ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

TITOLO DELLA RELAZIONE FINALE

Analisi efficiente di tracce prodotte da un simulatore di protocolli di rete

Relazione finale in

Reti di Calcolatori

Relatore

Gabriele D'Angelo

Presentata da

Ilaria Ciavatti

Sessione II sessione

Anno Accademico 2012/2013

INTRODUZIONE

Questa tesi consiste nello sviluppo di codice per un applicativo esistente. Tale applicativo si occupa di simulazione di protocolli di rete su varie topologie di rete.
La parte di cui si occupa questa tesi è la fase di analisi dei file di traccia generati. Tale fase è una delle tre fasi dell'applicativo. I file di traccia contengono informazioni che permettono di stimare due grandezze, copertura e ritardo. Per copertura si intende una media fra tutte le coperture. Ogni copertura sarà relativa ad uno specifico nodo della rete. Per copertura di uno specifico nodo della rete si intende la parte di messaggi arrivati a quel nodo relativamente a tutti i messaggi generati all'interno della rete.
Per ritardo si intende la media del numero di salti che avvengono per arrivare ai vari nodi che compongono la rete.
I valori calcolati, mediante le informazioni contenute nei trace file, vengono poi memorizzati su dei file.
Oltre al calcolo di queste specifiche grandezze, la tesi include anche una possibile soluzione relativa al parallelismo dei processi. Tale soluzione mira ad avere un numero, stabilito in base alla memoria principale libera, di processi che vengono eseguiti in maniera parallela.
In totale vi sono otto capitoli. Il primo riguarda in linea generale l'applicativo per il quale si produce codice. Si sottolinea inoltre la rilevanza di raccogliere dati sulla simulazione dei vari protocolli al fine di calcolare le due grandezze. Tali grandezze danno l'idea di quanto sia "buono" l'algoritmo su cui si basa un certo protocollo.
Nel capitolo due si spiega quali siano le funzionalità da implementare, viene esposto l'algoritmo che si è pensato per fornire tali funzionalità.
Il capitolo tre si occupa di capire quanto l'algoritmo trovato sia efficiente, in questo capitolo viene esposto il calcolo dei costi e prima di questo, le strutture che sono state usate.
Il capitolo quattro esamina alcune parti del codice preesistente per comprendere quali siano i valori di entrata e di uscita e provvedere il

nuovo codice dei medesimi, altrimenti non si potrebbe integrare il nuovo codice con il restante.
Il progetto di questa tesi è composto da tre file. In uno vi è la parte relativa alla creazione di strutture contenenti le informazioni che devono essere eleborate per poter ottenere le grandezze sopracitate.Perciò l'output di questo primo file consisterà in tali grandezze.
Il secondo file contiene il codice necessario ad eseguire l'eseguibile del primo file, fornendogli i giusti parametri di input.
Il terzo file di fatto è uno dei file presenti nell'applicativo modificato in un particolare punto. Tale modifica consiste nel calcolo di una particolare variabile necessaria.
Il capitolo cinque si occupa di illustrare le funzioni relative al primo file.
Il capitolo sei si occupa di illustrare il codice relativo al secondo file.
Nel capitolo sette viene illustrato il codice del terzo file.
Il capitolo otto motra l'output dell'algoritmo creato.
Alla fine di questa tesi vi sono le conclusioni che contiengono un riassunto di quanto fatto e tutti i possibili cambiamenti da effettuare per migliorare il codice.

CAPITOLO 1

LUNES

Paragrafo 1.0

LUNES è l'applicativo per il quale si produce codice.
LUNES permette di simulare reti complesse, composte da un alto numero di nodi.
L'obbiettivo di LUNES è capire il funzionamento delle reti complesse. L'interesse è relativo a più ambiti, sono infatti stati riconosciuti dei pattern o specifici schemi che si presentano in campo tecnologico e biologico.
La rappresentazione di una rete complessa è un grafo o topologia di rete. Si tratta di un insieme di nodi collegati fra loro da collegamenti detti link, ogni nodo individua un elemento dotato di specifiche funzionalità, i collegamenti rappresentano la connessione fisica fra i nodi.
La comunicazione fra due nodi avviene tramite il link che li collega, se è presente.
Ciò non significa che dato un messaggio, o un'entità con contenuto informativo, possa essere passata solo ai nodi subito adiacenti al nodo che ha creato tale messaggio. Significa che tale informazione arriverà attraverso il passaggio dell'informazione da nodo a nodo tramite i collegamenti esistenti. Ovviamente ci sono anche casi in cui un messaggio non può raggiungere un determinato host, un esempio è dato da una topologia in cui un nodo non è collegato a nessun altro nodo.

CLUSTERING MECHANISM

Paragrafo 1.1

LUNES è un “Agent -based Large Unstructured NEtwork Simulator”.
“Agent -based” si riferisce al paradigma di programmazione , prevede cioè un partizionamento in insiemi di SME (Simulation Model Entities). Ognuno di questi SME interagisce con gli altri . Tali entità vengono raggruppate a seconda delle interazioni. Quindi verranno creati dei 'gruppi' di entità che interagiscono frequentemente fra loro, tale meccanismo è chiamato “clustering mechanism”. Quello che avviene nella pratica è che gli agenti appartenenti ad uno stesso cluster verranno eseguiti dalla stessa unità di esecuzione in modo da ridurre i costi dovuti alla comunicazione fra gli SME. Tale costo è infatti maggiore per quanto riguarda la comunicazione fra SME che preveda l'uso di più CPU (unità di esecuzione) o che preveda l'interazione fra più host (infatti l'esecuzione può avvenire in sistemi distribuiti, quindi coinvolgendo più hosts nella simulazione).
L'obbiettivo di LUNES è la simulazione di protocolli di rete su grafi con varie e differenti topologie, tali grafi vengono generati da altri tool. Infine LUNES si occupa dell'analisi dei risultati ottenuti durante la simulazione.
Quindi si possono distinguere tre fasi:

- LUNES importa topologie di grafi, precedentemente create da appositi tool;
- viene simulato un certo protocollo di rete;
- si tiene traccia di alcune informazioni, le quali risiedono su trace files o file di traccia che vengono quindi analizzati.

La parte che verrà approfondita in questa tesi è l'analisi dei trace file.[1]

CAPITOLO 2

LINGUAGGIO DI PROGRAMMAZIONE

Paragrafo 2.0

“Un linguaggio di programmazione è un linguaggio formale eseguibile su un calcolatore.
Gli algoritmi sono spesso implementati come programmi per un calcolatore, ma possono essere tradotti in rete neurale, in circuito elettrico o in apparecchio meccanico.
L'analisi degli algoritmi è una disciplina dell'informatica, spesso affrontata teoricamente (senza riferimenti a linguaggi di programmazione o ad implementazioni).
Una definizione formale di algoritmo lo identifica come procedura che può essere implementata (espressa nel linguaggio eseguibile) su una macchina di Turing, conducendola in uno stato finale.”[2]
Un linguaggio formale è un linguaggio con sintassi e semantica definite in maniera molto precisa. Un linguaggio eseguibile da un calcolatore non sarebbe utile se fosse ambiguo. Si avrebbe la non prevedibilità delle istruzioni descritte.
La citazione continua parlando della definizione di algoritmo. L'algoritmo è quindi una procedura [3], specifica le istruzioni, i passi che dovranno essere svolti dalla macchina. Per esprimere un algoritmo non è richiesto l'uso di uno specifico linguaggio è possibile usare anche il linguaggio naturale. L'ambiguità del linguaggio naturale può però portare l'algoritmo scritto dall'autore, a non essere compreso da terzi. Per questa motivazione viene usato uno pseudolinguaggio. Un linguaggio formale, in cui i termini usati acquisiscono un ben preciso significato, che però non è un linguaggio di programmazione.

PAROLE CHIAVE

Paragrafo 2.1

In questo paragrafo verranno descritte tutte le parole chiave utilizzate all'interno dello pseudocodice, in modo da rendere più comprensibile la lettura dell'algoritmo.

HOPS: letteralmente “balzi” indicano il numero di collegamenti che vengono usati dal messaggio. Tale messaggio parte quindi da un nodo origine A per finire in un nodo destinazione B; durante tale tragitto il numero di collegamenti attraversati viene chiamato numero di hops .

BST: per esteso Binary Search Tree, albero di binario di ricerca, successivamente tale argomento verrrà esposto meglio, basti sapere che è una struttura dati per la memorizzazione, nel caso specifico, dei nodi.

Flusso di controllo: Nell'ambito della programmazione un flusso di controllo è la sequenza di istruzioni eseguite dal calcolatore durante l'esecuzione di un particolare eseguibile. Tale eseguibile è la traduzione in linguaggio macchina di un programma. Nel caso di questa tesi, il file sorgente in C verrà tradotto in linguaggio macchina, il quale verrà eseguito dal calcolatore. Le istruzioni eseguite ed il loro ordine costituiscono il flusso di controllo. Per “modellare” tale flusso si usano i controlli condizionali. Il flusso di controllo può essere interpretato tramite un diagramma.

Controlli condizionali: I controlli condizionali permettono al programmatore di definire l'ordine di esecuzione delle istruzioni. Infatti al verificarsi di particolari condizioni il flusso verrà “direzionato” da una parte piuttosto che dall'altra. Il fatto che il flusso venga direzionato significa che verranno eseguite determinate istruzioni piuttosto che altre.

Il direzionamento del flusso avviene in base alla verifica di particolari condizioni.

Nello pseudocodice venfono usati i controlli condizionali: while[5] ed if-else[6]. Il while è un controllo fondamentale e rappresenta un “ciclo”, letteralmente significa “mentre”. Sintassi: While(?), significa che mentre

è verificata la condizione “? “ bisogna continuare a iterare, ovvero a svolgere le istruzioni contenute all'interno del while. Le istruzioni relative ad un controllo sono racchiuse nelle parentesi graffe {} subito successive al controllo. Tali parentesi mancheranno solo nel caso in cui vi sia una sola istruzione nel controllo. Perciò se dopo il controllo appare una sola istruzione questa è assimilabile come istruzione contenuta pure in mancanza di parentesi graffe che la contengano.
L'if-else, o salto condizionale, leteralmente “se – altrimenti” consente di valutare un'esspressione “?”, la condizione dell'if(?), se tale condizione è verificata il flusso di esecuzione entra nel blocco di istruzioni contenute fra le parentesi graffe dell'if saltando quelle contenute nell'else. Nel caso in cui la condizione non sia vera vengono eseguite le istruzioni nell'else e saltate quelle dell'if (intendendo sempre sia nel caso dell'else che dell'if le istruzioni contenute nelle parentesi graffe subito successive alle parole chiave, if o else).
Assegnamento: L'assegnamento consiste nel mettere un valore contenuto nella variabile A , all'interno della variabile B.
B=A;
Le variabili sono spazi in memoria che contengono valori e sono manipolabili dal programmatore attraverso l'assegnamento.
Si noti che all'interno delle verifiche di una condizione l'uguale è doppio “==” , tale operazione di confronto fra due operatori non costituisce alcun assegnamento.

I FILE DI TRACCIA

Paragrafo 2.2

Il lavoro di questa tesi verte su quella che è la parte di analisi dei file di traccia all'interno di LUNES.
Lo scopo dell'analisi dei files di traccia è avere dei dati che permettano di stabilire la qualità del funzionamento dei vari protocolli di gossip sulle topologie di rete.
I files di traccia saranno generati durante le simulazioni di reti. Tali reti

saranno composte da centinaia di nodi. Da ciò deriva una grande quantità di informazioni da mantenere in memoria secondaria e da studiare. Lo studio di tali files avviene durante la fase di analisi dei trace files, come descritto in precedenza nel capitolo due.
Quindi in sintesi , la prima parte del progetto consiste nel calcolare delle grandezze, che sono il ritardo o delay e la copertura o coverage, attraverso l'analisi dei dati all'interno dei trace files.
Vediamo quindi come sono fatti i file di traccia e successivamente qual'è il significato delle due metriche menzionate, per capire cosa ci si aspetta dall'algoritmo di analisi dei files di traccia.

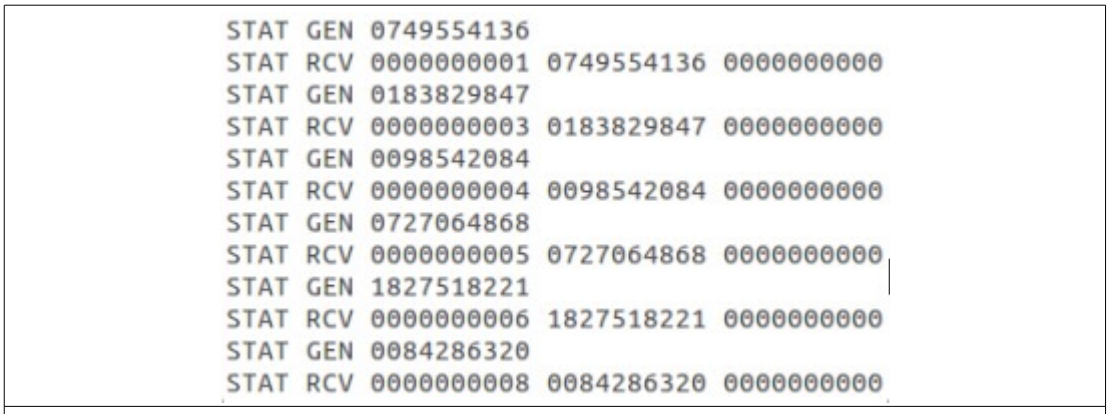


Illustrazione 2.0

Nell'illustrazione 2.0 è presente una (minima) parte di un file di traccia.
Per capire il file di traccia ci serviamo della seguente descrizione [4] :

Operation	Parameter(s)	Description	Parameters
STAT GEN	parameter	*A new message has been generated.*	The parameter is an integer value that identifies the newly generated message. All the message

			identifiers are chosen at random.
STAT RCV	parameter1, parameter2, parameter3 [parameter4]	*A new message has been received by a given node.*	All the parameter are integer values. The parameter1 is the identifiers of the receiver. The parameter2 is the identifier of the message that has been received. The parameter3 is the number of hops to arrive at this node.There is a parameter4 only if the message is not locally generated (i.e it has been delivered by a neighbor). The meaning of this last parameter is the residual TTL of the delivered messages.
STAT MSG	parameter	*Total number of delivered*	The parameter is a integer value

		messages.	that identifies the total number of messages delivered in this run.

Quindi ogni riga del trace file si interpreta attraverso l'uso della tabella posta qui sopra.
Ciò significa che una riga del file di traccia che presenti, inizialmente, la combinazione di parole STAT GEN ha un significato preciso ovvero "è avvenuta la generazione di un messaggio", il nome del messaggio generato è il parametro presente nella riga stessa successivamente allo STAT GEN.
Ciò significa che per sapere il numero di messaggi generati si devono contare le righe di generazione di messaggio. Infatti ogni messaggio viene generato una sola volta. Tale messaggio verà quindi propagato seguendo le modalità dello specifico protocollo di gossip.
Per quanto riguarda le rige del trace file che presentano la combinazione di parole STAT RCV , tali righe tengono traccia della ricezione di un messaggio da parte di uno specifico nodo ed il numero di hop che il messaggio ha effettuato prima di arrivare a tale nodo. L'identificativo del messaggio è il secondo parametro presente nella riga , il primo è l'identificativo del nodo ricevente mentre il terzo è il numero di hop.
Le grandezze che fanno parte dell'analisi sono la copertura ed il ritardo.
La copertura è una grandezza che indica la qualità della distribuzione dei messaggi sulla rete di no. Una copertura del 100% implica che tutti i messaggi siano stati ricevuti da tutti i nodi. La seconda grandezza, il ritardo, indica quanto impiega mediamente un nodo a ricevere un messaggio.

IDEA

Paragrafo 2.3
Per ogni dubbio sui termini usati in questo paragrafo ed in quello

successivo consultare il paragrafo 2.1.
L'idea proposta per il calcolo efficiente delle due grandezze, ritardo (delay) e copertura (coverage) si basa sulle seguenti osservazioni.
I dati che sono necessari al fine di calcolare le due grandezze sono le righe del trace file contenenti le informazioni di ricezione dei messaggi e le righe di generazione dei messaggi. Quindi le righe del file che iniziano con “STAT GEN” o con “STAT RCV”.
Le righe “STAT GEN” permettono di contare i messaggi , quelle “STAT RCV” forniscono i dati relativi alla ricezione dei messaggi (per la copertura si ha necessità di conoscere tali informazioni) ed i dati relativi agli hop effettuati da un certo messaggio per arrivare ad uno specifico nodo (tale informazione sarà invece utilizzata per il calcolo del ritardo).
Si immagini quindi di avere la necessità di memorizzare tali informazioni per poterle elaborare nella fase di calcolo di copertura e ritardo.
Il modo utilizzato in questa tesi è la creazione di una matrice che contenga tutti questi dati.
In tale matrice le righe rappresenteranno i vari nodi mentre le colonne i messaggi. Quindi data la riga 0 della matrice corrispondente al nodo identificato dalla stringa “xxxxxxxxxx”, si memorizzaranno le informazioni di ricezione ed il numero di hops del messaggio ripettivo ad una determinata colonna. In sintesi data la coordinata (x,y) della matrice, la x indicherà uno specifico nodo, la y uno specifico messaggio ed all'interno di tale cella si troverà un particolare valore che avvisi che tale messaggio non è stato ricevuto dallo specifico nodo, oppure un valore preciso che indichi il numero di hops (minimo) occorso per ricevere tale messaggio. Il fatto che sia stato puntualizzato che il numero di hop è il minimo è dovuto al fatto che ci sono più valori fra i quali 'scegliere' infatti uno stesso messaggio può essere ricevuto più volte da un nodo.
A questo punto ogni riga conterrà le informazioni relative a ciascun nodo , quindi per sapere qual'è il ritardo medio per arrivare a tale nodo , basta sommare i valori che sono i numeri di hops, presenti in quella determinata riga , tranne il valore che indica che il messaggio non è stato ricevuto e dividere per il numero di messaggi totale. Il numero di

messaggi totale si calcola come già detto dal trace file, contanto le righe STAT GEN. La copertura di un messaggio si calcola contando il numero di nodi che l'hanno ricevuto, quindi scorrendo la colonna relativa al messaggio e contando tutti i valori differenti dal valore che simboleggia non ricevuto. Si anticipa che all'interno della matrice come valore che simboleggi la non ricezione è stato scelto -1 in quanto un numero di hops è per forza positivo e non ci sono quindi rischi di ambiguità.

