1 Ricostruzione del vettore V_{RLE}

Per effettuare la ricostruzione del vettore V_{RLE} si procede tramite una visita in profondità del trie. Il primo problema che si pone nell'implementazione della procedura di ricostruzione di tale vettore riguarda il calcolo della dimensione corretta che questo avrà. Il problema viene risolto in maniera molto semplice: sia la procedura di inserimento che quella di cancellazione, quando invocate, restituiscono il numero di nuove foglie inserite, rispettivamente eliminate. La dimensione del vettore V_{RLE}, a questo punto, sarà quella precedente la chiamata della procedura di inserimento, rispettivamente cancellazione, a cui verrà aggiunto o sottratto il valore restituito.

Sia $|V|$ la dimensione del vettore V_{RLE} dopo la richiesta di inserimento o cancellazione. Per $1 \leq i < |V|$, siano N_i le foglie del trie nell'ordine con cui vengono esaminate tramite la visita in profondità.

Se N_i si trova a profondità k_i, si calcola $N_{l,i}$ come il valore di N_i a cui sono stati concatenati m - k_i zeri in coda.

La i-esima posizione del vettore V_{RLE} contiene la coppia (l_i, d_i) in cui $l_i = N_{l,i+1} - N_{l,i}$ e d_i è l'etichetta del nodo N_i.

In coda al vettore V_{RLE} viene inserita la coppia $(2^m - N_{l,|V|}, d_{|V|})$.

A questo punto il vettore V_{RLE} è stato ricostruito e può essere invocato il back end per ricostruire la tabella delle decisioni.

Capitolo 9

Verifica della correttezza degli algoritmi

La verifica della correttezza degli algoritmi di IP address lookup è un problema di difficile soluzione. Non esiste un metodo per dimostrare la loro correttezza in generale. Data una tabella di routing, però, si può verificare se l'algoritmo risponde esattamente alle richieste di IP lookup tramite una prova diretta. Questo tipo di verifica è esaustivo, ovvero richiede che vengano analizzate le risposte ottenute per ogni possibile indirizzo IP; questo significherebbe dover effettuare 2^{32} prove.

In questa tesi viene proposto e realizzato un algoritmo per la verifica della correttezza delle risposte fornite dal CDG lookup e dell'SCDG lookup per una determinata tabella di routing T che sfruttando le loro caratteristiche riesce a verificarne la correttezza con $O(|T|)$ prove anzichè con 2^{32} prove.

Il metodo di prova si basa sul fatto che il CDG lookup, come l'SCDG lookup. assicura la stessa risposta per qualunque elemeno appartenente ad un intervallo $[P_l, P_h]$ riferito ad un prefisso $P \in T$. Per questa ragione lo strumento di verifica presentato in questo capitolo è riutilizzabile per qualinque altro algoritmo di IP

address lookup per cui sia dimostrabile la stessa proprietà.

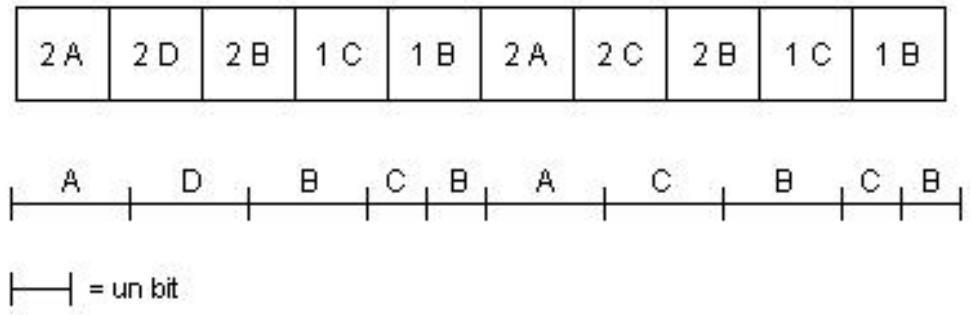
9.1 Proprietà del CDG lookup

Si verifica facilmente che le considerazioni che verranno fatte nel seguito riguardanti l'algoritmo CDG lookup sono valide anche nel caso SCDG. Per questa ragione si farà sempre riferimento soltanto alla struttura CDG.

Il vettore V_{RLE} è composto da $|V|$ coppie dove la coppia $V_i = (l_i, d_i)$.

In figura si vede il vettore V_{RLE} associato alla tabella usata come esempio nel capitolo 4 e gli intervalli ad esso associati. Nella figura si suppone che gli indirizzi IP siano di quattro bit e di conseguenza che lo spazio di indirizzamento sia interamente contenuto nell'intervallo $[0,\ 2^4 - 1]$.

Come si può notare il vettore V_{RLE} definisce una partizione dello spazio di indirizzamento.



Sia m il numero di bit da cui è formato un indirizzzo IP, per costruzione del vettore V_{RLE} si ha che $\sum_{i=1}^{|V|} l_i = 2^m$, questa proprietà sta a significare che lo spazio degli indirizzi viene partizionato in sottointervalli e non esiste una parte dello spazio di indirizzamento che non sia coperta.

Esempio 9.1.1. *Nella figura precedente si è fatto riferimento ad indirizzi lunghi 4 bit. Da una verifica diretta si può notare che* $\sum_{i=1}^{10} l_i = 2^4$.

Sia m il numero di bit di cui è composto un indirizzo IP, si scelga $K\_SIZE = k$ e $h = m - k$. La costruzione della tabella delle decisioni avviene a partire da V_{RLE} partizionando questo in 2^k porzioni chiamate cluster, tali che se l'i-esimo cluster T_i ha dimensione $|T_i|$ si ha $\sum_{j=1}^{|T_i|} l_j = 2^h$. Questa operazione preserva la suddivisione in intervalli.

Anche le operazioni successive nella costruzione dalla tabella CDG descritte nel capitolo 4 non modificano la suddivisione in intervalli dello spazio di indirizzamento.

Le coppie contenute nei cluster vengono scisse in accordo al lemma 4.1.1, che garantisce di lasciare inalterati gli intervalli definiti da V_{RLE}.

Nelle operazioni ulteriori che portano alla struttura CDG non vengono fatte ulteriori manipolazioni, ci si limita ad una memorizzazione piú "pratica" dei cluster.

In definitiva, dall'implementazione del back end si deduce che se l'algoritmo di lookup fornisce la risposta esatta per un qualsiasi punto di un intervallo, fornirà la stessa risposta per tutto l'intervallo. La garanzia del fatto che l'algoritmo copra l'intero spazio di indirizzamento è dato dal fatto che esso copre esattamente gli stessi intervalli coperti dal vettore V_{RLE}, come appena discusso, e che questo rappresenta una partizione dello spazio di indirizzamento di IP.