DESCRIZIONE DELL'ALGORITMO IN PSEUDOCODICE

Paragrafo 2.4

Descrizione della prima fase dell'algoritmo: nella prima fase si scorre il trace file per contare sia il numero di nodi che il numero di messaggi.

Paragrafo 2.4.0

Chiamiamo prima fase la fase in cui si stabiliscono le dimensioni della matrice.
Il numero di nodi può essere stabilito solo se si memorizzano man mano i nodi incontrati per evitare di contarli più volte (si ricorda che i nodi sono presenti nelle righe del file STAT RCV , a questo fine si userà un Binary Search Tree . Ogni volta che si incontrerà una riga STAT RCV verrà effettuato un controllo sui nomi di nodi presenti all'interno del BST. Se non viene trovato il nodo che si cerca si può procedere all'inserimento. Al termine della lettura del trace file verranno contati il numero di nodi presenti nel BST. Il numero di nodi ed il numero di messaggi saranno quindi le due dimensioni della matrice che deve essere allocata per poter mantenere in memoria, ad ogni posizione (x,y), il valore che mette in relazione il nodo x con il messaggio y, per quanto descritto in precedenza.
La prima fase è la seguente:

```
fine_file_traccia=0;
numero_messaggi=0;
```

```
numero_nodi=0;
while(!fine_file_traccia)
{
  per ogni riga del file
      if(la riga inizia con “STAT GEN”)
         numero_messaggi=numero_messaggi+1;
      else
         if(la riga inizia con “STAT RCV”)
         {
               presente=0;
               nodo=parameter1;
               leggere nel BST se sia o meno presente nodo (*)

               if (!presente)
               {
                     inserimento nel bst di nodo;
                     numero_nodi=numero_nodi+1;
               }

         }
}
```
(*) nel caso sia presente la variabile “presente” assumerà valore 1.

Paragrafo 2.4.1

La fase numero due invece consiste nell'allocare la matrice, ovvero “nell' occupare” posto in memoria per la matrice con le dimensioni sopra specificate nella prima parte dell'algoritmo
Quindi semplicemente:

Si alloca spazio in memoria per una matrice di numero_righe=numero_nodi e numero_colonne=numero_messaggi.

Paragrafo 2.4.2

La fase numero tre consiste , avendo la matrice allocata grazie alla fase due, nel riempirla con le informazioni necessarie all'analisi. Quindi si rileggerà il trace file o file di traccia e si inseriranno, per ogni messaggio ricevuto da un certo nodo, la relativa informazione nella giusta cella. Quindi ogni coordinata (x,y) verrà riempita del numero di hops necessari al messaggio y as essere ricevuto dal nodo x o in alternativa la cella conterrà, come già scritto, -1 che non simboleggia il numero di hops ma la mancata ricezione. Fase tre:

```
fine_file=0;
while(!fine_file)
{
        per ognuna delle R righe del file
        if(la riga inizia con “STAT GEN”)
        {
                messaggio=parameter;
                cerca messaggio nella matrice (prima riga contenente i
                nomi dei messaggi):
                if(messaggio non è presente nella matrice)
                {
                        inserimento del messaggio nella matrice;
                }
         }
         else
        {
               if(la riga inizia con “STAT RCV”)
                {
                        nodo=parameter1;
                        cerco nodo nel BST:
                        if(nodo non presente nel BST)
                        {
```

```
                    inserimento del nodo nel BST;
                    assegnamento numero_riga;
                }
                hops=parameter3;
                messaggio=parameter2;
                cercare numero_colonna nella matrice;
                if(hops<matrice(numero_riga,numero_colonna) )
                {
                    matrice(numero_riga,numero_colonna)=hops;
                }
                else
                {
                    if(matrice(numero_riga,numero_colonna)==-1)
                    {
                        matrice(numero_riga,numero_colonna)=hops;
                    }
                }

            }

        }
```

Paragrafo 2.4.3

La quarta ed ultima fase dell'algoritmo è quella di analisi delle informazioni contenute nella matrice create nelle precedenti tre fasi.

Il calcolo della copertura prevede che per ogni nodo calcoli la percentuale di messaggi ricevuta, quindi si faccia una media fra le medie di copertura di tutti i nodi.

Il calcolo del ritardo prevede che per ogni nodo si calcoli la media di hops che i messagi ricevuti impiegano per arrivare a tale nodo. Il ritardo totale è la media delle medie dei vari nodi.

In termini un po' più specifici , per il calcolo della singola copertura , si scorrerà la riga relativa ad un nodo ,contando il numero di messaggi che

sono stati ricevuti, si dividerà quindi tale numero per il totale di messaggi generati.
Per quanto riguarda invece le singole medie di ritardo, si sommeranno tutti i numeri diversi da -1 in una stessa riga, dividendo per il numero totale di messaggi generati.

Paragrafo 2.5
Avendo presente quanto detto nel paragrafo precedente, verrà presentato, in questo paragrafo, l'algoritmo in pseudocodice vero e proprio. Quello usato nel precedente paragrafo è un misto di linguaggio naturale e qualche parola chiave per rendere abbastanza chiaro il direzionamento del flusso.
Lo pseudocodice che verrà riportato costituisce un'informazione non ambigua, una prima versione di quello che sarà il codice vero e proprio. Si ricorda che lo pseudocodice non è un linguaggio di programmazione e che quindi l'algoritmo in pseudocodice non ne costituisce l'implementazione. Lo pseudocodice che segue è orientato alle funzioni. Significa che le istruzioni sono raggruppate in funzioni, ovvero insiemi di istruzioni che esistono al fine di, dato un input, attraverso una fase di elaborazione descritta all'interno, restituire un output.

```
fine_file_traccia=0;
numero_messaggi=0;
numero_nodi=0;
while(!fine_file_traccia)
{
        buffer=lettura_8_caratteri( trace_file);
        if(buffer== “STAT GEN”)
           numero_messaggi=numero_messaggi+1;
        else
           if(buffer== “STAT RCV”)
            {
               presente=0;
               value=leggi_parameter1(trace_file );
                 presente=trova_nodo_in_BST ( value);

                     if (!presente)
               {
                     inserisci_nodo_in_BST(value);
                     numero_nodi=numero_nodi+1;
               }
         }
           scorri_riga(trace_file);
           fine_file_traccia=controlla(trace_file);
}
```

lettura_8_caratteri(file)
{
descrizione input e output della funzione:
data la riga del trace_file che ha in input, ne legge i primi otto caratteri che vengono quindi restituiti in output.
}

leggi_parameter1(trace_file)
{
descrizione input e output della funzione:
data la riga del trace_file legge il primo parametro e lo restituisce.
}

trova_nodo_in_BST (value)
{
descrizione input e output della funzione:
dato un valore, quindi il nome di un nodo, analizza il BST in cerca di tale valore, restituisce “1” se il nome del nodo è presente altrimenti “0”.
}

```
inserisci_nodo_in_BST(value)
{
    descrizione input e output della funzione:
    inserisce all'interno del BST il valore ricevuto in
    input. In questo caso non restituisce nulla in
    output.
}
scorri_riga(trace_file)
{
    descrizione input e output della funzione:
    fa in modo che trace_file costituisca la riga
    successiva. trace_file dopo tale funzione
    rappresenta la riga successiva a quella
    precedentemente analizzata.
}
controlla(trace_file)
{
    descrizione input e output della funzione:
    verifica che trace_file non stia rappresentando
    nulla in seguito alla lettura di tutto il file.
    In altre parole se la funzione restituisce "1" il
    file di traccia è stato letto per intero, in caso
    contrario verrà restituito "0".
}
```

CAPITOLO 3

SCELTA DELLE STRUTTURE E CALCOLO DEI COSTI

Paragrafo 3.0

In questo paragrafo sono presenti alcune definizioni per agevolare la comprensione del resto del capitolo.
Tipi di dato: una variabile è un nome simbolico a cui viene assegnato un valore o un insieme di valori, tali valori non rimangono fissi per tutta la durata del programma ma possono essere modificati.[7] Ad una variabile viene di solito associato un tipo di dato che permette di stabilire quanta memoria sia necessaria per contenere l'informazione di quella variabile. Specificare un tipo di dato permette quindi al calcolatore di allocare la quantità di memoria che è caratteristica dello specifico tipo di dato.[8]
Struttura dati e struttura dati astratta: le strutture dati costituiscono uno dei principali mezzi per rappresentare ed organizzare le informazioni.
Null: il valore Null ha lo scopo di contrassegnare le variabili come vuote. Perciò se una variabile contiene Null significa che è vuota, se un puntatore, che è uno spazio di memoria contenente un indirizzo, *punta a* Null, significa che non punta a nulla.[9]
Costo di un algoritmo: il costo di un algoritmo si quantifica al fine di capire quanto l'algoritmo sia efficiente e data una specifica grandezza, che sia astratta o meno, quali siano i tempi necessari in relazione ad essa.
"In genere non serve conoscere esattamente il numero delle operazioni aritmetiche, ma basta quantificarne la grandezza in funzione di un parametro *d* legato alla dimensione del problema che si sta risolvendo". [10]

Paragrafo 3.1

In questo capitolo verranno descritte le strutture scelte per l'algoritmo con conseguente calcolo dei costi. Il calcolo dei costi dipende infatti anche dalle strutture dati che si intendono usare. L'organizzazione dei dati permette di effettuare ricerche più o meno veloci, inserimenti più o

meno veloci.
In parte la scelta delle strutture dati è già stata accennata per quanto riguarda le informazioni relative ai nodi del grafo. Come già detto tali informazioni verranno memorizzate in un “binary search tree” ovvero un albero binario di ricerca. Tale struttura dati è particolarmente indicata per la parte dell'algoritmo che prevede la memorizzazione degli identificativi dei nodi, nel caso in cui l'albero creato sia bilanciato.
Per quanto riguarda i messaggi del file di traccia, verranno memorizzati in un vettore disordinato.

Paragrafo 3.2
Al fine di comprendere il calcolo dei costi si analizza ora la differenza fra un vettore ordinato ed uno disordinato. Il calcolo dei costi verrà infatti diviso in due parti, una riguarda il calcolo dei costi con vettore dei messaggi ordinati, l'altro con vettore dei messaggi disordinato.
Partiamo dal considerare le operazioni che verranno effettuate su tale vettore. L'algoritmo prevede l'inserimento di tutti i nomi dei messaggi e la ricerca di uno specifico messaggio, al fine di inserire correttamente le informazioni relative al numero di hop. L'operazione di inserimento avviene quando l'algoritmo esamina una riga del trace file che inizia con “STAT GEN”. Siccome un messaggio viene generato una e una sola volta dopodichè viene propagato, si ha che l'inserimento in un vettore non comprende la ricerca del doppione. Il nome del messaggio dovrà essere necessariamente inserito. Fra i due tipi di vettori, nel caso dell'inserimento, ciò che varia sarà la necessità di posizionare il nuovo nome in modo corretto rispetto ad un ordine, caso del vettore ordinato, o semplicemente procedere all'immediato inserimento, nel caso in cui il vettore sia disordinato.
E' già chiaro che nel caso dell'inserimento la struttura dati che lo rende più veloce è il vettore disordinato, il quale non *chiede* che venga mantenuto un ordine e l'operazione risulta immediata.
L'altra operazione riguarda la ricerca. La ricerca in un vettore ordinato è semplificata, si potrà sfruttare l'esistenza dell'ordine. La ricerca

all'interno di un vettore disordinato prevede che vengano analizzati tutti i nomi ovvero che le informazioni vengano *scansionate* per intero. Quindi nel caso della ricerca è chiaramente più costosa la ricerca in un vettore disordinato rispetto a quello ordinato.

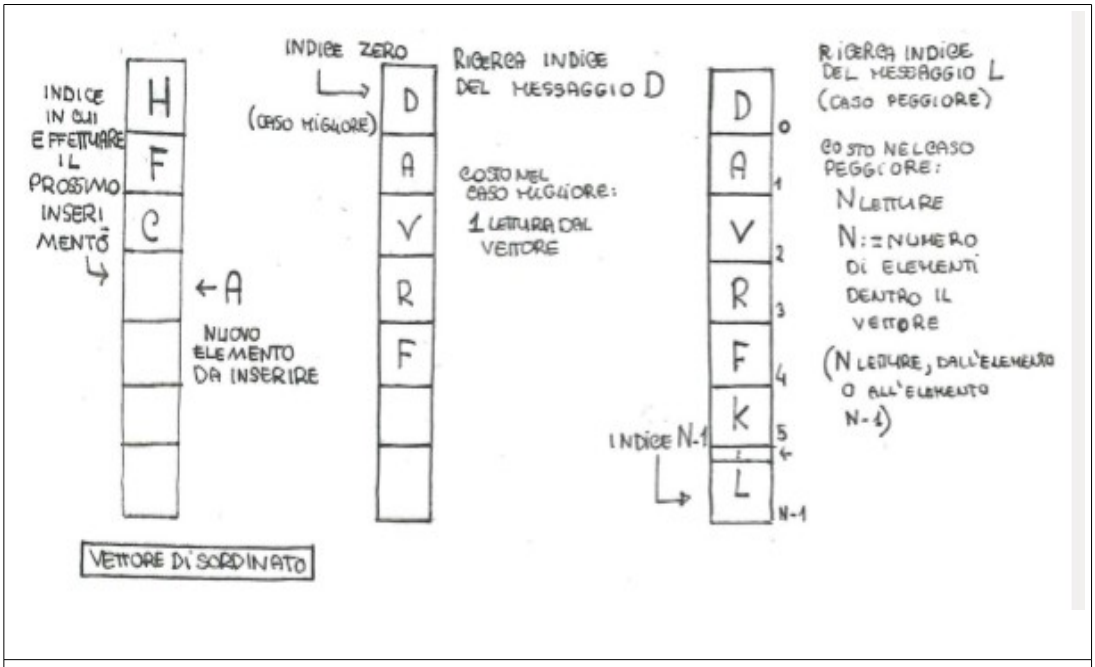


Illustrazione 3.0 : vettore disordinato

Come mostra anche l'illustrazione 3.0 il vettore disordinato effettua l'inserimento in modo immediato. Ciò significa che il costo di tale operazione non dipende dal numero totale degli elementi: l'inserimento ha un costo costante, costante rispetto al numero degli elementi del vettore. A sinistra dell'illustrazione viene mostrato proprio l'inserimento all'interno del vettore. Tutto ciò che l'algoritmo deve fare affinchè ogni inserimento sia consistente, è tenere in memoria il numero di elementi inseriti nel vettore ed aggiornarlo ogni volta che avviene un inserimento.
In questo modo si avrà subito l'indice di inserimento dell'elemento nel vettore.

Il disegno centrale mostra il caso migliore della ricerca in un vettore disordinato. La ricerca si riferisce alla ricerca della posizione in cui inserire un nuovo elemento. Si ricorda che i messaggi verranno generati una sola volta perciò non si tenterà mai di inserire doppioni.
La ricerca, nel caso di un vettore disordinato, avviene durante il riempimento della matrice. Di conseguenza si tratta solamente di trovare la coordinata y dell'informazione che si deve inserire. Si ricorda che l'informazione è il numero di hop che il messaggio y impiega per arrivare al nodo x. Tale informazione viene memorizzata nella cella (x,y). Il costo della ricerca è quindi nel caso migliore, pari ad una lettura. Sarebbe il caso in cui l'elemento che si cerca è memorizzato nella prima posizione dell'array. Il caso pessimo consiste nello scorrere tutti gli elementi inseriti, quindi il numero totale di messaggi generati.
L'illustrazione 3.1, relativa al vettore ordinato mostra cosa comporti l'inserimento in un vettore ordinato.
Al contrario di quello disordinato c'è un ordine da mantenere obbligatoriamente, quindi esiste una sola posizione corretta in cui inserire un determinato elemento. Ordinare un vettore ha un costo non inferiore a n*log2(n) se si utilizza un algoritmo di tipo “comparison -sort”, n è il numero di messaggi totali, ovvero il numero totale di elementi nel vettore.[11] Si suppone di usare un algoritmo di questo tipo.
Per quanto riguarda l'inserimento dei dati necessari all'interno della matrice, ciò che avviene è una ricerca della posizione corretta all'interno del vettore. Tale ricerca su un vettore ordinato ha costo logaritmico log2(n), dove n è il numero di messaggi totale. Tale ricerca è detta ricerca binaria.[12] E' è illustrata in figura 3.2.
In pratica l'algoritmo di ricerca binaria ad ogni passo scarta metà degli elementi del vettore. Si inizia partendo dall'elemento che si trova in una posizione centrale del vettore. L'elemento che si cerca, elementoR, viene comparato all'elemento al centro del vettore, elementoC. Se elementoR>elementoC, si focalizza l'attenzione sulla seconda metà del vettore, quindi non si considera più la metà al di sopra. Se elementoR<elementoC si esclude la metà sottostante. Iterando tale

procedimento si trova la posizione dell'elemento in circa log2(n), un'analisi dettagliata sarà svolta nel paragrafo successivo.

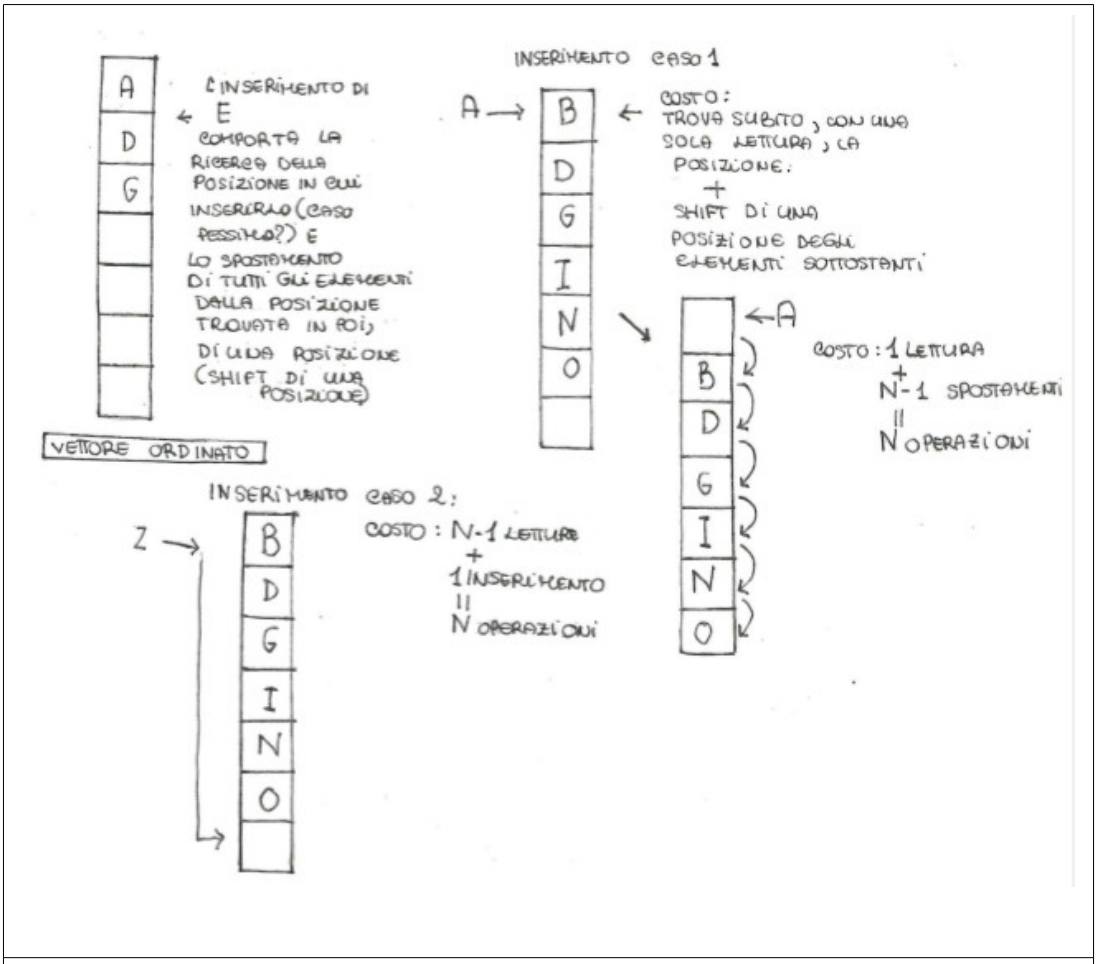


Illustrazione 3.1 :vettore ordinato

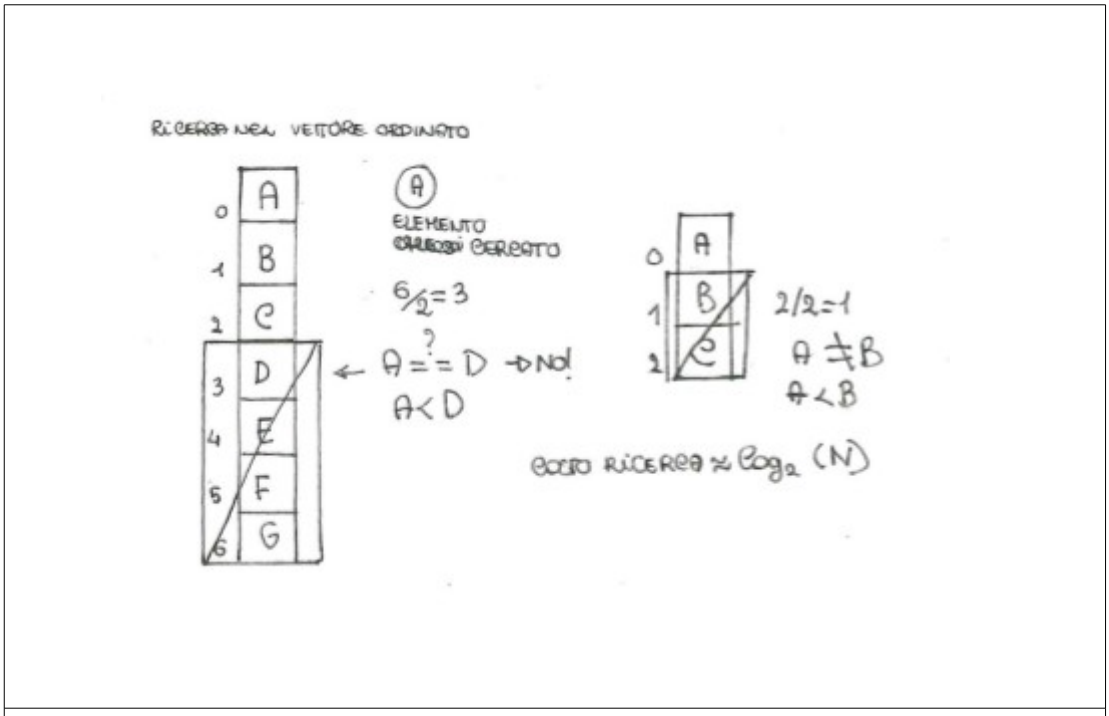
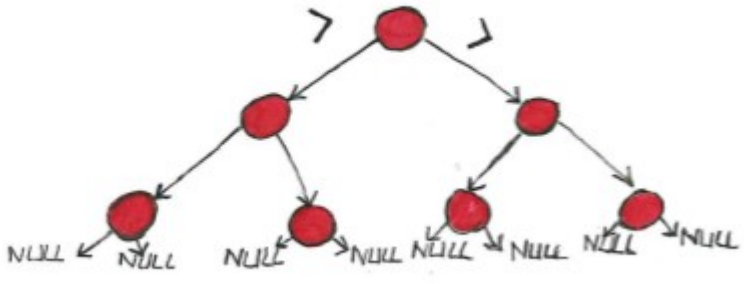


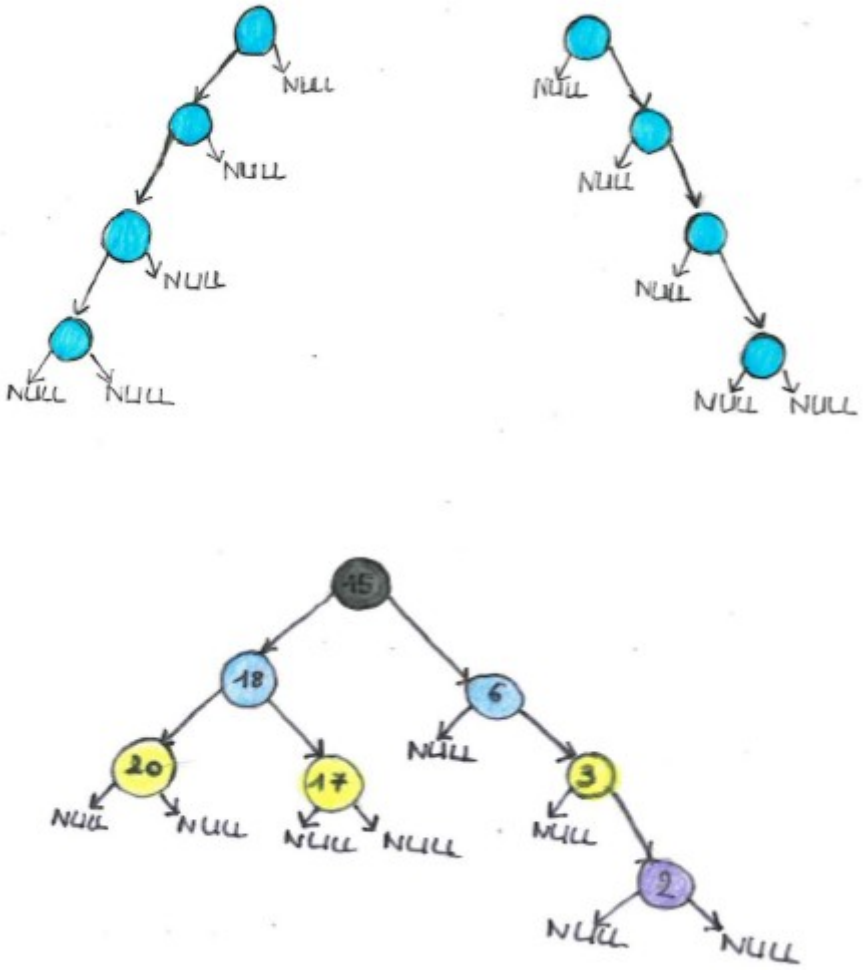
Illustrazione 3.2

Paragrafo 3.3

Sempre con lo scopo di facilitare la comprensione del calcolo dei costi presente nel paragrafo successivo, si analizza la struttura dati precedentemente nominata del “Binary Search Tree” abbreviato BST.



Un Binary Search Tree o BST, è una struttura dati astratta, ovvero non l'implementazione pratica di una struttura dati bensì un insieme di caratteristiche che definiscono un'idea astratta di un'organizzazione di dati. L'albero è composto da nodi. Ogni nodo ha al massimo due figli. Un nodo può avere un figlio sinistro e basta, un figlio destro e basta, entrambi o nessuno. Se un nodo non ha figli tale nodo è chiamato foglia. Nell' illustrazione i nodi foglia puntano a null, sia per quanto riguarda il puntatore al figlio sinistro che per quanto riguarda il puntatore al figlio destro. E' un modo di rappresentare le foglie.
Quando il BST è bilanciato l' altezza h è approssimativamente log2(n).
Quindi h ~log2(n) dove n è il numero di nodi presenti nel BST.
L'ordine presente nel BST è dato dalle seguenti regole[20]:
ogni nodo ha alla sua sinistra nodi con valori maggiori (o minori),
ogni nodo alla sua destra ha nodi con valori minori (o maggiori).
Il fatto che i nodi alla sinistra siano maggiori o viceversa non apporta modifiche alla struttura dati in termini di costi né di funzionalità. La scelta qualunque sia non cambia la sostanza della struttura dati. In questo progetto l'albero è ordinato con i figli a sinistra maggiori e quelli a destra minori, le figure rappresentano questo tipo di ordine.
L'immagine (albero BST bilanciato) è la rappresentazione di un albero binario di ricerca bilanciato. Un BST perfettamente bilanciato ha altezza pari a log2(n+1), dove n è il numero di nodi.[14] Sotto degli esempi di alberi non bilanciati e di alberi sbilanciati in modo estremo. Un albero sbilanciato *completamente* ha l'altezza che coincide con il numero di nodi, in pratica sono identici a liste linkate, liste di celle di memoria che puntano alla cella successiva mediante un puntatore, con gli elementi disposti secondo l'ordine indicato in precedenza.



L'immagine qui sopra è un altro esempio di albero binario. Tale immagine non corrisponde a nessuno dei due casi estremi delle immagini

precedenti. In questo caso si ha un albero abbastanza bilanciato ma non perfettamente bilanciato.
La ricerca in un albero BST è tanto meno costosa quanto più l'albero è bilanciato.
I costi di una ricerca all'interno di un BST sono infatti in relazione all'altezza dell'albero. Essa parte dal nodo radice e dato il numero contenuto in tale nodo si chiede se può fermarsi cioè se ha trovato l'identificativo che cercava o nel caso in cui l'identificativo non sia quello giusto, se debba andare nel sottoalbero a sinistra o in quello a destra. Quindi nel caso in cui il nome del messaggio radice sia uguale a quello cercato allora la ricerca si ferma. L'elemento è stato trovato. Se invece l'elemento cercato è maggiore di quello contenuto nella radice, si procede verso il sottoalbero di sinistra. Se l'elemento cercato è minore si procede verso il sottoalbero di destra. Per *procedere verso il sottoalbero* di sinistra o destra si intende in pratica leggere il figlio di sinistra o destra del nodo che si sta leggendo in quel momento. Il fatto che si stia su un particolare nodo, infatti, rende possibile la lettura del valore numerico di tale nodo ma anche la lettura dell'indirizzo del figlio di sinistra e di quello di destra.
Perciò ritornando al costo della ricerca, se tale è il procedimento con cui si cerca un nodo, si ha un massimo di letture pari all'altezza dell'albero. Si noti infatti come la lettura di un figlio implichi necessariamente la discesa di un livello lungo l'altezza. Ciò significa che dato uno stesso numero di nodi presenti in un albero binario di ricerca, tanto più è bilanciato, tanti meno nodi dovranno essere letti prima di arrivare alle foglie. Quindi supponendo il caso peggiore in nel quale si cerchi un elemento fino ad arrivare alle foglie (caso che si verifica se l'elemento è una foglia o non è presente nell'albero), dati gli n nodi, se l'albero è *perfettamente* bilanciato si effettueranno log2(n) letture se l'albero è *perfettamente* sbilanciato le operazioni saranno n.
Esempi numerici:
n=16
albero perfettamente bilanciato albero perfettamente sbilanciato

numero massimo di operazioni di ricerca: 4

numero massimo di operazioni di ricerca:16

Paragrafo 3.4

In questa tabella si trovano alcune delle espressioni e dei simboli utilizzati nel prossimo paragrafo per descrivere i costi.

Espressione	Significato
log2(k)	Logaritmo in base due di k
m	Numero di messaggi generati e numero di righe GEN nel file di traccia
R	Numero di righe totali del trace file
R-m	Numero di righe RCV nel file di traccia
n	Numero di nodi

Paragrafo 3.5

Costi in ordine temporale.
Nel caso di un albero binario di ricerca bilanciato:
(R-m)*log2(n)+n+m+mlog2(m)+(R-m)*(log2(n)+log2(m)).
Quindi escludendo le grandezze minori:
(R-m)*log2(n)+mlog2(m)+(R-m)*(log2(n))+(R-m)*(log2(m))
O((R-m)*(log2(n)+log2(m)))=O((R-m)*log2(n)+(R-m)*log2(m)).

(R-m) volte si effettua una ricerca all'interno del BST la ricerca nel BST nel caso bilanciato è log2(n).
Vi sono poi n inserimenti di elementi all'interno del BST.
Vi sono poi m inserimenti nel vettore dei messaggi. Si ordinano pagando un costo di mlog2(m).
Quindi per (R-m) volte si cerca un particolare elemento all'interno del

BST, come già detto tale ricerca costa log2(n). Oltre all'indice del nodo si cerca l'indice del messaggio. Quindi log2(m).

Paragrafo 3.6
Calcolo dei costi con vettore dei messaggi ordinato e albero non bilanciato:
(R-m)*n+n+m+mlog2(m)+(R-m)*(n+log2(m)).
Quindi:
O((R-m)*n+mlog2(m)+(R-m)n+ log2(m)(R-m))
O((R-m)(n+log2(m))).

I passaggi sono gli stessi del caso precedente con la differenza che la ricerca nel BST costa n e non log(n).

Paragrafo 3.7
Calcolo dei costi con vettore dei messaggi disordinato ed albero bilanciato:
(R-m)*log2(n)+n+m+(R-m)*(log2(n)+m).
Quindi:
O((R-m)*log2(n)+(R-m)*m)

Nel caso in cui non ci sia un ordine fra le varie colonne la ricerca costa zero poiché l'identificativo potrà essere inserito ovunque e visto che non si ha intenzione di mantenere un ordine il punto più conveniente è la posizione successiva all'ultimo identificativo aggiunto. Il costo, dati k identificativi già aggiunti, è il costo di inserimento O(1), una costante. Quando si parte ad esaminare il trace file si avrà quindi la riga di generazione(GEN) del messaggio che implicherà un inserimento. Quando si legge una riga di ricezione(RCV) di un messaggio allora si dovrà cercare il giusto valore, il caso peggiore sarà quello nel quale il messaggio si trova nell'ultima posizione. Perciò il costo peggiore che si può ottenere da questa operazione è leggere tutti gli m identificativi.

Paragrafo 3.8
Calcolo dei costi con vettore dei messaggi disordinato ed albero non bilanciato:
(R-m)*n+n+m+(R-m)*(n+m).
Quindi:
O((R-m)*n+(R-m)*m).

La spiegazione è analoga a quella sopra ma cambia il costo di ricerca nel BST.

Paragrafo 3.9
Confronto fra il costo con vettore ordinato e albero bilanciato ed il costo con vettore disordinato e albero bilanciato:
(R-m)*log2(n)+n+m+mlog2(m)+(R-m)*(log2(n)+log2(m)).
(R-m)*log2(n)+n+m+(R-m)*(log2(n)+m).

Si pone (R-m)=K.

K*log2(n)+n+m+m*log2(m)+K*log2(n)+K*log2(m).
La più grande di queste grandezze è probabilmente K*log2(m).

K*log2(n)+n+m+K*log2(n)+K*m.
La più grande di queste grandezze è probabilmente K*m.

Tali supposizioni sono vere solo nel momento in cui m è maggiore di n.
Ovvero il numero di messaggi che girano in rete sono maggiori del numero di nodi che la compongono.
Inoltre si suppone anche che K>>m. Ovvero che il numero di righe di tipo RCV siano presenti in numero maggiore alle righe di tipo GEN all'interno del trace file.