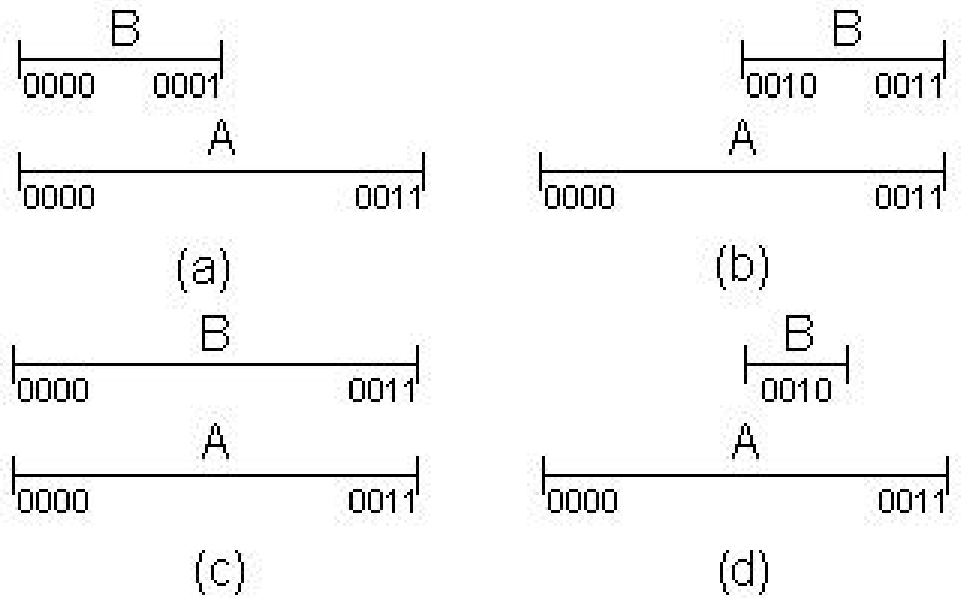
La costruzione del back end garantisce anche una corrispondenza biunivoca fra un intervallo contenuto nel vettore V_{RLE} e la sua rappresentazione nella tabella delle decisioni.

9.2 Descrizione logica dello strumento di verifica

A partire dalle considerazioni fatte nel paragrafo precedente, la verifica della correttezza della struttura CDG può essere fatta verificando i valori agli estremi di ogni intervallo del vettore V_{RLE}. La verifica del primo elemento di ogni intervallo, garantisce la sola correttezza del valore contenuto in esso; per essere certi che oltre a

contenere il valore corretto, l'intervallo abbia anche le dimensioni esatte è necessario effettuare un'interrogazione anche l'ultimo elemento dell'intervallo.

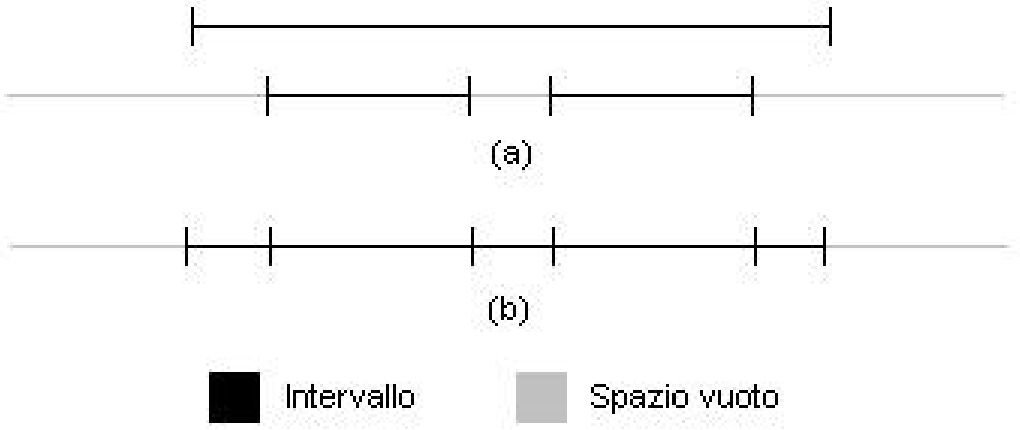
Si nota facilmente che si effettuano $2|V|$ interrogazioni. Una maggiorazione grossolana del valore di $|V|$ si basa sul fatto che l'inserimento di un nuovo prefisso può, nella peggiore delle ipotesi, partizionare un intervallo in tre come mostrato in figura. (Vedere anche paragrafo 4.3.1)



Ciò implica anche che l'inserimento di un nuovo prefisso, nell'ipotesi piú sfortunata, fa incrementare di due il numero di intervalli in cui è partizionato lo spazio degli indirizzi IP. Questo vuol dire che, se $|T|$ è il numero di voci presenti nella tabella di routing T, $|V| \leq 2|T|$.

Si noti che, se si mantengono gli intervalli come una partizione dello spazio di indirizzamento, per il lemma 2.7.1 non sono possibili altri casi oltre quelli presentati nella figura precedente.

Contrariamente se gli intervalli non rappresentassero una partizione dello spazio di indirizzamento, il numero di intervalli generato ad ogni inserimento non sarebbe maggiorabile come mostra il controesempio nella figura seguente



Si nota infatti che l'inserimento di un nuovo segmento fa in maniera tale che il numero di intervalli passi da due a cinque facendo si che si generino tre nuovi intervalli.

Per mantenere lo spazio di indirizzamento costantemente come una partizione, è sufficiente inserire l'intervallo $[0, 2^{32}]$ ed assegnare ad esso il valore di **no route to host**.

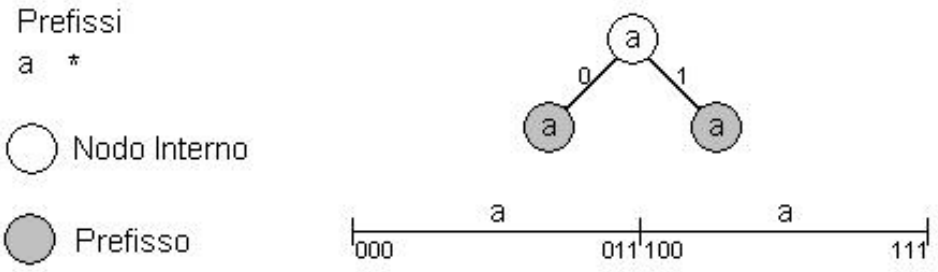
Si può finalmente concludere che il numero delle interrogazioni da fare vale al massimo $4|T|$ che è nell'ordine $O(|T|)$ ed è molto minore di 2^{32}.