Paragrafo 3.10
Confronto fra il costo con vettore ordinato e albero sbilanciato ed il costo

con vettore disordinato e albero sbilanciato.

(R-m)*n+n+m+mlog2(m)+(R-m)*(n+log2(m)).

(R-m)*n+n+m+(R-m)*(n+m).

Si pone (R-m)=K.

K*n+n+m+m*log2(m)+K*n+K*log2(m).
La maggiore di tali grandezze è probabilmente K*n.
Supponendo che il numero di nodi sia maggiore di log2(m).
Se così non fosse allora si avrebbe K*log2(m).

K*n+n+m+K*n+K*m.
La maggiore è probabilmente K*m, supponendo che il numero di messaggi sia maggiore al numero di nodi.

Paragrafo 3.11
La matrice contiene n*m elementi. Quindi nel calcolo del ritardo e della copertura verranno letti n*m elementi.
O(n*m).
Per quanto visto sopra nel caso in cui l'albero sia sbilanciato, il caso peggiore, si hanno i segueni costi:
O(K*m) o O(K*n).
Supponendo che K=R-m>>n e K>>m si ha che la parte più costosa rimarrà quindi la parte che si impegna nella creazione della matrice.
La prima soluzione implementata prevede che il vettore sia disordinato.
Ovviamente è sempre possibile modificarla inserendo una parte relativa all'ordinamento di tale vettore.

CAPITOLO 4

CODICE PREESISTENTE

Paragrafo 4.0

In questo capitolo si faranno alcune osservazioni per comprendere parte del codice della versione precedente ed anche quella sviluppata per questa tesi e per spiegare brevemente l'interazione che avviene fra alcune parti del simulatore LUNES.
Tali considerazioni servono a dare un'idea di come i processi, istanze del codice creato, interagiscano fra loro.
Prima di iniziare con l'analisi del codice preesistente vengono introdotti alcuni elementi.
Il Commento:Si tratta di frasi inserite all'interno del codice.
Un commento non ha alcun valore per l'elaboratore ma lo ha solo per il programmatore che lo legge, il fine è quello di "spiegare" o "specificare"; in generale un commento può chiarire un passaggio all'interno del codice oppure il significato di una variabile, qualsiasi elemento che il programmatore ritenga di dover chiarire attraverso ulteriori spiegazioni, ulteriori rispetto al codice già scritto. Innanzitutto si noti che i file in linguaggio C si individuano tramite l'estensione ".c", mentre i file bash, si individuano tramite l'estensione ".sh"[17]. Questi sono i due tipi di file che si analizzeranno in questa tesi. I commenti si possono individuare, in un file C, o in un file in linguaggio scripting che verrà interpretato dalla shell [21], prestando attenzione alle seguenti modalità:

Modalità	Caratteristiche
//	Tutto ciò che si trova dopo questi due caratteri,in un file che contiene codice in linguaggio C, su una stessa riga, fa parte di un commento.

/* */	Sempre all'interno di un file in linguaggio C, tutto ciò che si trova all'interno di tali caratteri, ovvero dove c'è lo spazio bianco, fa parte di un commento.
#	In un file bash il commento è successivo a tale simbolo.

Argomenti a riga di comando:Un programma nel momento in cui viene lanciato, può ricevere degli argomenti in input, dei parametri contenenti informazioni utili allo svolgimento del programma [15].
Infine vi è la hash table , che è una struttura dati[16].

Paragrafo 4.1
In questo paragrafo si pone l'attenzione sul file scripts_configuration.sh e sulle variabili che permettono di comprendere il resto del capitolo.
L'illustrazione 5.0 mostra alcune variabili che poi verranno usate in altri file. Si noti che vi è il numero di CPU, ovvero le unità di elaborazione centrali [18] della macchina.
Nell' illustrazione 4.1 invece vengono elencati vari nomi di file, posti quindi dentro a delle variabili. In particolare si notino le variabili NODES e MSGDIS, come suggeriscono i nomi il primo file NODES conterrà gli identificativi dei nodi, il secondo MSGDIS gli identificativi dei messaggi.

Paragrafo 4.2
Si osservino i file con nome sim-metrics-*-corpus, illustrazione 4.2.
Tali file includono le variabili definite in scripts_configuration.sh tramite l'istruzione seguente :
source scripts_configuration.sh.
Si noti ora la parte di codice riportata dall'illustrazione 4.2, tale parte di codice è identica per tutti i file di nome sim-metrics-*-corpus, (l'asterisco

indica la presenza di una parola, la sola parola che differisce da file a file, all'interno del rispettivo nome).

```
#############################################################
#       EXECUTION VALUES
#############################################################
#
#       Number of CPUs (cores) in the system
CPUNUM=`cat /proc/cpuinfo  | grep processor | wc -l`
#
#       Number of LPs to be used:
#               1 = monolithic (sequential) simulation
#               > 1 parallel simulation
LPS=1
```

Illustrazione 4.0 : parte di codice presa dal file scripts_configuration.sh

```
#############################################################
# File names definition, used for statistics purposes
#############################################################
#
#       output files
#
DELAYS="STAT_msg_delays.txt"
MEAN="STAT_mean_delay.txt"
MSGIDS="STAT_msg_ids.txt"
NODES="STAT_nodes_ids.txt"
OUTPUT="STAT_coverage.txt"
DISTRIBUTION="STAT_missing_distribution.txt"
```

Illustrazione 4.1 : parte di codice presa dal file scripts_configuration.sh

Osservando il codice sempre in figura 4.2, si evince che verranno lanciate tante istanze di “get_coverage_script” quante sono le CPU.

```
while [ $RUN -le $NUMBERRUNS ]; do

RUNNING=`ps aux 2> /dev/null | grep "get_coverage_script" | wc -l`
echo $RUN / $RUNNING

  if [ $RUNNING -le $CPUNUM ]; then
   mkdir -p $RESULTS_DIRECTORY/$TESTNAME/$RUN
   ./get_coverage_script "$TRACE_DIRECTORY/$TESTNAME/$RUN"
   $RUN $TESTNAME $LPS &
   let RUN=RUN+1
  else
   sleep 3
  fi
done
```

Illustrazione 4.2 : parte di codice presa da sim-metrics-*-corpus

CPUNUM è infatti una variabile contenente tale grandezza per quanto visto nel paragrafo precedente.
Tali processi vengono lanciati in background[19], notare infatti il carattere & successivo ai parametri di input del file get_coverage_script.
Lanciare processi in background significa che tali processi verranno eseguiti parallelamente.
Notare infine che nel caso in cui ci siano già un numero di processi istanze di get_coverage_script pari a $CPUNUM, ovvero al contenuto della variabile CPUNUM, si ha che il flusso di esecuzione prosegue nell' *else,* quindi il processo istanza di sim-metrics-*-corpus "dorme" per tre secondi.

Paragrafo 4.3

In questo paragrafo si analizza brevemente lo script che lancia gli eseguibili dei file in linguaggio C. Tale script verrà modificato nella versione elaborata per questa tesi.

```
source scripts_configuration.sh

SIMTRACEDIR=$1
RUNS=$2
TESTNAME=$3
LPS=$4

CURDIR=$PWD
rm -fr $WORKING_DIRECTORY/$TESTNAME/$RUNS
mkdir -p $WORKING_DIRECTORY/$TESTNAME/$RUNS
cd $WORKING_DIRECTORY/$TESTNAME/$RUNS
```

Illustrazione 4.3: parte di codice del file get_coverage_script

```
# First of all I've to find what
#are the message IDs generated in the
# simulation run and the IDs of all nodes
$CURDIR/./get_ids_next $SIMTRACEDIR "STAT_msg_ids.txt"
                            "STAT_nodes_ids.txt" $LPS
```

Illustrazione 4.4: parte di codice del file get_coverage_script

Come si può osservare nell'illustrazione 4.3, è presente l'istruzione "source" che permette di utilizzare variabili definite in scripts_configuration.sh, per quanto visto nel paragrafo 4.1.
Subito sotto a tale istruzione vi è l'assegnazione dei parametri di input dello script a delle variabili. Nell'illustrazione viene mostrato come venga lanciata un'istanza del file get_ids_next.c con i relativi parametri di input. Tali parametri sono raffigurati nell'illustrazione 4.6 e 4.8, la prima illustrazione è un commento che specifica il significato dei quattro parametri di input, la 4.8 invece è la parte del codice nel quale si procede

all'inizializzazione di alcune variabili con i parametri di input.
Nell' illustrazione 4.5 viene riportata la parte dello script che lancia il secondo file in linguaggio C.
I file lanciati sono quindi due, il primo ad essere lanciato è l'eseguibile del file get_ids_next.c, illustrazione 4.4, il secondo ad essere lanciato è l'eseguibile del file get_coverage_next.c. Entrambi sono file scritti in linguaggio C, mentre get_coverage_script.sh è uno script. L'immagine 4.5 mostra poi tutti i parametri passati in input all'istanza del file get_coverage_next.c, compresi NUM_NODES, NUM_MSGIDS e MSGIDS, l'illustrazione 4.13 permette di interpretare questi e gli altri parametri utilizzati, anche se si noterà come in realtà nel commento ce ne siano alcuni non utilizzati.
Nel caso specifico le due variabili NUM_NODES e NUM_MSGDIS sono rispettivamente il numero di nodi contenuti nel file il cui nome è memorizzato in NODES e il numero di messaggi contenuti nel file il cui nome è memorizzato in MSGDIS.

Paragrafo 4.4
Come già scritto la parte di codice che è di interesse in questo capitolo è organizzata in due file in linguaggio C. Gli eseguibili di tali files vengono lanciati da uno script di nome “get_coverage_script”. I due files in C si chiamano “get_coverage_next.c” e “get_ids_next.c”.
Nel paragrafo 4.4 si analizza il funzionamento del codice get_ids_next.c, partendo dai parametri di input.
Per quanto scritto nel paragrafo 4.0 la porzione di codice della figura 4.6 è un commento. Un commento che specifica quali siano gli argomenti ricevuti dal file C nel momento in cui viene lanciato dallo script.
Nello specifico tale illustrazione riporta i parametri di input del file get_ids_next.c.
Si legge che il primo argomento è la directory in cui si trovano i trace file. Tale informazione ha come ultima stringa “(input)”. Al contrario gli argomenti due e tre presentano una stringa “(output)”.

```
NUM_NODES=`wc -l $NODES | cut -f 1 -d" "`
NUM_MSGIDS=`wc -l $MSGIDS | cut -f 1 -d" "`

echo "get_coverage - processing traces in directory: $SIMTRACEDIR"

$CURDIR/./get_coverage_next \
$NUM_NODES \
$NUM_MSGIDS \
$MSGIDS \
$SIMTRACEDIR \
$WORKING_DIRECTORY/$TESTNAME/$RUNS/$RUNSCOVERAGETMP \
$WORKING_DIRECTORY/$TESTNAME/$RUNS/$RUNSDELAYTMP \
$RESULTS_DIRECTORY/$TESTNAME/$RUNS/$DISTRIBUTION \
$LPS
```

Illustrazione 4.5: parte di codice del file get_coverage_script

La stringa “(input)” in questo contesto significa che tale informazione verrà usata attivamente durante l'elaborazione, il risultato di tale processo utilizzerà invece gli argomenti di output, quelli che presentano la stringa “(output)”. Nel caso specifico, gli argomenti di output sono i due files in cui verranno memorizzate le informazioni che sono il risultato dell'elaborazione.
Il primo ed il quarto argomento invece verranno utilizzati all'interno dell'elaborazione stessa.
Perciò unendo le informazioni date dalle illustrazioni 4.2 , 4.4 e 4.6 si ha che:
"$TRACE_DIRECTORY/$TESTNAME/$RUN" viene passato come parametro di input da sim-metrics-*-corpus, si noti poi come RUN sia una variabile controllata all'interno della condizione di guardia del ciclo e ad ogni lancio di un nuovo processo viene incrementata, vedere illustrazione 4.2.

```
/*
 Input arguments and their semantic:

 argv[1]    Directory where to find the trace files (input)
 argv[2]    Filename of the message IDs file (output)
 argv[3]    Filename of the senders file (output)
 argv[4]    Total number of LPs in each simulaton run (input)
*/
```

Illustrazione 4.6: parametri passati in input al file get_ids_next.c

Quindi ad ogni processo lanciato viene assegnato come parametro di input, nello specifico si tratta della cartella che contiene i file di traccia, una cartella differente, contenente file di traccia differenti. Quindi ci sono al massimo CPUNUM processi che vengono eseguiti parallelamente e ad ognuno viene data una cartella contenente dei file di traccia. Quindi ogni istanza di get_coverage_script.sh esaminerà una specifica cartella di trace file.
Vi è poi il parametro numero quattro. Tale parametro verrà memorizzato nella variabile LPS come si legge nell'illustrazione 4.3.
L'illustrazione 4.7 mostra come venga utilizzato il parametro LPS all'interno del file get_ids_next.c .
Guardando l'immagine 4.8 e 4.6 si osserva che il parametro numero quattro, passato al file get_ids_next.c viene memorizzato nella variabile LPS.
Dall' illustrazione 4.7 si può osservare come poi tale parametro in input venga utilizzato dall'algoritmo.
Vi è infatti un primo while, quello più esterno, che permette di scorrere più trace file all'interno della stessa directory di input. Il nome di tale trace file viene ricavato dal nome della directory e dal current_lp, illustrazione 4.7. Dopodichè si inizia a leggere il file di traccia e ad elaborarlo, riga per riga. Le immagini 4.9 e 4.10 descrivono ciò che

avviene all'interno del ciclo.

```
while(current_lp<LPs)
{
  sprintf(messages_file,"%sSIM_TRACE_%03D.log",
                        messages_dir,current_lp);
  printf("get_ids_next:processing...%s
                     \n",messages_file);
  f_messages_file=fopen64(messages_file,"r");
  finished=0;
  while(!finished)
  {
    .
    .
    g_hash_table_insert(hash_table_ids,key,value);
    fprintf(f_ids_file,"%s\n",key);
    .
    .
}
```

Illustrazione 4.7 : parte del codice del file get_ids_next.c

Si analizza più approfonditamente il contenuto delle figure 4.9, 4.10, 4.7. Ciò che è riportato nelle illustrazioni 4.9, 4.10 è incluso all'interno del ciclo della figura 4.7.
Da tali porzioni di codice quì appaiono evidenti i seguenti due aspetti: l'utilizzo della directory, come già scritto all'interno di tale directory ci sono un numero pari a LPS file di traccia, il secondo aspetto che si nota è che per ogni variazione di current_lp si ha quindi la lettura di un file diverso ma la manipolazione delle stesse hash table, ovvero delle stesse strutture dati. Quindi i dati dei file contenuti nella directory in input, tali che abbiano nel nome un indice minore del numero di LPs passato in input, vengono memorizzati nelle stesse hash table.
Più nel dettaglio il file viene letto riga per riga, se si tratta di una riga che

contiene la stringa GEN, allora si effettua una ricerca dell'identificativo sulla hash table di nome hash_table_ids.

```
/*      Command-line parameters */
messages_dir = argv[1];
ids_file = argv[2];
senders_file = argv[3];
LPs = atoi(argv[4]);
```

Illustrazione 4.8: parte del codice del file get_ids_next.c

Nel caso in cui l'identificativo non sia presente verrà allora inserito e stampato sul file ids_file.
Nel caso in cui invece si tratti di una riga di tipo RCV si ha la ricerca all'interno della hash table hash_table_senders e l'eventuale inserimento in mancanza dell'identificativo.
Quindi nel caso manchi l'identificativo verrà pure stampato sul file senders_file.

Paragrafo 4.5

In questo paragrafo si analizzerà il file get_coverage_next.c. Il secondo file che viene lanciato dallo script get_coverage_script. Prima di iniziare si noti che il precedente file, get_ids_next.c ha permesso di ricavare dai file di traccia, gli identificativi di nodi e di messaggi. Tali identificativi sono stati memorizzati nei due file menzionati nel paragrafo precedente.
Tali file servono (illustrazione 4.5) a contare il numero di messaggi e di nodi, tali numeri verranno quindi passati come argomenti al processo get_coverage_next.c .
Come si osserva nell'illustrazione 4.11 anche get_coverage_next.c ha un ciclo del tutto simile a quello presente nell'altro file, get_ids_next.c.
Anch'esso riceve in input la stessa cartella contenente i file di traccia ed il

numero di LP che l'algoritmo scorre per comporre i nomi dei file contenenti i dati.

```
while (!finished)
{
  if (fgets(buffer, 1024, f_messages_file) != NULL )
 {
   memcpy(command, buffer, 9);
   command[9]='\0';
   if (strncmp(command, "STAT GEN", 8) == 0)
  {
    memcpy(messageid, buffer+9, 10);
    messageid[10]='\0';
    key = calloc(10, sizeof(char));
    memcpy(key, messageid, 10);
    return_value = g_hash_table_lookup (hash_table_ids, key);
    if (return_value == NULL)
    {
     value = malloc(sizeof(char));
     *value = 0;
     g_hash_table_insert (hash_table_ids, key, value);
     fprintf(f_ids_file, "%s\n", key);
     }
  }
```

Illustrazione 4.9 : parte del codice del file get_ids_next

```
if (strncmp(command, "STAT RCV", 8) == 0)
{
  memcpy(sender, buffer+9, 10);
  sender[10]='\0';
  key = calloc(10, sizeof(char));
  memcpy(key, sender, 10);
  return_value = g_hash_table_lookup (hash_table_senders, key);
  if (return_value == NULL)
  {
   value = malloc(sizeof(char));
   *value = 0;
   g_hash_table_insert (hash_table_senders, key, value);
   fprintf(f_senders_file, "%s\n", key);
  }
 }
}
else
finished = 1;
```

Illustrazione 4.10: parte del codice di get_ids_next.c, sempre interno al while presente in 4.9

Analizzando il file get_coverage_next.c si osserva che la parte di codice nell'illustrazione 4.12 mostra come vengano spostati gli identificativi contenuti nel file messages_file, che è stato passato in input al processo, all'interno di una hash table. Perciò la prima cosa che viene fatta è riempire una tabella hash con i valori all'interno del file presente fra i parametri di input, creato grazie al processo precedente. Nel codice vi è poi la scansione dei trace file. Riga per riga, vengono esaminate quelle che sono le righe di ricezione, ovvero dove compare la stringa “RCV”.
Vengono quindi letti gli identificativi contenuti in tale riga ovvero l'identificativo (o id) del messaggio ricevuto e l'identificativo del nodo ricevente. Si cerca quindi l'id del messaggio all'interno della hash table creata in precedenza.

```
while(current_lp<LPs)
{
  sprintf(log_file,"%sSIM_TRACE_%03D.log",
                        messages_dir,current_lp);
  printf("get_coverage_next:processing...%s
                    \n",log_file);

    .
    .
}
```

Illustrazione 4.11 : parte di codice in get_coverage_next.c

Nel caso in cui il messaggio sia presente all'interno di tale struttura si memorizza il terzo parametro presente nella riga letta dal trace file, ovvero il numero di hop impiegati dal messaggio per raggiungere il nodo. Tale parametro viene inserito nella hash table di nome bigtable ed il suo indirizzo viene scelto in base all'identificativo del nodo, immagine 4.15. Da tutto ciò si deduce che i dati di tutti i file contenuti nella stessa cartella vengono, come anche nel file get_ids_next, accorpati nelle stesse strutture dati.

```
while (!finished) {

        if (fgets(buffer, 1024, f_messages_file) != NULL ) {

                key = calloc(10, sizeof(char));
                memcpy(key, buffer, 10);

                value = malloc(sizeof(int));
                *value = counter;

                g_hash_table_insert (hash_table, key, value);

                counter++;
        }
        else
                finished = 1;
}
```

Illustrazione 4.12 : parte del codice di get_coverage_next.c

Tale funzione calcola le grandezze ritardo e copertura sui dati elaborati. Dopodichè la funzione restituisce il comando al main e vengono scritte sui file di output le due grandezze, ovvero ritardo e copertura.
La differenza fondamentale fra questa prima versione e la versione scritta nella tesi è che la prima è divisa in due file, ovvero l'elaborazione che permette di arrivare al calcolo delle due grandezze è “distribuita” in due processi lanciati dallo script get_coverage_script. Nella seconda versione invece si tratta di un unico file. Fra le varie conseguenze vi è la mancanza di necessità relativamente allo scrivere gli identificativi di nodi e messaggi su dei file da passare poi in input al secondo processo, istanza del file C get_coverage_next.c. Infatti le strutture dati presenti all'interno dell'algoritmo, di quest'unico file, manterranno in memoria, tutte le informazioni.

```
/*
 Input arguments and their semantic:

  argv[1]    Total number of nodes
  argv[2]    Total number of messages
  argv[3]    Filename of the messages file (input)
  argv[4]    Directory of the log (trace) files (input)
  argv[5]    File name of the coverage file (output)
  argv[6]    File name of the delay file (output)
  argv[7]    File name of the nodes and degree file (input)
  argv[8]    File name of the degree distribution file (input)
  argv[9]    File name of the average missing messages
             per degree file (output)
  argv[10]   Total number of LPs in this run (input)
*/
```

Illustrazione 4. 13: parametri di input del fiel get_coverage_next.c

Paragrafo 4.6

In questo paragrafo vengono elencati i valori di input del nuovo file get_ids_coverage_next.c.
Quello che è indispensabile al nuovo programma sono gli elementi descritti nell'illustrazione 4.17.
Quindi in sostanza la directory nella quale risiedono i file di traccia che devono essere analizzati dal processo, il numero di tali file che è rappresentato dal secondo argomento. Infine il terzo ed il quarto argomento sono i file dove verrano scritte le grandezze calcolate.

Paragrafo 4.7

Ricapitolando ciò che è stato scritto in questo capitolo, si ha una visione di ciò che fanno alcuni dei file del simulatore. Si conoscono il numero di istanze dello script lanciate in parallelo, si conoscono i parametri passati dallo script alle due istanze dei programmi in C, ed il loro significato quindi la funzione che svolgono all'interno del codice.

```
while (!finished)
{
 if (fgets(buffer, 1024, f_log_file) != NULL )
 {
   memcpy(command, buffer, 9);
   command[9]='\0';
   if (strncmp(command, "STAT RCV", 8) == 0)
   {
     memcpy(c_node, &(buffer[9]), 10);
     c_node[10]='\0';
     memcpy(c_message, &(buffer[20]), 10);
     c_message[10]='\0';
     value = g_hash_table_lookup (hash_table, c_message);
     if (value == NULL)
     {
       printf("GET_COVERAGE_NEXT: message identifier:
       %s NOT FOUND in the hash table\n", c_message);
       exit(0);
     }
```

Illustrazione 4.14: parte del codice del file get_coverage_next.c

```
        memcpy(c_delay, &(buffer[31]), 10);
        c_delay[10]='\0';
        i_delay = atoi(c_delay);

        index = (*value) * nodes + atoi(c_node);
        bigtable_rep[index]++;
```

Illustrazione 4.15: parte del codice del file get_coverage_next.c

```
        compute_coverage_and_delay();

        f_coverage_file = fopen(coverage_file, "a");
        fprintf(f_coverage_file, "%.2f\n", coverage);
        fclose(f_coverage_file);

        f_delay_file = fopen(delay_file, "a");
        fprintf(f_delay_file, "%.2f\n", delay);
        fclose(f_delay_file);
```

Illustrazione 4.16: parte di codice del file get_coverage_next.c

```
/*
    Input arguments and their semantic:
    argv[1]     Directory where to find the trace files (input)
    argv[2]     Total number of LPs in each simulaton run (input)
    argv[3]     File name of the coverage file (output)
    argv[4]     File name of the delay file (output)
*/
```

Illustrazione 4.17: parametri di input di get_ids_coverage_next.c

E' stata descritta la lettura dei file di traccia e la distribuzione di tale analisi fra i vari processi. Infine è ora chiaro su quali dei dati vengano calcolate le due grandezze, ritardo e copertura in modo da poter riproporre la stima relativa agli stessi dati anche nella nuova versione.
Si ripete infine che get_coverage_next.c, get_ids_next.c e get_coverage_script.sh sono i file che contengono le funzionalità sviluppate da questa tesi. Nel caso di questo progetto i due file in C si "fondono " in un unico file di nome get_ids_coverage_next.c. Lo script rimane uno ma cambia struttura in modo da avere un numero fisso di processi eseguiti in parallelo. La parte sull'esecuzione in parallelo si

trova nel capitolo sette.

CAPITOLO 5

ANALISI DEL CODICE

Paragrafo 5.0

Tale capitolo raccoglie l'analisi del codice scritto in C. Tale codice si trova all'interno del file get_coverage_and_delay.c.
Il paragrafo 5.0 raccoglie la definizione di tutti i termini che verranno usati in seguito.
La funzione main [22]: è la funzione che viene eseguita per prima all'avvio quando il processo, istanza del file in C, viene lanciato.
Vettore di stringhe in C: data la definizione di stringa[28] “In generale i testi sono chiamati stringhe e sono una sequenza di caratteri. Nel linguaggio C, le stringhe sono implementate come vettori di char che terminano con il carattere \0” e la definizione di array o vettore in C[29] “Un vettore è un gruppo di posizioni di memoria correlate dal fatto che tutte hanno lo stesso nome e tipo di dato” ; si ha che un vettore di stringhe può essere descritto in maniera astratta come una sequenza di stringhe, che a loro volta sono sequenze di caratteri. In seguito viene mostrato in pratica cosa siano e come vengano create usando il C, illustrazione 5.5, array_messages_names è un array di stringhe.
Simbolo “^”: Questo simbolo “^” rappresenta l'elevamento a potenza.
Variabile puntatore: è una variabile sufficientemente grande per contenere un indirizzo di memoria.[24]

Paragrafo 5.0.0

Questo paragrafo illustra la struttura del nodo di un BST, ovvero i tipi di dato che la compongono e la loro funzione. Tale nozione servirà a capire le successive funzioni riguardanti la “manutenzione” del BST.
L'illustrazione 5.0 mostra il nodo di un albero BST come composto da una stringa di undici caratteri, che sarebbe l'identificativo del nodo, tre puntatori ad un altro nodo, un carattere che ne rappresenta il colore ed infine l'indice di riga di tale nodo all'interno della matrice.

Il colore serve alla funzione che dovrà contare il numero di nodi presenti nel BST. Una volta contato il nodo passerà dal colore blue al rosso, in questo modo non vi è rischio di contare più volte lo stesso nodo. I tre puntatori sono, in ordine, il puntatore alla madre, quello alla figlia destra e quello alla figlia sinistra.

```
struct node{
char  name_of_the_node[11];
struct node * mother;
struct node * dx_daughter;
struct node * sx_daughter;
char color;
long number;
};
```

Illustrazione 5.0

Paragrafo 5.1

Come visto nel capitolo cinque relativamente al codice preesistente si ha che il processo, data una input directory, scorre tutti i trace file contenuti in tale directory. Il numero di file di traccia è dato da un ulteriore argomento passato in input al processo.

Quindi nella nuova versione del codice tale logica deve essere mantenuta, sono stati quindi aggiunti due cicli di tale "tipo", ovvero identici a quello che si trovano all'interno del codice precedente. Si tratta di due cicli in quanto si ricorda che nell'algoritmo presentato in precedenza, si hanno due letture dei trace file. In realtà unendo i due sorgenti precedenti, dove per sorgenti si intende file contenenti codice, si hanno comunque due cicli. Si veda il capitolo precedente a questo per ulteriori chiarimenti.

Si ricorda che il ciclo che si tratta, è identico a quello presentato nel capitolo precedente, dove in sostanza si scorrono tutti i trace file contenuti nella directory passata. Il nome di ogni file di traccia viene

composto grazie al nome della directory e al numero di current_lp, illustrazione 5.1.

```
int main(int argc, char * argv[])
{
  while(current_lp<LPs)
  {
    .
    .
  }
  while(current_lp<LPs)
  {
    .
    .
  }
}
```

Illustrazione 5.1: cicli all'interno dell'algoritmo in get_coverage_and_delay.c

Paragrafo 5.2

Questo paragrafo riguarda l'analisi del codice prensente nel main. I paragrafi successivi invece affronteranno l'analisi delle funzioni chiamate all'interno del main.

L'illustrazione 5.2 mostra la porzione del main in cui è stata inserita la fase di “prima lettura” dei file di traccia. Tale fase si trova all'interno del primo ciclo, illustrazione 5.1.

Quello che accade è che ogni volta che nel buffer, ovvero la stringa in cui è stata copiata una riga del file, è stata caricata una riga si controlla il “tipo ” di riga, ovvero se contiene GEN o RCV, nel caso in cui contenga RCV verrà letta anche la parte relativa al numero di hop, nonostante non sia necessaria al fine di creare il BST con gli identificativi. Il numero di hop serve al fine di creare una matrice contenente un certo tipo di dato, il minimo in grado di contenere le informazioni necessarie. In altre parole

in questa fase del codice si stima il valore massimo che dovrà essere contenuto all'interno della matrice, ma questo verrà meglio descritto più avanti in questo capitolo.
Le illustrazioni 5.3 e 5.4 mostrano nel dettaglio cosa accade ogni volta che si analizza la riga di ricezione, o in altri termini la riga che contiene la stringa RCV, relativamente alla gestione del binary search tree e alla ricerca del massimo numero di hop. Si osservi quindi la figura 5.4, la codizione dell'if chiede se il binary search tree contiene almeno un nodo. Se contiene almeno un nodo si cerca il nome dell'identificativo all'interno del BST. Ciò che viene restituito è il puntatore al nodo madre (o nodo padre). I casi sono due. L'identificativo è presente all'interno dell'albero, in quel caso verrà restituito NULL. Nel caso in cui l'identificativo non sia presente nell'albero si dovrà procedere all'inserimento. L'inserimento avviene facendo puntare quello che dovrà essere il nodo madre al nuovo nodo. Per questa ragione viene restituito, dalla funzione preposta a svolgere tale compito, che verrà affrontata in seguito, il puntatore al nodo che dovrà essere la madre di quello da inserire.
Si noti che tale parte di codice prosegue verificando appunto se il valore restituito dalla funzione di ricerca nel BST sia pari a NULL, e nel caso non lo sia si provvederà creare un nuovo nodo e ad inserirlo nell'albero.
La funzione che se ne occupa, come si legge dal sorgente, è Find_value_in_BST(). La creazione di un nuovo nodo avverrà invece tramite la funzione To_Set_Up_a_node().
Quindi si prosegue con l'inserimento vero e proprio, tramite la funzione Insert_node_in_BST().

```
while (!finished)
{
  if (fgets(buffer, 1024, f_messages_file) != NULL )
  {
    memcpy(command, buffer, 9);
    command[9]='\0';
    if (strncmp(command, "STAT GEN", 8) == 0)
    {

    }
  }
  if (strncmp(command, "STAT RCV", 8) == 0)
  {

   //operazioni relative alla gestione del BST

  }
 }
 else
finished = 1;
}
```

Illustrazione 5.2

Nel caso in cui il BST non contenga ancora alcun nodo, si tratta cioè di un abero vuoto la cui radice punta a NULL, si provvederà semplicemente a creare un nodo con l'identificativo. Tale nodo verrà puntato dal BST, quindi BST conterrà l'output della funzione che permette di creare un nodo, ovvero To_Set_Up_a_node().
Si prosegue azzerando il contatore degli lp (current_lp), illustrazione 5.5.

```
if(strncmp(command,"STAT RCV",8)==0)
{
    memcpy(sender,buffer+9,10);
    sender[10]='\0';
    memcpy(number_of_hops_string,buffer+31,10);
    number_of_hops_string[10]='\0';
    number_of_hops=Conversion_String_to_number(number_of_hops_string);
    if(number_of_hops>max_number_of_hops)
    {
      max_number_of_hops=number_of_hops;
    }
          .
          .
          .
```

Illustrazione 5.3: parte del codice del file get_coverage_and_delay.c.

```
     if(BST!=NULL)
     {
        pointer_to_node_mother=Find_value_in_BST(sender,
                                                 BST);
        if(pointer_to_node_mother!=NULL)
        {
          pointer_to_new_node=To_Set_Up_a_node(sender,
                                          inserted_nodes);
          Insert_node_in_BST(pointer_to_new_node,
                             pointer_to_node_mother);
        }
     }
     else
     {
        BST=To_Set_Up_a_node(sender,inserted_nodes);
     }
```

Illustrazione 5.4: parte del codice del file get_coverage_and_delay.c.

A questo punto viene creato l'array dei messaggi, si tratta di un array di stringhe. Si crea utilizzando il numero ora noto, dopo la prima scansione dei trace file, dei messaggi presenti nella simulazione.
Quindi si calcola il numero di nodi presenti nella simulazione tramite l'uso della funzione Discover_number_of_nodes_in_the_BST(), la quale esplora l'albero contandone gli elementi.

```
current_lp=0;
char * array_messages_names[number_messages];
number_nodes=Discover_number_of_nodes_in_the_BST(BST);
max_short_int=(long)(pow(2,(long)sizeof(short int)*8-1)-1);
max_int=(long)(pow(2,(long)sizeof(int)*8-1)-1);
```

Illustrazione 5.5: parte del codice del file get_coverage_and_delay.c.

Viene quindi calcolato il massimo numero rappresentabile[27] dalla macchina a disposizione nel caso in cui si utilizzi uno short int e un int, tali valori verranno posti rispettivamente nelle variabili max_short_int e max_int.
Per farlo si utilizza la funzione “sizeof” [26] del linguaggio C che restituisce il numero di byte che vengono allocati per il tipo di dato passato in input.
La formula che permette di comprendere quale sia il numero massimo istanziabile può essere sinteticamente spiegata così: dato un numero di bit a disposizione il numero che può essere rappresentato in tale spazio di memoria equivale a riempire ogni bit col valore 1, infatti in binario le cifre di rappresentazione sono due, 0 e 1; tale cifra è data dall'espressione scritta sopra ovvero da 2^(numero di bit) -1, a parole sarebbe due elevato al numero di bit sottraendo 1 al risultato.