Restano aperti due problemi:

- come trovare gli estremi degli intervalli memorizzati in V_{RLE}
- come ottenere la risposta corretta alle interrogazioni da fare.

La soluzione a questi due problemi si basa sull'impiego di un binary trie. A partire da una tabella di routing si costruisce un albero come quello del paragrafo 3.1.1. Per comodità ai nodi non marcati come prefissi viene assegnato il valore del loro diretto predecessore marcato come prefisso. Per assicurarsi che questo albero partizioni lo spazio di indirizzamento prima di effettuare la costruzione vengono inseriti i prefissi $0*$ e $1*$ a cui viene assegnato il valore della rotta di default.

Esempio 9.2.1. *In figura si mostra l'albero inizializzato senza che siano stati ancora aggiunti gli altri prefissi e la proiezione degli intervalli che esso genera, si suppone sempre che gli indirizzi IP siano di quattro bit.*



Il paragrafo 3.1.1 ci fornisce un metodo facile per interrogare il binary trie la cui correttezza, oltre ad essere facile da dimostrare, è universalmente riconosciuta da tutti i lavori presenti in letteratura.

In questo modo si è risolto il problema di effettuare delle interrogazioni corrette, bisogna ancora calcolare gli estremi degli intervalli generati dal vettore V_{RLE}.

Per far questo ci si serve del seguente lemma:

Lemma 9.2.1. *Sia A il trie descritto nel paragrafo 3.1.1, sia N un qualunque nodo di A tale che N non possiede il figlio sinistro, equivalentemente destro, sia h il valore del next hop associato ad N. L'albero generato da A al cui nodo N è stato aggiunto il figlio sinistro, equivalentemente destro, al quale è stato associato h per il valore del next hop, è equivalente ad A.*

La verifica dell lemma 9.2.1 è molto semplice e si basa sul lemma 2.7.2.

In base a quanto affermato dal lemma precedente, il trie può essere costruito incrementalmente inserendo in esso i prefissi utilizzando la procedura di inserzione discussa nel paragrafo 8.2.

A questo punto tramite una visita in profondità si ottengono i valori dell'estremo sinistro di ogni intervallo contenuto in V_{RLE}. Per ottenere il valore dell'estremo

destro di un intervallo basta sottrarre uno al valore dell'estremo sinistro dell'intervallo successivo. Ottenuti i valori degli estremi degli intervalli si effettuano i lookup secondo la procedura del paragrafo 3.1.1.

9.3 Dettagli implementativi

Oltre ad una realizzazione teorica di uno strumento per la verifica della correttezza dei lookup effettuati tramite gli algoritmi implementati in questa tesi, si fa ora una breve panoramica su come è stato implementato tale strumento.

Il programma di verifica prende in input un file, **nomefile**, contenente una tabella di routing e restituisce i file: **nomefile.query** e **nomefile.reply**. Nel primo vengono inseriti gli indirizzi IP su cui effettuare il lookup, nel secondo vengono fornite le risposte nello stesso formato del file di log del programma di lookup. In questo modo il confronto tra le risposte fornite dal programma di lookup e quello corrette può essere fatta utilizzando il comando unix **diff** o un qualsiasi analogo.

Il programma di verifica passa attraverso le seguenti fasi:

- Viene creata la radice del trie.
- Si creano i figli sinistro e destro della radice che vengono contrassegnati come prefissi e contengono nel campo hop il valore di "*no route to host*". Questo passo equivale all'inserimento dei due prefissi $0*$ e $1*$ che generano una partizione dello spazio di indirizzamento.
- Per ogni prefisso appartenente alla tabella di routing da verificare si aggiorna il trie tramite la procedura di inserimento discussa nel paragrafo 8.2.
- Si effettua una visita in profondità del trie cosí costruito. Per ogni nodo N_i visitato si procede in questo modo: se questo non è una foglia lo si ignora

e si procede nella visita. Altrimenti sia P_i il prefisso associato ad N_i, e sia d_i la profondità di N_i, allora N_i rappresenta l'intervallo costruito secondo la definizione 2.7.1 dove $P_i = P$ e $d_i = k$. Siano $P_{i,l}$ $P_{i,h}$ gli estremi di questo intervallo, essi rappresentano i due indirizzi IP su cui effettuare l'interrogazione per l'intervallo rappresentato da N_i. Non è necessario effettuare il lookup nel trie come discusso nel paragrafo precedente, infatti, come valore per next hop di entrambi gli estremi, si usa quello contenuto nel nodo N_i.

Capitolo 10

Test e confronto delle prestazioni

In questo capitolo vengono presentati i risultati ottenuti dai test degli algoritmi presentati negli scorsi capitoli su 2040 tabelle di routing, raccolte dall'IPMA [10] nel corso di tre anni e mezzo.

10.1 La piattaforma utilizzata

Per condurre gli esperimenti è stato usato un personal computer con le seguenti caratteristiche tecniche:

- Processore AMD Athlon XP 1900+ a 1.6 GHz
- Red Hat Linux 7.3 (kernel 2.4.18-3 per i686)
- gcc v 2.96

10.1.1 Caratteristiche tecniche del processore

Per i test è stato utilizzato un processore AMD Athlon XP 1900+ con una frequenza interna di 1,6 Ghz, ovvero capace di eseguire un'istruzione in 0.625ns. La cache L1

è di 128Kb di cui 64Kb dati ed i rimanenti 64Kb istruzioni. La cache è costituita da pagine di 4Kb ed ha una latenza di 1.3ns. La cache L2 è di 256Kb con una latenza di 7ns

10.1.2 Ambiente di test

I test sono stati condotti in ambiente Linux, per evitare quanto piú possibile l'interazione con altri processi, si è scelto di utilizzare il sistema in modalità single user scegliendo il runlevel 1.

10.2 Tabelle BGP

I test sono stati condotti utilizzando 2040 tabelle BGP fornite dall'IPMA project. Esse appartengono a cinque router e coprono un periodo che va dall'inizio del 1999 a maggio 2002. Purtroppo non sono state ancora rese pubbliche tabelle piú recenti.

Nella tabella seguente vengono indicati: il nome del router, il numero di tabelle a disposizione per esso, il numero medio di next hop contenuti nelle tabelle ed il numero medio di prefissi.