Come esempio prendiamo un vettore di 6 bit:

0	1	0	0	0	1

In binario tale vettore corrisponde al numero 1+2^4=1+16=17.
Il motivo è che a partire da destra verso sinistra ogni cella rappresenta un fattore che verrà moltiplicato per due elevato alla potenza data dalla sua posizione.
Di conseguenza per sapere quale sia il numero massimo rappresentabile con sei bit basterà riempire il vettore di uno, come segue:

1	1	1	1	1	1

In questo caso il numero rappresentato è, per esteso, (2^0)*1+(2^1)*1+(2^2)*1+(2^3)*1+(2^4)*1+(2^5)*1=1+2+4+8+16+32 =63.
Si noti infine che 63=64-1=2^6-1.
Ritornando al codice dell'illustrazione 5.4 l'operazione aritmetica che viene fatta consiste proprio nel calcolo illustrato qui sopra, dove il numero di bit si ottiene tramite la funzione sizeof moltiplicata al numero otto, in quanto ogni byte è composto da otto bit.
Quindi una volta calcolato il massimo valore che si può rappresentare si effettuano dei confronti, l'immagine 5.6 ne riporta uno. Se il numero di hop più grande è minore del massimo numero rappresentabile con il tipo short int, quello che occupa fra i tipi di interi meno spazio, allora verrà allocata una matrice di short int. Nel caso in cui non fosse sufficiente per rappresentare tutti i valori si utilizzerà una matrice di interi int oppure una di long seguendo il ragionamento analogo a quello effettuato nell'illustrazione 5.6. Infine viene memorizzato il tipo di dato scelto nella variabile type perchè tale conoscenza è necessaria per le funzioni successive.
In seguito all'allocazione che avviene nella parte di codice della figura

5.6 si ha l'inizializzazione della matrice, l'illustrazione 5.7. La condizione dell'if ha il compito di verificare quale matrice sia stata allocata, quella degli short, di int o di long.
L'accesso alla matrice avviene tramite il calcolo dell'indice giusto noti il numero di colonna e quello di righe.

```
    if(max_number_of_hops<=max_short_int)
        {
           Short_Matrix=(short *)malloc(sizeof(short)
           *number_nodes*number_messages);
           if(Short_Matrix==NULL)
           {
              printf("Problema nella creazione
                     della matrice!\n");
              return 1;
           }
           printf("Ho creato la matrice di short\n");
           strcpy(type,"short");
        }
```

Illustrazione 5.6: parte di codice di get_coverage_and_delay.c

```
if(Short_Matrix!=NULL)
        {
            for(i=0;i<=number_nodes-1;i++)
            {
                for(j=0;j<=number_messages-1;j++)
                {
                  Short_Matrix[i*number_messages+j]=-1;
                }
            }
        }
```

Illustrazione 5.7: parte di codice di get_coverage_and_delay.c

```
if(Short_Matrix!=NULL)
       {
        Re_reading_of_trace_file( pointer_to_trace_file,
                                  Short_Matrix,
                                  BST,
                                  number_messages,
                                  type,
                                  array_messages_names,
                                  messages_dir,
                                  LPs);
         remove(filename);
         Print_the_Matrix(Short_Matrix,number_nodes,number_messages,type);
         Compute_coverage_and_delay(Short_Matrix,
                                    number_messages,
                                    number_nodes,
                                    &coverage,
                                    &delay,
                                    type);
      }
```

Illustrazione 5.8

Viene quindi chiamata la funzione Re_reading_of_trace_file() che contiene il secondo ciclo che scorre tutti i file. Come si vedrà in seguito in questo capitolo tale funzione serve a riempire la matrice delle informazioni necessarie al calcolo delle due grandezze, per farlo occorre visionare nuovamente i trace file.
La funzione (Illustrazione 5.8) di nome Print_the_Matrix riguardano la stampa dei valori calcolati all'interno della matrice, è stata utilizzata per fare dei test di controllo. Infine attraverso la funzione Compute_coverage_and_delay() verranno calcolati i due valori, copertura e ritardo.

```
        f_coverage_file = fopen(coverage_file, "a");
        fprintf(f_coverage_file, "%.2f\n", coverage);
        fclose(f_coverage_file);
        f_delay_file = fopen(delay_file, "a");
        fprintf(f_delay_file, "%.2f\n", delay);
        fclose(f_delay_file);
```

Illustrazione 5.9: parte di codice del file get_coverage_and_delay.c

Nell'illustrazione 5.9 viene mostrata l'ultima parte del main. Usciti dal secondo ciclo di esplorazione della cartella dei trace file, dove tale esplorazione ha permesso di calcolare il ritardo e la copertura medi, vengono scritte le due grandezze calcolate e memorizzate nelle variabili "coverage" e "delay", nei rispettivi file, "f_coverage_file" e "f_delay_file".

Paragrafo 5.3

In questo paragrafo si analizza la funzione Re_reading_of_trace_file().
Essendo una funzione piuttosto estesa si propongo immagini parziali di codice disposte nell'ordine corretto.
Nell'immagine 5.10 vi sono alcune variabili della funzione, i parametri di input ed il tipo dell'output della funzione. I parametri di input sono il puntatore al file, il riferimento alla matrice (puntatore e riferimento sono la stessa cosa, un riferimento permette alla funzione che lo conosca di accedere alla risorsa "puntata"), l'albero dei nodi, il numero di messaggi che come già detto permette di accedere in modo corretto alla matrice, il tipo della matrice ed infine il vettore contenente gli identificativi dei messaggi.

```
void Re_reading_of_trace_file(FILE * pointer_to_trace_file,
                                                 void * Matrix,
                                             struct node * BST,
                                             long tot_messages,
                                                   char * type,
                                    char ** array_messages_names,
                                              char * messages_dir,
                                                        int LPs )
{
    char buffer[1024];
    char command[10],sender[11],messageid[11],
                      number_of_hops_string[11],
                            messages_file[1024];
    long number_of_hops, inserted_messages=0,
                                       index_row,
                                  index_column,
                                             i;
    struct node * pointer_to_BST_node;
    int finished=0,current_lp=0;


    short var;
```

Illustrazione 5.10

```
while (current_lp < LPs)
{
   sprintf(messages_file, "%s/SIM_TRACE_%03d.log",
                                      messages_dir,
                                       current_lp);
   pointer_to_trace_file = fopen64(messages_file,
                                            "r");
   finished=0;
   while(!finished)
        {
           if(fgets(buffer,1024,pointer_to_trace_file)!=NULL)
            {
                memcpy(command,buffer,9);
                command[9]='\0';
                if(strncmp(command,"STAT GEN",8)==0)
                {
                    memcpy(messageid,buffer+9,10);
                    messageid[10]='\0';
                    Inserting_message(inserted_messages,
                                  array_messages_names,
                                        messageid);
                    inserted_messages++;
                }
```

Illustrazione 5.11

Quindi vi è la seconda scansione di tutti i trace file contenuti nella cartella da analizzare, illustrazione 5.11.
Vi è poi un ciclo che riesamina nuovamente il file riga per riga.
Nel caso in cui la riga sia di tipo GEN allora si inserisce l'identificativo del messaggio all'interno del vettore dei messaggi, che si ricorda esser stato allocato ma non riempito.
Si effettua l'inserimento attraverso la funzione Inserting_message(), dopo aver effettuato l'inserimento si aumenta il numero di messaggi inseriti, necessario se si vuole riempire velocemente il vettore.

La funzione Inserting_message() viene mostrata dalla figura 5.12.

```
void Inserting_message(long inserted_messages,
                  char ** array_messages_names,
                              char * message_id)
{
    char * var;
    var=(char *)malloc(sizeof(char)*11);
    strcpy(var,message_id);
    array_messages_names[inserted_messages]=var;
    printf("Ho inserito %s nell'array dei messaggi
                           alla posizione %ld\n",
                                    message_id,
                                inserted_messages);
    printf("Perciò ho : %s\n",
           array_messages_names[inserted_messages]);
    return;
}
```

Illustrazione 5.12

In pratica la funzione crea una variabile di tipo stringa lunga undici caratteri, ovvero la lunghezza necessaria per memorizzare l'identificativo del messaggio.
Tale identificativo viene passato in questa nuova area di memoria allocata, quindi aggiunto all'interno del vettore secondo l'indicazione del parametro in input che consiste nell'indice che deve essere occupato, ovvero il primo posto in cui non è ancora stato inserito nulla.
Restituisce quindi il controllo alla funzione che l'ha invocata ovvero Re_reading_of_trace_file().
In questa parte, figura 5.13, viene studiato il tipo di riga in cui è presente la stringa “RCV”. Vengono letti gli identificativi di messaggio e nodo, quindi attraverso le appossite funzioni Find_index_message() e Find_node_in_BST (), verranno ricavati gli indici corretti per posizionare il numero di hop.

```
if(strncmp(command,"STAT RCV",8)==0)
{
   memcpy(sender,buffer+9,10);
   sender[10]='\0';
   memcpy(messageid,buffer+20,10);
   messageid[10]='\0';
   index_column=Find_index_message(messageid,
                        array_messages_names,
                              tot_messages );
   pointer_to_BST_node=Find_node_in_BST(sender,BST);
   if(pointer_to_BST_node!=NULL)
   {
      index_row=pointer_to_BST_node->number;
   }
   else{
          printf("BST is not complete!\n");
          exit(1);
   }
   memcpy(number_of_hops_string,buffer+31,10);
   number_of_hops_string[10]='\0';
   number_of_hops=Conversion_String_to_number
                     (number_of_hops_string);
```

Illustrazione 5.13

La funzione Find_index_message(), figura 5.14, effettua una ricerca su vettore disordinato quindi lo scorre per intero, se necessario, fino a trovare l'elemento che viene passato come parametro. La funzione di ricerca si basa su un confronto mediante la funzione del linguaggio C, strcmp[23]. Tale funzione date due stringhe restituisce zero se le due stringhe sono uguali. Una volta trovato l'indice che soddisfa l'uguaglianza con il parametro di input “messageid”, lo restituisce come valore di output alla funzione che l'ha invocata.

```
long Find_index_message(char * messageid,
            char ** array_messages_names,
                        long Tot_messages)
{
   long i;
   for(i=0;i<=Tot_messages-1;i++)
   {
     if(strcmp(array_messages_names[i],messageid)==0)
     return i;
   }
   return -1;
}
```

Illustrazione 5.14

Per quanto riguarda la ricerca dell'indice di riga, ovvero quello del nodo, si utilizza la funzione in figura 5.13. In pratica tale funzione scorre l'abero alla ricerca dell'identificativo passato come parametro di input, in questo caso “sender”. Altro ovvio parametro di input è il riferimento all'albero.
Nell'illustrazione 5.15 si effettua un confronto fra il parametro passato in input che rappresenta l'identificativo del nodo e quello all'interno del nodo. Si effettua una traduzione da stringa a numero di entrambi i valori e poi si effettua un raffronto fra i due. Lo stesso identico risultato si ottiene confrontando direttamente le due stringhe attraverso la funzione strcmp(). La funzione di conversione infatti è superflua in questa parte del codice ma utile per convertire la stringa rappresentante il numero di hop in un numero su cui effettuare calcoli attraverso operatori aritmetici, come avviene anche nelle ultime righe della figura 5.13. Una volta trovate le coordinate di inserimento all'interno della matrice si procede con la conversione e l'inserimento all'interno della stessa. La parte di inserimento non è mostrata nell'immagine. Tale inserimento tiene conto

del tipo degli elementi della matrice istanziata.

Paragrafo 5.4

In questo paragrafo si illustrano altre funzioni create ed utilizzate nel codice. Di volta in volta verrà esplicitata la posizione del codice in cui la funzione viene invocata.

```
struct node * Find_node_in_BST(char *sender,
                               struct node * BST)
{
  struct node * Scan_BST=BST;
  long Conversion_number, Conversion_number1;
  Conversion_number=Conversion_String_to_number(sender);

  while((Scan_BST!=NULL))
  {
    Conversion_number1=Conversion_String_to_number
                      (Scan_BST->name_of_the_node);
    if(Conversion_number1==Conversion_number)
    {
      return Scan_BST;
    }
    else
    {
      if(Conversion_number1<Conversion_number)
      Scan_BST=Scan_BST->sx_daughter;
      else
      Scan_BST=Scan_BST->dx_daughter;
    }
  }
return Scan_BST;
}
```

Illustrazione 5.15

Paragrafo 5.4.0

Si inizia analizzando la funzione Discover_number_of_nodes_in_the_BST(). Tale funzione viene utilizzata all'interno del main() come mostra l'immagine 5.5.

La funzione viene mostrata tramite molteplici illustrazioni, esposte nell'ordine corretto.

Tale funzione, come specifcato nel commento sottostante il nome della funzione, ha come input il riferimento al BST, al quale dovrà accedere per poterne contare gli elementi e come output il numero di nodi contati. Oltre a contare il numero dei nodi tale funzione assegna ad ogni nodo un indice di riga.

Quest'ultima funzionalità si basa sull'osservazione che in qualche modo debba essere associato ogni nodo ad uno specifico indice di riga della matrice che verrà creata per contenere le informazioni, inoltre

ogni nodo viene contato una sola volta, nel momento in cui viene contato gli viene assegnato anche l'indice. Se si analizza il codice della funzione, partendo dal principio (illustrazione 5.16), si nota che viene controllato il riferimento al BST, se tale riferimento punta a NULL si tratterà di un BST vuoto poiché il puntatore al BST sarà inizializzato a NULL ma durante l'analisi dei trace file, dopo il primo nodo incontrato, il BST dovrà puntare a tale nodo.

Nel caso in cui BST non punti a NULL, quindi ha almeno un elemento, ma entrambe le sue figlie sono NULL, ovvero i puntatori interni all'unico nodo presente nell'albero puntano a NULL, si avrà un BST composto da un unico nodo. Perciò verrà restituito il valore uno.

Se invece la figlia sinistra non è uguale a NULL allora si scende lungo la sinistra. Scendere lungo la sinistra significa che la variabile puntatore, adibita ad analizzare l'albero, sarà posta uguale al valore, contenuto nel puntatore alla figlia sinistra, del nodo attualmente puntato.

Altrimenti si scende lungo la destra. Si inizializza quindi il counter dei nodi.

```
int Discover_number_of_nodes_in_the_BST(struct node * BST)
{
    long counter;
    struct node * scan_BST;
    if(BST==NULL)
    return 0;
    if(BST->dx_daughter==NULL && BST->sx_daughter==NULL)
    return 1;

    scan_BST=BST;

    if(scan_BST->sx_daughter!=NULL)
    scan_BST=scan_BST->sx_daughter;
    else
    scan_BST=scan_BST->dx_daughter;

    counter=0;
```

Illustrazione 5.16

Prima di continuare si evidenziano i punti principali dell'algoritmo implementato all'interno di questa funzione.
Si conta un nodo quando è possibile colorarlo.
Colorare un nodo significa modificare il campo color da zero a uno dello stesso.
E' possibile colorare un nodo solo quando tutte le figlie che ha sono colorate, quindi il loro campo color è settato ad uno. Nel caso in cui il nodo non abbia figlie può essere colorato.
I casi possibili sono illustrati dalle figure 5.18, 5.19 e 5.20.
Queste situazioni sono poi tradotte in codice , illustrazioni 5.17, 5.21 e 5.22.
Si parta dalla prima immagine, quella più in alto, all'interno della figura 5.18.

```
while(scan_BST!=BST)
{
  if(scan_BST->sx_daughter!=NULL &&
  scan_BST->dx_daughter!=NULL)
  {
    if(scan_BST->sx_daughter->color=='1' &&
         scan_BST->dx_daughter->color=='1')
    {
      scan_BST->number=counter;
      counter++;
      scan_BST->color='1';
      scan_BST=scan_BST->mother;
     }
     else
     {
       if(scan_BST->sx_daughter->color=='0')
       scan_BST=scan_BST->sx_daughter;
       else
       scan_BST=scan_BST->dx_daughter;
      }
  }
```

Illustrazione 5.17

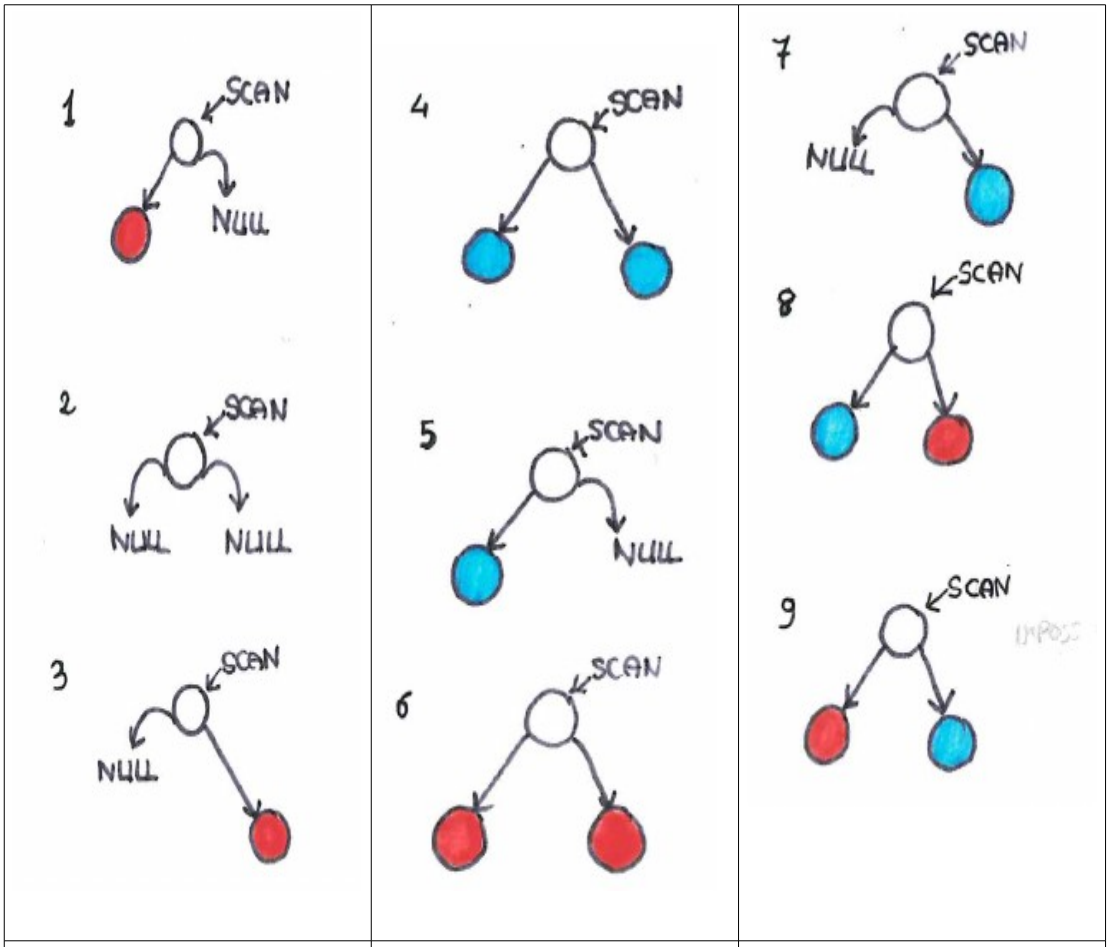


Illustrazione 5.18: rosso='1',blu='0'	Illustrazione 5.19	Illustrazione 5.20

Si ha che il puntatore per l'analisi del BST punta un certo nodo ancora non colorato. La situazione del nodo puntato è di avere una figlia sinistra rossa, quindi con color impostato ad uno, mentre la figlia destra è inesistente. In tale caso l'unica figlia che ha è colorata perciò si può procedere a colorare il nodo
stesso. Si guardi ora il codice dell'illustrazione 5.21. Il caso di questa immagine è individuato grazie alle seguenti condizioni:
1)scan_BST->dx_daughter==NULL && scan_BST->sx_daughter!

=NULL.
2)scan_BST->sx_daughter->color=='1'.
Ciò che fa la funzione in questo caso, come illustrato dall'immagine 5.21, è impostare l'indice di riga nel nodo, aumentare il numero di nodi contati, impostare il colore a '1' e risalire. Risale mediante il puntatore al nodo madre presente all'interno del nodo puntato.
Perciò, come già scritto, ogni situazione presente nei disegni è stata riportata nel codice. In realtà è inesatto in quanto non tutte le situazioni disegnate sono possibili. L'illustrazione numero 8 nella figura 5.20 non può essere ottenuta. L'algoritmo infatti si spinge il puntatore di analisi sempre prima a destra piuttosto che a sinistra. Si procede a destra solo nel caso in cui non ci sia una figlia a sinistra oppure sia già colorata. Si aggiunga che nessun nodo può essere rosso senza che il puntatore di analisi sia sceso nella sua direzione prendendolo direttamente in analisi. Dati questi elementi è ovvio che un nodo con entrambe le figlie non può avere la figlia sinistra blu e quella destra rossa.

Paragrafo 5.4.1
La prossima funzione che si procede ad analizzare è la funzione di creazione di un'istanza di struct node, illustrazione 5.23. Tale funzione viene utilizzata all'interno del main, tutte le volte che si deve aggiugere un nodo al BST. Propedeutica all'inserimento di un nodo vi è ovviamente la creazione dello stesso.
Il parametro richiesto in input è l'identificativo del nodo. L'output della funzione è il puntatore al nodo creato. Viene dichiarata la variabile Node che è un puntatore ad una struct node. In seguito tramite la funzione malloc viene allocato lo spazio necessario. La funzione malloc restituisce l'indirizzo di tale spazio e tale informazione viene memorizzata in Node.
Node ora punta al nuovo nodo sebbene tale nodo non contenga ancora le informazioni corrette.
Vi è un cotrollo sul valore di Node, nel caso in cui sia NULL non sarà possibile proseguire con degli inserimenti visto che non vi è alcuno spazio di memoria allocato per il nodo stesso.

Si copia quindi l'identificativo passato in input all'interno del campo name_of_the_node.
Si imposta il nodo in modo che non abbia figlie e il colore settato a blu che sarebbe il valore del campo color a zero. Infine si restituisce l'indirizzo in memoria dell'area appena allocata[25].

```
while(scan_BST!=BST)
{
  if(scan_BST->sx_daughter!=NULL &&
  scan_BST->dx_daughter!=NULL)
  {
    if(scan_BST->sx_daughter->color=='1' &&
         scan_BST->dx_daughter->color=='1')
    {
      scan_BST->number=counter;
      counter++;
      scan_BST->color='1';
      scan_BST=scan_BST->mother;
     }
     else
     {
       if(scan_BST->sx_daughter->color=='0')
       scan_BST=scan_BST->sx_daughter;
       else
       scan_BST=scan_BST->dx_daughter;
      }
  }
```

Illustrazione 5.21

```
else
{
  if(scan_BST->sx_daughter==NULL &&
       scan_BST->dx_daughter==NULL)
  {
    scan_BST->number=counter;
    counter++;
    scan_BST->color='1';
    scan_BST=scan_BST->mother;
   }
   else
   {
     if(scan_BST->dx_daughter==NULL &&
          scan_BST->sx_daughter!=NULL)
     {
       if(scan_BST->sx_daughter->color=='1')
       {
         scan_BST->number=counter;
         counter++;
         scan_BST->color='1';
         scan_BST=scan_BST->mother;
       }
       else
         scan_BST=scan_BST->sx_daughter;
     }
```

Illustrazione 5.22

```
        else
        {
          if(scan_BST->dx_daughter->color=='1')
          {
           scan_BST->number=counter;
           counter++;
           scan_BST->color='1';
           scan_BST=scan_BST->mother;
         }
         else
          scan_BST=scan_BST->dx_daughter;
        }
       }
      }
     }
    scan_BST->number=counter;
    BST->color='1';
    counter++;
    return counter;
```

Illustrazione 5.23

Paragrafo 5.4.2

La prossima funzione che si procede con l'analizzare è Insert_node_in_BST(), illustrazione 5.24.

Tale funzione ha il compito di inserire un nodo all'interno dell'albero binario di ricerca. Le informazioni in possesso sono il riferimento al nodo che deve essere inserito ed il riferimento al nodo che diventerà madre di questo nuovo elemento.

Vi è un controllo prima di effettuare l'inserimento.

Il controllo si basa sull'ordine presente nel BST. Se l'identificativo del nodo da inserire è maggiore di quello del nodo madre, allora se il nodo madre ha già un riferimento ad una figlia sinistra, ciò significa che non

potrà essere inserito il nuovo elemento. Se tale riferimento non è NULL si ha che nella ricerca della posizione corretta di inserimento del nodo qualcosa è andato storto.

```
int Insert_node_in_BST(struct node * In, struct node * mother)
{
   if(Conversion_String_to_number(In->name_of_the_node)
      >Conversion_String_to_number(mother->name_of_the_node))
   {
      if(mother->sx_daughter!=NULL)
      {
        printf("error , impossible to insert the node
                in a not empty space into the bst\n");
        return 1;
       }
       mother->sx_daughter=In;
   }
   else
   {
      if(mother->dx_daughter!=NULL)
      {
        printf("error , impossible to insert the
                node in a not empty space into the
                bst\n");
        return 1;
      }
      mother->dx_daughter=In;
   }
   In->mother=mother;
   return 0;
}
```

Illustrazione 5.24

Nel caso in cui sia NULL si effettua l'inserimento facendo puntare il nodo madre al nuovo elemento, tramite il campo sx_daughter. Lo stesso identico ragionamento viene fatto per il nodo a destra. Questo controllo una volta certi della correttezza dell'algoritmo è inutile. E' stato inserito al fine di individuare errori all'interno dell'algoritmo nella fase iniziale di sviluppo.

Paragrafo 5.4.3

Sono poi presenti due funzioni molto simili per la ricerca all'interno del BST. Le due funzioni sono Find_value_in_BST() e Find_node_in_BST(), illustrazione 5.25 e 5.15.
Entrambe ricevono lo stesso input ma hanno un output diverso. Find_value_in_BST() ha come output il puntatore alla futura madre del nodo, informazione che ne permetterà l'inserimento, o NULL. Verrà restituito NULL nel caso in cui sia già presente il nodo con l'identificativo cercato.
L'altra funzione, Find_node_in_BST invece, ha come output il puntatore al nodo contenente un certo identificativo.
La differenza è dovuta al contesto nel quale vengono usate. Find_value_in_BST() è la funzione usata durante l'inserimento dei valori nel BST. Find_node_in_BST() è la funzione usata dopo che il BST è stato creato e riempito di tutti i nodi presenti nei trace file. Serve a poter accedere alle informazioni del nodo con un certo identificativo. Durante la rilettura di un trace file infatti è necessario conoscere la riga d'inserimento. Tale riga di inserimento, come già detto, è memorizzata nel campo number del nodo. Ciò significa che quando si rilegge la seconda volta il trace file, capitando su una riga di tipo RCV, si ha l'identificativo del messaggio e quello del nodo. Grazie al vettore dei messaggi ed alla ricerca che si può effettuare su di esso, si avrà l'indice di quel messaggio, che sarebbe l'indice di colonna; grazie al BST si ha l'indice del nodo, che sarebbe l'indice di riga.

```
struct node * Find_value_in_BST(char * String_number,
                                struct node * BST)
{
  struct node * Scan_BST=BST , * Previous=BST;
  long Conversion_number, Conversion_number1;
  Conversion_number=Conversion_String_to_number(String_number);
  while((Scan_BST!=NULL))
  {
    Conversion_number1=Conversion_String_to_number
                    (Scan_BST->name_of_the_node);
    if(Conversion_number1==Conversion_number)
    {
      return NULL;
    }
    else
    {
      Previous=Scan_BST;
      if(Conversion_number1<Conversion_number)
      Scan_BST=Scan_BST->sx_daughter;
      else
      Scan_BST=Scan_BST->dx_daughter;
    }
  }
return Previous;
}
```

Illustrazione 5.25

Paragrafo 5.4.4

Quest'ultimo paragrafo tratta la parte di codice ripresa dalle illustrazioni 5.26, 5.27, 5.28. Tale parte di codice permette di calcolare le grandezze che verranno poi copiate sugli appositi file. Quindi riguarda il calcolo di copertura e ritardo. Come si può osservare dall'immagine i parametri di input sono il puntatore alla matrice, il numero di messaggi e di nodi, il tipo di matrice, i puntatori alle variabili che conterranno le grandezze.

Come nei casi precedenti, il riferimento alla matrice, il numero di messaggi ed il numero di nodi sono necessari ai fini della lettura di informazioni consistenti memorizzate nella matrice.

```
void Compute_coverage_and_delay(void * M,long num_messages,
                                        long num_nodes,
                                      float * coverage,
                                         float * delay,
                                           char * type)
{
  int tot_pairs=num_messages*num_nodes, received_pairs=0;
  int i,j;
  if(strcmp(type,"short")==0)
  {
    short  sum_delayS=0;
    for(i=0;i<=num_nodes-1;i++)
    {
      for(j=0;j<=num_messages-1;j++)
      {
        if((((short*)M)[i*num_messages+j])!=-1)
        {
          received_pairs++;
          sum_delayS=((short *)M)[i*num_messages+j]+sum_delayS;
        }
      }
    }
    *coverage=(float)received_pairs*100/tot_pairs;
    *delay=(float)sum_delayS/received_pairs;
  }
```

Illustrazione 5.26

```
if(strcmp(type,"int")==0)
{
  int sum_delayI=0;
  for(i=0;i<=num_nodes-1;i++)
  {
    for(j=0;j<=num_messages-1;j++)
    {
      if((((int *)M)[i*num_messages+j])!=-1)
      {
        received_pairs++;
        sum_delayI=((int *)M)[i*num_messages+j]+sum_delayI;
      }
    }
  }
  *coverage=(float)received_pairs*100/tot_pairs;
  *delay=(float)sum_delayI/received_pairs;
}
```

Illustrazione 5.27

```
  if(strcmp(type,"long")==0)
  {
    long sum_delayL=0;
    for(i=0;i<=num_nodes-1;i++)
    {
      for(j=0;j<=num_messages-1;j++)
      {
        if((((long *)M)[i*num_messages+j])!=-1)
        {
          received_pairs++;
          sum_delayL=((long *)M)[i*num_messages+j]+sum_delayL;
        }
      }
    }
    *coverage=(float)received_pairs*100/tot_pairs;
    *delay=(float)sum_delayL/received_pairs;
  }
return;
}
```

Illustrazione 5.28

CAPITOLO 6

ESECUZIONE PARALLELA

Paragrafo 6.0

Questo paragrafo fornisce la basi per poter affrontare gli argomenti successivi all'interno di questo capitolo.

Paragrafo 6.0.0

Il filesystem di UNIX è un insieme di file e directory basato sulla struttura ad albero.[31]
Il nome di un file all'interno del file system, è costituito dalle varie parti del percorso. [30] Un veloce esempio: /home/Desktop/filename1.txt è il nome di un file, /home/filename1.txt è il nome di un altro file. Quindi si analizza il nome completo per stabilire se si sta parlando dello stesso oggetto.

Paragrafo 6.0.1

Ritornando al progetto quel che si vuole fare è creare un nuovo file get_coverage_script.sh (script), si ricordi che la versione precedente è stata analizzata nel capitolo cinque sul codice preesistente, in modo che un solo un numero prestabilito di script esegua la parte di accesso ai trace file in maniera concorrente. Prima di proseguire si osservi che il file get_coverage_and_delay.c si può suddividere in due parti, seguento la logica di accesso alla memoria secondaria. Ovvero, la prima parte accede alla memoria secondaria mentre la seconda no. La prima parte è la parte in cui vengono analizzati i trace file. E' composta da due letture di ognuno dei file di traccia contenuti nella directory passata in input. Successivamente alla prima parte si ha il codice che lavora esclusivamente sulla matrice creata, calcolandone delay e coverage. Per ulteriori chiarimenti è consultabile il capitolo sei che mostra nel dettaglio il codice scritto in get_coverage_and_delay.c.

Data tale suddivisione, si preferisce che siano limitate le richieste di accesso ai file in memoria secondaria, da parte delle varie istanze di get_coverage_and_delay.c. Troppe richieste alla memoria secondaria provocano un degrado delle prestazioni.
Avere processi che concorrentemente tentano l'accesso in memoria secondara quindi significa che vi sono richieste di accesso alla risorsa nonostante il primo processo ad aver fatto richiesta non abbia ancora finito. Riepilogando si ha sim-metrics-*-corpus che lancia un numero inferiore o uguale al numero di unità di elaborazione, di istanze di get_coverage_script. Quel che si vuole è che non tutte ma le istanze di get_coverage_script tentino di accedere alla memoria secondaria in modo concorrente, ma solo un numero stabilito.

Paragrafo 6.1
Il file get_coverage_script.sh deve perciò lasciare che solo K processi eseguano concorrentemente il processo istanza di get_coverage_and_delay.c. Ciò che accade per implementare tale caratteristica è illustrato quì di seguito.
Vi sono un massimo di cartelle che possono essere create, tale massimo è K. Ogni processo istanza di get_coverage_script.sh può lanciare il processo istanza di get_coverage_and_delay.c solamente nel caso in cui abbia creato una delle K cartelle. Una volta che quest'ultimo processo avrà finito di accedere alla memoria secondaria, comunicherà ai restanti processi di aver raggiunto la fase di elaborazione delle informazioni e che quindi è possibile cancellare la cartella relativa alla sua esecuzione.
La comunicazione ai restanti avviene usando un file.
Perciò in termini un po' più astratti quel che accade è che il processo A crea una cartella con indice t, 1<=t<=K ovvero t minore o uguale a K e maggiore o uguale a 1. Se A riesce a creare tale cartella lancia il processo C. Se non riesce entra in un ciclo che gli permette di domandare, ogni intervallo I di tempo, se una certa cartella può essere creata. Tale cartella non è sempre la stessa, ma ogni volta il processo proverà a creare quella di indice successivo all'indice della pecedente. Una volta arrivati

all'indice maggiore possibile per una cartella si torna a fare richiesta per il minimo indice possibile. (Si ricorda infatti che gli indici non sono arbitrari in quanto le cartelle sono al massimo K). Perciò se è avvenuta la creazione di tre cartelle significa che tre processi stanno eseguendo la parte che concerne anche la lettura dei trace file.
Ritornando nel caso specifico del processo C che è stato lanciato dal processo A, si ha che prima che il processo C venga lanciato il processo A crea un file che viene quindi passato fra i parametri di input del processo C. Il processo C una volta terminata la parte di accesso alla memoria secondaria cancella il file. A intanto continua a chiedersi a intervalli regolari, se il file passato in input sia stato o meno cancellato. Tale domanda equivale a chiedersi se il processo C abbia finito di accedere alla memoria secondaria e se sia quindi possibile cancellare la cartella creata in precedenza. Una volta che tale cartella verrà cancellata un nuovo processo potrà essere lanciato e quindi eseguire la parte relativa all'accesso alla memoria secondaria. La prima versione è riportata nell'illustrazione 6.0.
La seconda illustrazione, 6.1 è un programma di prova che viene lanciato dallo script.

Paragrafo 6.2
In questo paragrafo viene spiegato il contenuto del file scritto in C riportato nell'immagine 6.1. In particolare verrà posta l'attenzione su una riga del file e di come è stata inserita all'interno del file del progetto get_coverage_and_delay.c.
La funzione viene chiamata dallo script di figura 6.0 una volta che quest'ultimo è riuscito a creare una cartella. Lo script passa come parametro, alla funzione, il nome di un file. Tale file è stato creato dallo script stesso ed ha indice identico a quello della cartella creata in precedenza. In questo modo i processi non tenteranno di generare file con lo stesso nome. Nel caso due processi tentassero di fare ciò, si avrebbe che il primo processo crea il file, supponiamo si chiami file1.txt, mentre l'altro non potrebbe crearlo in quanto già esistente. A questo punto uno

solo dei due processi potrà comunicare, al processo che l'ha lanciato, di aver finito di eseguire la parte che concerne gli accessi alla memoria secondaria. Per questa ragione quindi l'indice della cartella sarà lo stesso del file.

```
#! /bin/bash

k=2
i=1
trovato=0
CURDIR=$PWD

while [ $trovato -eq 0 ]
do
echo iterazione
let successivo=k+1
if mkdir -p $CURDIR/cartella$i ; then
  let trovato=trovato+1
else
let i=i+1
if [ $i -eq $successivo ] ; then
let i=1
sleep 3
fi
fi
done
fatto=0
touch $CURDIR/filename$i.txt
./processo_c "$CURDIR/filename$i.txt" &
while [  $fatto -ne 1 ]; do
if ! [ -f "$CURDIR/filename$i.txt" ]; then
echo è stato rimosso filename$i
rmdir $CURDIR/cartella$i
let fatto=1
else
sleep 3
fi
```

Illustrazione 6.0: prima versione dello script

```c
#include <stdio.h>
#include <string.h>
void main(int argc, char * argv[])
{
        FILE * fp;
        char numero;
        char * filename,buffer[20],filename2[10];
        int a=0;
        filename=argv[1];
        numero=argv[2];
        strcpy(filename2, "filename");
        filename2[8]=numero;
        filename2[9]='\0';
        fp=fopen(filename2,"r");
        fprintf(filename2,"%d",a);
        printf("Ecco\n");
        printf("\n%s\n",filename);
        int f=2000;
        int bo;
        while(f>0)
        {
          printf("inizio a dormire\n");
          sleep(0.1);
          fscanf(filename2,"%d",&bo);
          printf("mi sveglio\n");
          f--;
        }
        remove(filename);
        //eventuale parte 2
        int var;
        f=300;
        while(f>0){
        var ++;
        sleep(0.1);
        }
       return;
```

Illustrazione 6. 1: file in c

Un modo alternativo potrebbe essere quello di creare il file sempre con lo stesso nome e posizionarlo all'interno della cartella creata appositamente per l'esecuzione del tal processo, sempre per la logica già descritta.