Router	Totale tabelle	N° next hop	N° route
Aads	538	33	30377
Mae West	618	40	34215
Mae East	230	42	23125
Paix	78	23	15352
Pacbell	576	1	27945

Distribuzione tabelle per router

10.3 Distribuzione degli address prefix nelle tabelle

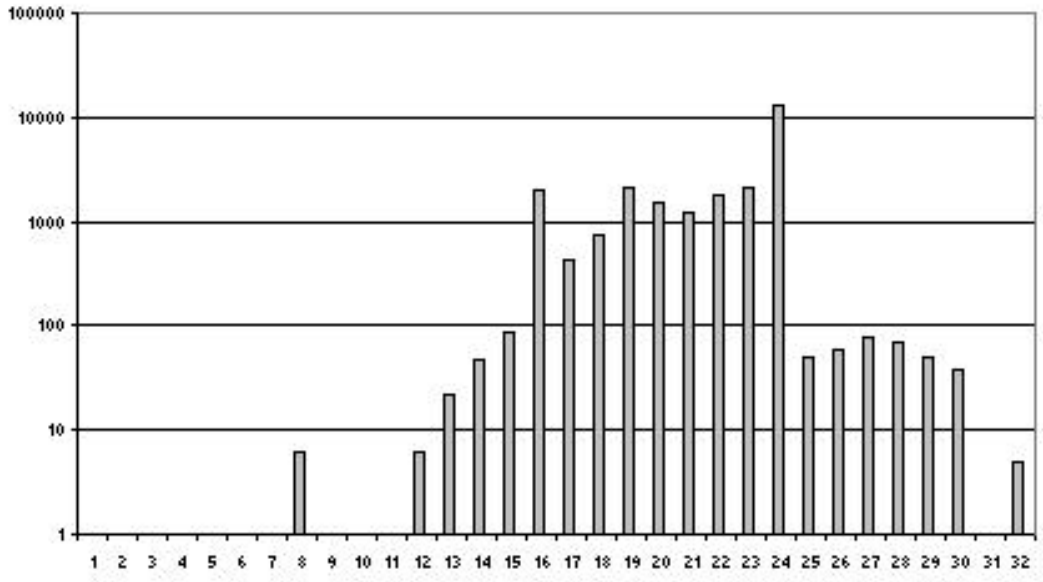
Nella tabella seguente è riportata la distribuzione media, divisa per router, delle route in base alla lunghezza dei prefissi; nell'ultima colonna viene indicata la media su tutte le tabelle a disposizione.

Lunghezza	Aads	Mae West	Mae East	Paix	Pacbell	Media
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	7	11	5	2	6	5.8
9	1	2	0	0	1	0.8
10	0	2	0	0	0	0.4
11	0	3	0	0	1	0.8
12	4	12	5	4	4	5.8
13	21	34	14	24	17	22
14	49	79	35	33	40	47.2
15	89	134	76	57	79	87
16	2419	2773	1743	1102	2161	2039.6
17	481	517	350	395	391	426.8
18	835	917	655	599	716	744.4
19	2351	2665	2103	1596	2089	2160.8

20	1774	1882	1376	1151	1453	1527.2
21	1390	1629	1193	762	1287	1252.2
22	2054	2303	1644	1156	1834	1798.2
23	2431	2748	1914	1292	2174	2111.8
24	15901	18240	11923	3392	15308	12952.8
25	70	31	22	89	36	49.6
26	88	39	20	92	59	59.6
27	124	54	19	110	82	77.8
28	114	51	9	86	77	69.4
29	88	43	4	56	65	51.2
30	64	32	3	41	46	37.2
31	0	0	0	0	0	0
32	10	5	1	1	7	4.8

Distribuzione delle rotte sulla lunghezza dei prefissi

Come si può notare dal grafico sottostante, la distribuzione delle route analizzate è in linea con quella riportata in letteratura. [18]



10.4 Tecniche per la misurazione dei tempi

La misurazione dei tempi di esecuzione viene fatta leggendo l'orario di sistema prima e dopo aver eseguito la procedura di cui si vuole conoscere la performance. Per La lettura del tempo di sistema è stata usata la chiamata $gettimeofday()$ che restituisce la misura in microsecondi ($10^{-6}sec$).

Come tempo di esecuzione delle procedure, è stata scelta la media di tre misurazioni indipendenti effettuate in condizioni, quanto piú possibile uguali, di carico minimo del sistema. Questo ci ha permesso di minimizzare l'errore assoluto e l'errore relativo commessi a causa della mancanza di controllo sullo scheduling dei processi.

10.5 Dimensione media dei vettori delle colonne

Le dimensioni del vettore delle colonne influenzano le prestazioni della procedura di lookup in quanto la probabilità di trovare un frammento di tale vettore nella

cache L1 è inversamente proporzionale alle sue dimensioni. Il vettore delle colonne è costituito, come già spiegato, da 2^h elementi dove, se gli indirizzi IP sono formati da m bit e $K\_SIZE = k$, $h = m - k$. Ogni elemento può essere memorizzato in un tipo **unsigned char** se il vettore delle colonne contiene fino a 256 elementi, altrimenti è necessario un tipo **unsigned short int**. Avendo scelto $K\_SIZE = 16$, anche nel caso pessimo, non è possibile che sia necessario un tipo di dimensione superiore.

Nella tabella seguente, per la struttura CDG, vengono riportati per ogni router il numero di tabelle per cui è stato possibile scegliere il tipo **unsigned char** (colonna char) ed il numero di tabelle per cui, invece, è stato necessario impiegare il tipo **unsigned short int**. (colonna int). L'ultima colonna contiene il numero medio di elementi differenti contenuti nel vettore delle colonne.

Router	Tot. Tabelle	char	int	Media
Aads	538	0	538	568
Mae West	618	0	618	397
Mae East	230	0	230	278
Paix	78	0	78	581
Pacbell	578	0	578	445

Distribuzione del vettore delle colonne nella struttura CDG

Nella tabella seguente vengono riportate le stesse indicazioni nel caso si sia impiegata la struttura SCDG. Si osserva immediatamente che nel caso della struttura CDG il vettore delle colonne deve essere sempre memorizzato nel tipo **unsigned short int**, invece nel caso della struttura SCDG è spesso possibile (nel 75.29% dei casi) impiegare il tipo **unsigned char** e dimezzare conseguentemente le dimensioni del vettore delle colonne. L'ultima colonna riporta la percentuale di tabelle per cui è stato possibile usare il tipo **unsigned char.**

Router	Tot. Tabelle	char	int	Media	% char
Aads	538	294	244	301	54.6
Mae West	618	540	78	209	87.3
Mae East	230	230	0	142	100.0
Paix	78	18	60	296	23.0
Pacbell	578	455	121	235	78.7

Distribuzione del vettore delle colonne nella struttura SCDG

Nel caso CDG, il valore della colonna *Media* rappresenta anche il valore mediamente assunto dalle dimensioni del vettore metrica. Nel caso SCDG, tale colonna rappresenta il valore medio assunto dalle dimensioni del piú grande tra la metrica della tabella sinistra e quella della destra. Dalle tabelle si verifica quindi che $|M| \approx 2\max(|M_1|, |M_2|)$ giustificando le considerazioni del paragrafo 6.3.

Le dimensioni del vettore dei bucket dipende da diversi fattori: il numero di bucket, la loro dimensione, il numero di bucket contenenti fault. Nella tabella seguente sono riportati i dati riguardanti i vettori dei bucket divisi per router. Nella terza colonna viene riportato il numero medio di bucket, nella colonna successiva viene si riporta il numero di elementi mediamente contenuti in un singolo bucket. La colonna successiva riporta il numero di bucket per cui si è riscontrata la presenza di un fault. L'ultima colonna riporta la dimensione media in byte del vettore dei bucket. Nel calcolo si consideri che un elemento del vettore dei bucket viene memorizzato in un tipo **short int** per cui il numero complessivo di elementi di tale vettore è dato dal numero di byte diviso due.

Router	Tot. tabelle	N° bucket	N° elementi	N^ fault	Byte
Aads	538	467	186	113	26526
Mae West	618	342	233	55	15120
Mae East	230	256	256	13	7670
Paix	78	452	157	120	32822
Pacbell	576	388	246	80	26728

Dimensioni del vettore dei bucket CDG o SCDG

Avendo scelto di usare la funzione identità per l'ordinamento delle colonne della tabella destra prima di unirle a quelle della tabella sinistra per la creazione della struttura SCDG, la procedura di creazione del vettore dei bucket è indipendente dalla struttura creata dal back end. Per questa ragione i dati appena presentati riguardo le caratteristiche del vettore dei bucket sono valide per entrambe le strutture.

La tabella seguente mostra le dimensioni in byte del segmento dei riferimenti, del segmento dei bucket e dell'intero vettore dei bucket. Si osservi che: il segmento dei riferimenti è sempre piú piccolo di un kilobyte. Questo non solo garantisce la sua costante residenza nella cache L1 del processore impiegato per la sperimentazione, ma anche per tutti i processori che abbiano cache di dimensioni inferiori.

Router	Riferimenti	Bucket	Totale
Aads	934	25592	26526
Mae West	684	14436	15120
Mae East	512	7158	7670
Paix	904	31918	32822
Pacbell	776	25952	26728

Dimensioni dei segmenti del vettore dei bucket

Si è già osservato che, avendo scelto $K\_SIZE = 16$, il vettore delle colonne contiene $2^{16} = 65536$ elementi. A seconda del tipo necessario per memorizzare ogni elemento, la dimensione di tale vettore può essere di 65536 byte o di 131072 byte. Ovviamente non sono ammesse dimensioni differenti. Nella tabella seguente viene fatto un raffronto tra le dimensioni medie dei vettori delle colonne per il CDG lookup, per l'SCDG lookup e per il bucket lookup. Tali dimensioni vanno intese come un valor medio e non come valore "lecito" per le dimensioni dei vettori. Nelle ultime due colonne viene riportato il risparmio percentuale del bucket lookup nei confronti del CDG e dell'SCDG.

Router	Bucket	CDG	SCDG	CDG %	SCDG %
Aads	26526	131072	95258	79.76	72.15
Mae West	15120	131072	73807	88.47	79.51
Mae East	7670	131072	65536	94.15	88.29
Paix	32822	131072	115948	74.95	71.69
Pacbell	26728	131072	79028	79.60	66.18

Raffronto tra il vettore delle colonne e quello dei bucket

Dall'analisi di tutte le tabelle si ottiene che il vettore dei bucket è mediamente di 21242 byte, per la struttura CDG il vettore delle colonne è di 131072 byte e per la struttura SCDG di 81617 byte.

Questo significa che il vettore dei bucket rispetto al vettore delle colonne risparmia il 83.80% di spazio usando la struttura CDG ed il 73.98% usando la struttura SCDG. Inoltre, il vettore delle colonne della struttura SCDG risparmia il 37.74% di spazio rispetto allo stesso vettore per la struttura CDG.

Si vedrà più avanti che il risparmio di spazio nella memorizzazione del vettore

delle colonne è fortemente collegato all'incremento delle prestazioni della procedura di lookup.

10.6 Prestazioni dei front end

Come già spiegato nel paragrafo 5.3 entrambe le realizzazioni del front end partendo dallo stesso input costruiscono le stesse strutture. Il raffronto tra le due realizzazioni consiste soltanto nel tempo di esecuzione. L'importanza di una realizzazione efficiente del front end risiede nel fatto che questa influenza il tempo di costruzione della tabella delle decisioni.

La realizzazione con trie, oltre a realizzare il vettore V_{RLE}, costruisce contemporaneamente anche la struttura dati necessaria per le operazioni di aggiornamento della tabella delle decisioni. Nel caso della realizzazione con quadruple del front end la costruzione della struttura per l'update richiede l'esecuzione di una procedura in piú rispetto alla sola creazione del vettore V_{RLE}. La scelta di non far costruire questa struttura comporta l'impossibilità di aggiornare la tabella delle decisioni. In entrambi i casi la realizzazione del front end mediante trie ha prestazioni migliori rispetto a quella che usa le quadruple.

Nella tabella seguente viene riassunto il tempo medio, in microsecondi, di esecuzione del front end con entrambe le realizzazioni. Nell'ultima colonna viene riportato in percentuale il risparmio di tempo della realizzazione con trie. I confronti sono stati fatti scegliendo di far realizzare ad entrambi i front end sia il vettore V_{RLE} che il trie necessario alla procedura di aggiornamento.

Router	Quadruple	Trie	Scarto %
Aads	80929	49929	38.31
Mae West	90731	54074	40.40
Mae East	61799	38505	37.69
Paix	40924	27592	32.58
Pacbell	73921	45994	37.78

Tempo di esecuzione delle realizzazioni del front end

Si conclude che in questo caso, sulla totalità delle tabelle di routing a disposizione di questa sperimentazione la realizzazione con trie è il 37.35% piú veloce della realizzazione con quadruple.

Nella tabella seguente viene fatto il raffronto tra i due front end nel caso in cui la realizzazione con quadruple non costruisca la struttura per l'aggiornamento.

Router	Quadruple	Trie	Scarto %
Aads	56347	49929	11.40
Mae West	63523	54074	14.88
Mae East	42657	38505	9.74
Paix	27449	27592	-0.52
Pacbell	51008	45994	9.83

Tempo di realizzazione del vettore V_{RLE}

Anche se il confronto tre le due realizzazioni appare iniquo la realizzazione con trie risulta comunque piú veloce in media del 9.06%.

10.7 Confronto tra struttura CDG ed SCDG

L'obiettivo della struttura SCDG è quello di riuscire a memorizzare la tabella delle decisioni in una struttura che impieghi meno memoria senza penalizzare le prestazioni della procedura di lookup.

Nella tabella seguente vengono riportate le dimensioni in byte della tabella delle decisioni della struttura CDG e di quella della struttura SCDG. L'ultima colonna mostra in percentuale lo spazio risparmiato con la struttura SCDG.

Router	CDG	SCDG	Scarto %
Aads	1692158	1118002	33.93
Mae West	1268868	873440	31.16
Mae East	750331	471898	37.10
Paix	1254038	759423	39.44
Pacbell	935746	560060	40.14

Raffronto dello spazio utilizzato dalle tabelle delle decisioni

Il risparmio medio di spazio per le tabelle delle decisioni SCDG rispetto a quelle CDG sulla totalità delle tabelle a disposizione ammonta al 35.45%.

L'impiego della struttura SCDG paga il prezzo di dover raddoppiare le dimensioni del vettore delle righe. Per ragioni di efficienza questo vettore va comunque implementato come un array di puntatori, ragion per cui, nel caso di questa sperimentazione in cui si è scelto $K\_SIZE = 16$, alla dimensione complessiva vanno aggiunti $4*2^{16} = 262144$ byte. La sperimentazione ha dimostrato che il numero delle righe della struttura SCDG nel 75.29% dei casi è abbastanza basso da consentire di usare il tipo **unsigned char** al posto del tipo **unsigned short int** dimezzando di fatto il numero di byte necessari alla memorizzazione del vettore delle colonne.

La tabella di seguito riportata mostra le dimensioni della struttura CDG e della struttura SCDG tenendo presente la variazione delle dimensioni del vettore delle righe e, in percentuale, del vettore delle colonne.

Router	CDG	SCDG	Scarto %
Aads	2085374	1737548	16.68
Mae West	1662084	1471535	11.47
Mae East	1143547	1061722	7.16
Paix	1647254	1399659	15.04
Pacbell	1328962	1163603	12.45

Raffronto dello spazio utilizzato dalle strutture per il lookup

Si nota che in questo caso il risparmio medio di spazio per l'intera struttura scende al 12.78%. Questo è dovuto al fatto che la maggiore necessità di spazio per il vettore delle righe ha un peso capace di ridimensionare notevolmente il risparmio di spazio della tabella delle decisioni. Però, con l'attuale trend di crescita delle tabelle di routing in internet, la dimensione delle strutture dovrebbe crescere abbastanza da ridurre notevolmente il peso dell'aumento delle dimensioni del vettore delle righe e portare il risparmio percentuale intorno ai valori ottenuti per la sola tabella delle decisioni. Pur non potendo fornire un'adeguata stima in quanto non si possiedono dati sufficienti, questa ipotesi sembra confermata.

10.7.1 Misurazione del tempo di esecuzione del back end

Nella tabella seguente vengono riportate le medie dei tempi d'esecuzione dei due back end. La costruzione della tabella SCDG impiega un tempo maggiore come era già stato spiegato nel paragrafo 6.4. L'obiettivo di tale struttura, però, non consiste

nell'avere una performance migliore a livello di tempo d'esecuzione ma bensí a livello di memoria risparmiata. L'ultima colonna riporta la percentuale di risparmio di tempo nella costruzione del back end CDG rispetto a quello SCDG.

Router	CDG	SCDG	Scarto %
Aads	70356	128218	45.13
Mae West	66029	126716	47.90
Mae East	55142	111577	50.58
Paix	58758	108869	46.03
Pacbell	57448	111528	48.50

Tempo di esecuzione delle realizzazioni del back end

10.8 Tempo di costruzione delle strutture CDG ed SCDG

In questa sezione si riportano i dati riguardanti la misurazione del tempo necessario per la costruzione della struttura CDG e di quella SCDG a partire dalla tabella di routing presa in input.

Nella tabella seguente sono riportati i tempi, misurati in microsecondi, necessari per la costruzione della struttura CDG. La seconda colonna riporta il tempo nel caso si utilizzi la realizzazione del front end mediante quadruple, la terza colonna riporta il tempo nel caso si usi il front end realizzato mediante trie. I tempi misurati si intendono comprensivi di quelli impiegati per la costruzione del trie necessario alla procedura di aggiornamento.

L'ultima colonna riporta la percentuale di tempo in piú necessaria nel caso si utilizzi il front end realizzato tramite quadruple.

Router	Quadruple	Trie	Scarto %
Aads	151285	120285	20.50
Mae West	156760	120103	23.39
Mae East	116941	93647	19.92
Paix	99682	86350	13.38
Pacbell	131369	103442	21.26

Tempo di costruzione della struttura CDG

Dalla tabella si verifica che per la totalità delle tabelle esaminate, l'introduzione della realizzazione con trie del front end fa ottenere un risparmio medio di tempo del 21.27%.

La tabella seguente è analoga a quella precedente con la differenza di essere riferita alla costruzione della struttura SCDG. Come si può notare, anche in questo caso la realizzazione mediante trie del front end riduce il tempo medio di costruzione della tabella anche se di una percentuale minore. Questo è dovuto alla maggiore complessità del back end SCDG.

Router	Quadruple	Trie	Scarto %
Aads	209147	178147	14.83
Mae West	217447	180790	16.86
Mae East	173376	150082	13.44
Paix	149793	136461	8.91
Pacbell	185449	157522	15.06

Tempo di costruzione della struttura SCDG

Sulla totalità delle tabelle esaminate l'introduzione del front end realizzato mediante trie fa ottenere un risparmio percentuale di tempo pari al 15.12%.

10.9 Prestazioni delle procedure di lookup

La tabella seguente mostra le prestazioni delle procedure di lookup per la struttura CDG e per la struttura SCDG. In entrambi i casi vengono confrontate le prestazioni della procedura standard e di quella qui proposta di bucket lookup.

I dati riportati sono misurati in microsecondi e riguardano il tempo impiegato per effettuare un milione di lookup.

Le colonne CDG ed BCDG riportano le misurazioni dei tempi effettuati sulla struttura CDG effettuando prima le misurazioni con la procedura standard e poi utilizzando il bucket lookup. La colonna successiva riporta il risparmio percentuale di tempo utilizzando il bucket lookup.

Le colonne SCDG ed BSCDG riportano i tempi misurati sia con la procedura standard che con il bucket lookup per la struttura SCDG. La colonna successiva riporta il risparmio di tempo in percentuale della procedura di bucket lookup.

L'ultima colonna riporta in percentuale il confronto tra la struttura CDG e la struttura SCDG utilizzando la procedura standard di lookup.

Router	CDG	BCDG	%	SCDG	BSCDG	%	%
Aads	71410	56008	21.57	62989	58666	6.87	11.80
Mae West	69891	44729	36.01	51917	48556	6.48	25.72
Mae East	63215	34348	45.67	42577	37689	11.49	32.65
Paix	63495	50487	20.49	62160	51122	17.76	2.11
Pacbell	60549	44999	25.69	46714	47345	-1.35	22.85

Prestazioni delle procedure di lookup sulle strutture CDG ed SCDG

Come è stato messo in evidenza dai risultati presentati in questo capitolo, spesso l'impiego della struttura SCDG ha permesso di utilizzare il tipo **unsigned char** per

memorizzare il vettore delle colonne. Della procedura di lookup standard esistono due copie identiche a livello logico, ma differenti per il tipo degli elementi del vettore delle colonne. Ovviamente per effettuare il lookup standard è stata scelta la procedura che usa il tipo appropriato.

L'ultima colonna mette in evidenza che la procedura di lookup standard sulla struttura SCDG è piú rapida, in media del 21.11%, rispetto alla stessa procedura riferita alla struttura CDG. Questo è dovuto alle seguenti ragioni:

- minore occupazione di memoria della tabella delle decisioni
- minore occupazione del vettore delle colonne

La prima fa aumentare le probabilità di trovare il frammento di tabella nella cache L2, la seconda fa incrementare la probabilità di trovare il frammento richiesto del vettore delle colonne nella cache L1.

La tabella seguente mostra il tempo medio, misurato in nanosecondi, del singolo lookup. La prima colonna riporta il nome del router. Nella seconda e terza colonna, con riferimento alla struttura CDG si riportano i tempi impiegando rispettivamente la procedura standard ed il bucket lookup. Le ultime due colonne fanno, invece, riferimento alla struttura SCDG.

Router	CDG	BCDG	SCDG	BSCDG
Aads	71	56	62	58
Mae West	69	44	51	48
Mae East	63	34	42	37
Paix	63	50	62	51
Pacbell	60	44	46	47

Tempo medio per un lookup in nanosecondi

La tabella seguente mostra i milioni di lookup al secondo effettuato. Il significato delle colonne è lo stesso di quello della tabella precedente.

Router	CDG	BCDG	SCDG	BSCDG
Aads	14.00	17.85	15.87	17.04
Mae West	14.30	22.35	19.26	20.59
Mae East	15.81	29.11	23.48	26.53
Paix	15.74	19.80	16.08	19.56
Pacbell	16.51	22.22	21.40	21.12

Milioni di lookup per secondo

L'obiettivo del CDG lookup era quello di riuscire ad effettuare qualche milione di lookup al secondo in modo di poter garantire ai router prestazioni nell'ordine del gigabit al secondo. La procedura di bucket lookup è stata qui proposta per provare ad ottenere prestazioni migliori rispetto a quelle del CDG lookup. Dall'analisi delle tabelle di tutti i router si osserva come sia stato possibile passare da 15.07 milioni di lookup per secondo a 21.79 milioni con un incremento rispetto alla procedura standard del 44.59%

10.10 Prestazioni della procedura di update

La seguente tabella mostra le prestazioni della procedura di aggiornamento. Tali prestazioni, come è già stato spiegato, dipendono fortemente dalle prestazioni del back end. Questo giustifica il fatto che l'aggiornamento della struttura SCDG risulta essere piú oneroso.

La seconda e la terza colonna mostrano il tempo medio in microsecondi necessario

per la singola operazione di aggiornamento. Le ultime due colonne mostrano il numero medio di aggiornamenti che è possibile effettuare in un secondo.

Router	CDG	SCDG	CDG	SCDG
Aads	95703	153565	10.44	6.51
Mae West	92895	153582	10.76	6.51
Mae East	74505	130940	13.42	7.63
Paix	72875	122986	13.72	8.13
Pacbell	80529	134609	12.41	7.42

Tempo in microsecondi per una operazione di aggiornamento

Si osservi che non viene fatta differenza tra l'operazione di inserimento e quella di cancellazione, in quanto i test hanno mostrato che entrambe impiegano circa gli stessi tempi.

Capitolo 11

Conclusioni

Per la soluzione del problema dell'IP address lookup, per diversi anni, sono state impiegate le stesse tecniche utilizzate per risolvere il problema dello shortest matching prefix. Con la costante crescita di internet e del traffico queste soluzioni non sono piú state in grado di fornire prestazioni adeguate alle necessità.

Ci si è trovati a credere che il problema dell'IP address lookup non fosse risolvibile in tempi accettabili se non con l'impiego di hardware specifico. La smentita di questa convinzione è arrivata con la presentazione di alcuni algoritmi che riuscivano ad ottenere prestazioni significativamente migliori sfruttando alcune caratteristiche dei processori, ma concentrandosi comunque sul problema del longest matching prefix.

Gli autori del CDG lookup sono stati i primi a cambiare radicalmente l'approccio al problema. Infatti, invece di cercare una soluzione per il problema generale e poi applicarlo al problema dell'IP address lookup, hanno cercato una soluzione che funzionasse bene per il problema specifico. Per far ciò, essi hanno effettuato un'analisi delle tabelle BGP per piú di tre anni.

Questo approccio, risultato vincente, è stato seguito piú recentemente dagli autori del retrie[3] descritto nella sezione 3.2.3.3. Il CDG lookup ed il retrie sono attualmente ritenuti i piú performanti algoritmi per l'IP address lookup.

11.1 Metodica

La scelta di analizzare i dati reali e, partendo da questi, cercare delle soluzioni ad hoc è da intendere come chiave del successo del CDG lookup. In questa tesi si è scelto di utilizzare lo stesso approccio per cercare di ottenere prestazioni migliori. Infatti, dall'analisi delle proprietà dal vettore delle colonne è stato possibile progettare il vettore dei bucket e la procedura di bucket lookup che rappresentano i risultato piú importanti di questa tesi.

L'analisi delle tabelle delle decisioni, invece, ha suggerito un metodo naturale per costruire una struttura simile che è possibile memorizzare in spazi piú ridotti ed il cui accesso è risultato naturalmente piú rapido grazie alla possibilità di utilizzare un tipo piú piccolo per memorizzare il vettore delle colonne.

Le ottimizzazioni proposte per la procedura di bucket lookup sono state realizzate in maniera tale da essere indipendenti dal tipo di processore e dalla dimensione della cache L1. Il segmento dei riferimenti, infatti, non supera mai il kilobyte e quindi possibile memorizzarlo in una singola pagina di cache L1 sia nel caso di processori AMD che Intel.

11.2 Risultati ottenuti

Il primo obiettivo che ci si è posti in questa tesi è stato quello di realizzare la procedura di costruzione della tabella CDG e la procedura di lookup cosí come proposto nell'articolo originale degli autori [4].

CI si è resi subito conto la procedura di costruzione della tabella CDG era logicamente divisa in due parti: il front end ed il back end.

E' stata cercata una realizzazione piú veloce del front end che costruisse la struttura per l'update contestualmente al vettore V_{RLE}. La soluzione che si è riusciti a

trovare è piú veloce del 37.35% rispetto a quella originale se entrambe realizzano anche il trie necessario per l'update. Rispetto alla versione originale, ovvero quella con quadruple, se si è interessati solo a che questa realizzi il vettore V_{RLE}, la soluzione proposta è comunque piú veloce del 9.06%. Inoltre in questo caso la procedura originale non fornisce il supporto per l'aggiornamento della struttura che andrebbe quindi ricostruita interamente ad ogni aggiornamento.

Nel realizzare il back end, oltre a cercare una soluzione quanto migliore possibile dal punto di vista del tempo di esecuzione, si è cercata una soluzione che riuscisse a ridurre le dimensioni della struttura. La possibilità di ridurre tali dimensioni è stata ritenuta prioritaria rispetto alla velocità di esecuzione. Infatti, una struttura piú compatta garantisce delle prestazioni di lookup migliori. La conferma è arrivata dalla sperimentazione la quale ha mostrato che, utilizzando la procedura standard di lookup, l'accesso alla struttura SCDG è piú veloce rispetto a quello alla struttura CDG del 21.11%.

Oltre al pregio di un tempo di lookup piú basso, la struttura SCDG ha il pregio di una minore occupazione di memoria. La sola tabella delle decisioni, nel caso SCDG, risulta essere piú piccola del 35.45%. La percentuale di spazio mediamente risparmiato dall'intera struttura SCDG rispetto alla struttura CDG risulta invece pari al 12.78%. Questo, come già discusso, è dovuto alla necessità di raddoppiare le dimensioni del vettore delle righe. Il contributo negativo dato da questa necessità risulta essere talmente pesante da far diminuire i benefici della compressione della tabella delle decisioni di piú del 50%. Con la crescita delle tabelle di routing questo fenomeno dovrebbe tendere a diminuire drasticamente. In ogni caso, la dimensione della struttura SCDG risulta inferiore, rispetto a quelle della struttura CDG, anche con le tabelle di routing attuali.

Per quanto riguarda i tempi di costruzione dell'intera struttura, l'introduzione

di un front end piú performante ha fatto in modo che la costruzione della struttura CDG sia piú veloce del 21.27% rispetto alla realizzazione precedente. In questo modo il maggior tempo richiesto per la costruzione della struttura SCDG è stato arginato ed è mediamente del 16.21%.

Non esistendo una procedura di aggiornamento prima di questa tesi, un raffronto di prestazioni risulta complicato. Quello che veniva fatto era di ricostruire per intero la struttura CDG. Il tempo medio per effettuare questa operazione si ottiene sommando il tempo di esecuzione del front end realizzato con le quadruple (Solo il tempo di costruzione del vettore V_{RLE}) al tempo di esecuzione del back end CDG. Un aggiornamento veniva fatto mediamente in 117606 microsecondi che equivalgono a 8.50 aggiornamenti al secondo.

La procedura di update qui proposta fa mediamente 11.55 aggiornamenti al secondo della struttura CDG. Questo equivale ad un incremento delle prestazioni del 35.88%. Il confronto con l'aggiornamento della struttura SCDG non ha senso in quanto si confronterebbe l'aggiornamento di strutture differenti.

I dati piú significativi e piú importanti per questa tesi, riguardano i tempi di lookup. La sola introduzione della struttura SCDG migliora le prestazioni della procedura standard del 21.11%. L'introduzione del vettore dei bucket e della procedura di bucket lookup fa passare da una media di 15.07 milioni di lookup al secondo ad una media di 21.79 milioni. L'incremento medio di prestazioni della procedura di bucket lookup, rispetto alla procedura standard su struttura CDG, raggiunge quindi il 44.59%.

Bibliografia

[1] S. Bellovin, R. Bush, T. Griffin, and J. Rexford. Slowing routing table growth by filtering based on address allocation policies, June 2001. http://www.research.att.com/jrex/.

[2] Maurizio Bonuccelli. Routing. In *Appunti del corso di sistemi per l'elaborazione dell'informazione:Reti di calcolatori*, pages 47–49, 1998.

[3] Adam L. Buchsbaum, Glenn S. Fowler, Balachander Krishnamurthy, Kiem-Phong Vo, and Jia Wang. Fast prefix matching of bounded strings, 2003.

[4] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi. IP address lookup made fast and simple. In *European Symposium on Algorithms*, pages 65–76, 1999.

[5] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, 1997.

[6] V. Fuller, T. Li, J. Yu, and K. Varadhan. Classless inter-domain routing (cidr) an address assignment and aggregation strategy. RFC 1519, sep 1993.

[7] Cyril Gavoille. A survey on interval routing. *Theoretical Computer Science*, 245(2):217–253, 2000.

[8] G. Huston. Analyzing the Internet's BGP Routing Table. *The Internet Protocol Journal*, 4(1), mar 2001.

[9] Y. Choi I. Lee, K. Park and S.K. Chung. A simple and scalable algorithm for the ip address lookup problem. *13th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, pages 209–220, July 2002.

[10] Nerit inc. Ipma project, 2002.

[11] F. Thomson Leighton. Introduction to parallel algorithms and architecture: arrays - trees - hipercubes, 1992.

[12] Anthony J. McAuley and Paul Francis. Fast routing table lookup using CAMs. In *INFOCOM (3)*, pages 1382–1391, 1993.

[13] Pietro Piram and Francesco Romani. Codifica in assenza di rumore. In *Appunti di teoria dell'informazione*, pages 39–49, 2001.

[14] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). RFC 1771, IETF, March 1995.

[15] M. A. Ruiz-Sanchez, Ernst W. Biersack, and Walid Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, March/April 2001.

[16] G. Bini A. Frangioni G. Gallo S. Pallottino M. Scutellà. Grafi e reti di flusso. In *Appunti di Ricerca Operativa*, pages 110–122, 2001/02.

[17] Cisco system Inc. Border gateway protocol.

[18] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing lookups. In *Proceedings of SIGCOMM '97*, pages 25–36, September 1997.