Paragrafo 6.2

In questo paragrafo vengono illustrati i vari cambiamenti che hanno portato alla modifica della prima versione dello script.

```
messages_dir = argv[1];
LPs = atoi(argv[2]);
coverage_file = argv[3];
delay_file = argv[4];
filename=argv[5];
```

Illustrazione 6.2: argomenti passati in input a get_coverage_and_delay.c

Osservando l'illustrazione 6.2 si può notare che la funzione get_coverage_and_delay.c deve ricevere cinque argomenti in input. Il primo è il nome della cartella contenente i trace file, il secondo è il numero di file all'interno di tale cartella, il terzo è il nome del file nel quale scrivere il valore di copertura calcolato, il quarto è il valore del file nel quale scrivere il valore del ritardo calcolato infine il quinto è il nome del file che verrà cancellato dal processo una volta terminata la prima parte dell'elaborazione. Nell'illustrazione 6.3 viene mostrata la parte del codice di get_coverage_and_ delay.c in cui è presente la cancellazione del file passato in input. La prima lettura è stata già precedentemente effettuata, infatti la seconda riga riguarda la ri-lettura del file, quindi la seconda lettura. In seguito alla seconda lettura, per quanto visto nel capitolo sei, si ha la matrice con all'interno le informazioni da elaborare.
Quindi la fase di lettura dei file è l'ultima funzione che deve manipolare i trace file.

```
if(Short_Matrix!=NULL)
{
  Re_reading_of_trace_file( pointer_to_trace_file,
                                  Short_Matrix,BST,
                              number_messages,type,
                              array_messages_names,
                                 messages_dir,LPs);
  remove(filename);
  Print_the_Matrix(Short_Matrix,number_nodes,
                          number_messages,type);
  Compute_coverage_and_delay(Short_Matrix,
                             number_messages,
                                number_nodes,
                    &coverage,&delay,type);
}
```

```
if(Int_Matrix!=NULL)
{
  Re_reading_of_trace_file( pointer_to_trace_file,
                                 Int_Matrix,BST,
                              number_messages,type,
                              array_messages_names,
                                 messages_dir,LPs);
  remove(filename);
  Print_the_Matrix(Int_Matrix,number_nodes,
                          number_messages,type);
  Compute_coverage_and_delay(Int_Matrix,
                             number_messages,
                                number_nodes,
                    &coverage,&delay,type);
}
```

```
if(Long_Matrix!=NULL)
{
  Re_reading_of_trace_file( pointer_to_trace_file,
                                  Long_Matrix,BST,
                              number_messages,type,
                              array_messages_names,
                                 messages_dir,LPs);
  remove(filename);
  Print_the_Matrix(Long_Matrix,number_nodes,
                         number_messages,type);
  Compute_coverage_and_delay(Long_Matrix,
                             number_messages,
                                number_nodes,
                    &coverage,&delay,type);
}
```

Illustrazione 6.3: get_coverage_and_delay.c, parti di rimozione del file

Dopo tale funzione viene quindi rimosso il file, comunicando al processo

che la prima fase è terminata e che si può quindi cancellare una cartella.
Si ponga quindi l'attenzione sull'illustrazione 6.5, chiaramente il processo da lanciare è ora un altro processo, quindi non più processo_c con input il solo nome del file di controllo del numero di processi paralleli (ovvero il parametro cinque di get_coverage_and_delay.c), ma è get_coverage_and_delay con i suoi cinque parametri di input.

```
#! /bin/bash
source scripts_configuration.sh
SIMTRACEDIR=$1
RUNS=$2
TESTNAME=$3
LPS=$4
CURDIR=$PWD
rm -fr $WORKING_DIRECTORY/$TESTNAME/$RUNS
mkdir -p $WORKING_DIRECTORY/$TESTNAME/$RUNS
cd $WORKING_DIRECTORY/$TESTNAME/$RUNS
k=10
i=1
trovato=0
while [ $trovato -eq 0 ]
do
echo iterazione
let successivo=k+1
if  mkdir $CURDIR/cartella$i ; then
  let trovato=trovato+1
else
let i=i+1
if [ $i -eq $successivo ] ; then
let i=1
sleep 3
fi
fi
done
```

Illustrazione 6.4: bash finale, parte1

Scritto ciò rimane solo da chiarire che l'ultima versione dello script illustrata dalle immagini 6.4 e 6.5, comprende alcune parti del vecchio script come ad esempio i parametri di input dello script, il giusto processo da lanciare con i giusti parametri come scritto sopra ed infine l'algoritmo illustrato dalla 6.0. Il numero di processi da eseguire è stato fissato a 10. Chiaramente per modificare tale valore deve essere cambiata la variabile K. Ultima nota, si noti che nella creazione di file e directory viene utilizzata la variabile CURDIR, serve a fare in modo che tutti i file e le cartelle vengano creati nello stesso spazio. Le cartelle devono necessariamente essere create nello stesso spazio altrimenti dati path differenti, possono essere create più cartelle con parte finale del nome uguale [30], ad esempio, a “cartella1”. In questo modo ci saranno un numero arbitrario di processi paralleli e non limitato al K fissato.

```
fatto=0
touch $CURDIR/filename$i.txt
$CURDIR/./get_coverage_and_delay $SIMTRACEDIR $LPS \
$WORKING_DIRECTORY/$TESTNAME/$RUNS/$RUNSCOVERAGETMP \
$WORKING_DIRECTORY/$TESTNAME/$RUNS/$RUNSDELAYTMP \
"$CURDIR/filename$i.txt" &
while [  $fatto -ne 1 ]; do
 if ! [ -f "$CURDIR/filename$i.txt" ]; then
echo è stato rimosso filename$i
rmdir $CURDIR/cartella$i
let fatto=1
else
sleep 3
fi
done
touch $WORKING_DIRECTORY/$TESTNAME/$RUN/$TESTNAME-$RUNS.finished
```

Illustrazione 6.5: bash finale, parte 2

CAPITOLO 7

STABILIRE IL NUMERO DI PROCESSI PARALLELI

Paragrafo 7.0

In questo paragrafo si danno delle definizioni relativamente a termini che verranno usati successivamente.

Paragrafo 7.0.0

Memoria principale: “La memoria che il calcolatore usa per conservare il programma attualmente in esecuzione e che il programma usa durante l'esecuzione per le operazioni intermedie è detta memoria principale.”[32]
L'accesso alla memoria principale da parte della CPU è molto più veloce rispetto a quello relativo alla memoria secondaria. Per questa ragione quando un programma viene avviato una copia viene creata all'interno della RAM. [33]

Paragrafo 7.0.1

Memoria secondaria: “La memoria ausiliaria o secondaria è utilizzata per conservare programmi e dati in modo più o meno permanente. Questa memoria consente una registrazione dei dati meno costosa e che perdura anche in assenza di elettricità”.[32]

Paragrafo 7.0.2

Memoria di swap: “Le partizioni di swap consentono di avere una memoria virtuale, ossia una risorsa di memoria aggiuntiva ottenuta utilizzando parte dello spazio del disco come un'estensione della memoria (RAM). Questa partizione prende il nome di spazio di swap

perchè il sistema la può usare per scambiare informazioni (il termine inglese “swap” significa proprio “scambio”) tra il disco e la RAM. L'impostazione dello spazio di swap permette al computer di offrire maggiore velocità ed efficienza”.[34]

Paragrafo 7.0.3

Page fault: Il page fault è un'eccezione generata quando un processo cerca di accedere ad una pagina che è mappata nello spazio di indirizzamento virtuale, ma non ancora caricata nella memoria fisica. Tipicamente il sistema operativo tenta di risolvere il page fault rendendo accessibile la pagina richiesta o in alternativa terminando la richiesta, nel caso in cui essa sia “illegale”. Quindi dopo il verificarsi di un page fault fra le vari operazioni effettuate vi è la ricerca di un frame libero e caricamento della pagina nella memoria fisica, in uno spazio libero o sostituendola con un'altra pagina tramite uno swap. La scelta della pagina che verrà sostituita è dovuto al particolare algoritmo col quale opera il sistema.[35]

Paragrafo 7.0.4

Trashing o paginazione degenere: “La degenerazione dell'attività di paginazione causa parecchi problemi di prestazioni. Si considera il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione. Il sistema operativo vigila sull'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Per ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di assenze di pagine , cui segue la sottrazione di nuove pagine ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi assenze di pagine, con conseguente sottrazione di pagine ad altri processi”. [36]

Paragrafo 7.1
In questo paragrafo vengono effettuate alcune osservazioni inerenti all'esecuzione parallela dei processi ed ai possibili cali di prestazioni.
Come visto nel capitolo precedente, si ha la possibilità di eseguire in parallelo un certo numero di processi. Il parallelismo riguarda però solo la prima parte , tale parte di codice accede alla memoria secondaria per leggere i file di traccia. Quello che avviene quindi è che un certo numero di processi vengono caricati in memoria e vengono anche caricati, i dati che risiedono in memoria secondaria utili e necessari all'elaborazione di tali processi. Ogni singola istanza del programma get_coverage_and_delay.c necessita infatti l'accesso alle informazioni contenute nei trace file, per quanto già scritto nei capitoli di analisi dei compiti svolti da tale porzione di codice.
Per quanto scritto sopra nel primo paragrafo, quello che si cerca di evitare è che ci siano troppi page fault. Nel seguito si propone un metodo che cerca di dare un'approssimazione del numero di processi da lanciare in esecuzione parallela.

Paragrafo 7.2
In questo paragrafo si analizzano tutti i comandi utilizzati nel paragrafo successivo. Per una descrizione dettagliata di tutti i vari comandi è possibile consultare le “manual pages”.[37]

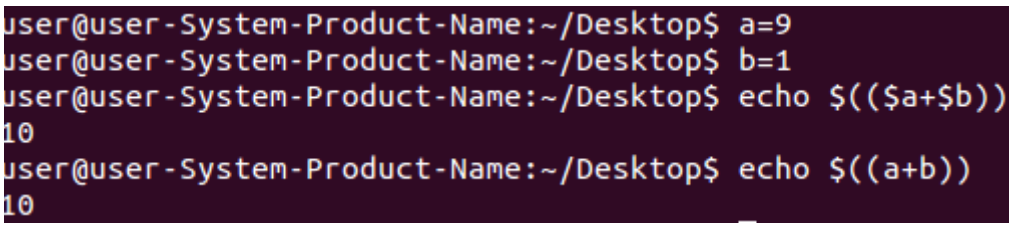


Illustrazione 7.0

L'illustrazione 7.0 mostra come si possano effettuare le operazioni aritmetiche fra due variabili all'interno dell'esecuzione di una shell.
Sono presenti due possibili sintassi per farlo. Per quanto riguarda la sintassi delle altre operazioni aritmetiche è la stessa di quella riportata, dove al posto dell'operazione di somma si digita l'operatore dell'operazione scelta: - , sottrazione, / divisione, * moltiplicazione.

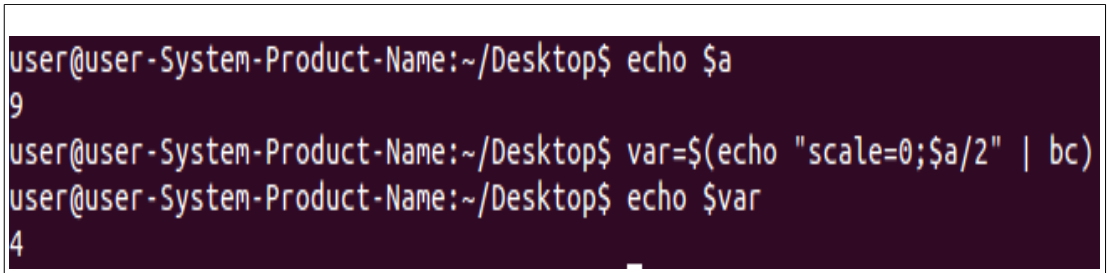


Illustrazione 7.1

L'illustrazione 7.1 mostra il comando utilizzato per selezionare il numero, risultato di un'operazione in questo caso “$a/2”, escludendone le cifre decimali, tramite “scale=0” , 0 è il numero di cifre decimali che si vogliono mantenere.

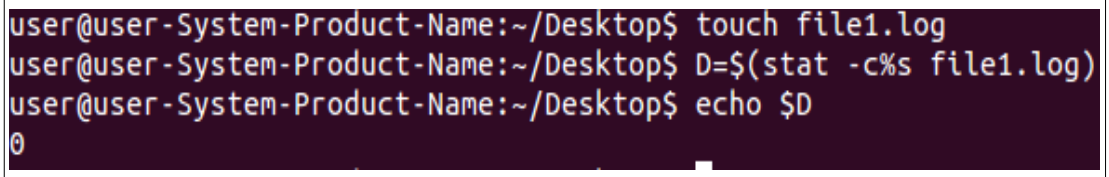


Illustrazione 7.2

L'illustrazione 7.2 mostra la creazione di un file e la stima della sua grandezza, tale grandezza viene memorizzata in D. Inizialmente il file è vuoto. Successivamente viene manipolato inserendo al suo interno dei

dati. L'illustrazione 7.3 mostra la dimensione del file in seguito alla scrittura di informazioni al suo interno.

```
user@user-System-Product-Name:~/Desktop$ D=$(stat -c%s file1.log)
user@user-System-Product-Name:~/Desktop$ echo $D
147
```

Illustrazione 7.3

```
user@user-System-Product-Name:~/Desktop$ free -b
             total       used       free     shared    buffers     cached
Mem:    16511733760 2318180352 14193553408          0  153198592 1287819264
-/+ buffers/cache:  877162496 15634571264
Swap:    3970953216          0 3970953216
```

Illustrazione 7.4

L'illustrazione 7.4 mostra l'output della funzione free che illustra, fra le tante informazioni, la quantità di memoria libera. La memoria rappresentata è quella principale, quella di buffer e cache infine quella di swap. E' di interesse la quantità di memoria RAM libera ovvero: 14193553408.

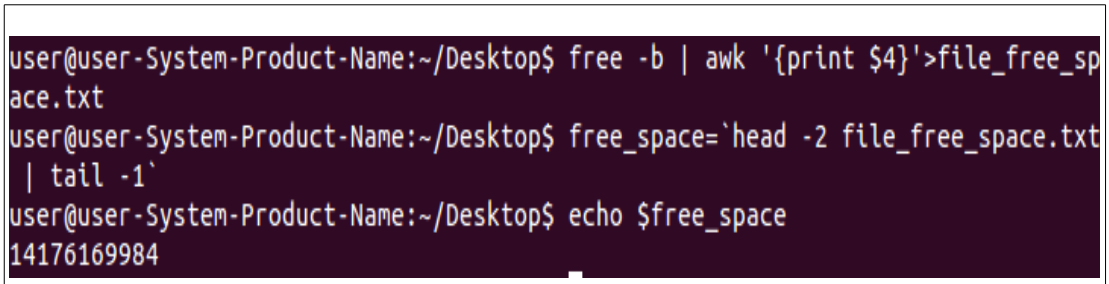


Illustrazione 7.5

L'illustrazione 7.5 mostra il modo utilizzato per porre il valore della memoria disponibile all'interno della variabile di nome free_space.
Notare che il valore trovato differisce un po' da quello visualizzato nell'immagine precedente. Il computer infatti elabora continuamente ed è naturale che tale valore cambi.

Paragrafo 7.3
L'illustrazione 7.6 mostra la prima parte del codice relativo alla scelta del numero di processi da mandare in esecuzione.
Ciò che viene fatto è calcolare la dimensione media di tutti i trace file scorrendo i file di tutte le cartelle che verranno analizzate ciascuna da un differente processo, per quanto già visto nei capitoli precedenti.
L'illustrazione 7.7 mostra la seconda parte del codice. Viene calcolata la dimensione media di un trace file, la quale si usa come approssimazione dello spazio che si vuole lasciare ad un singolo processo, memorizzata in M. A questo punto viene calcolato lo spazio libero di RAM. Per sapere quanti processi debbano essere lanciati in parallelo si calcola quante volte, l'approssimazione di spazio necessario calcolata, sia contenuta nella parte libera di memoria.
Tale quantità viene posta nella variabile number_parallel_processes.
Nel caso in cui tale numero sia inferiore a uno, si lancierà un solo processo. Naturalmente tale valore “di soglia minima” può essere modificato a seconda del valore minimo di processi che si vuole che

partano.
Tale parte di codice viene inserita nei file sim-metrics-*-corpus.
Il punto di inserimento è mostrato dall'illustrazione 7.8.

```
RUN=1
SUM=0
while [ $RUN -le $NUMBERRUNS ]; do

   #calculate average of dimensions
   #for every run take every file in the folder using lps
        lp=0
        M=0
        while [ $(($lp+1)) -le $LPS ]; do
                filename=$TRACE_DIRECTORY/$TESTNAME/$RUN/SIM_TRACE_$lp.log
                D=$(stat -c%s $filename)
                M=$(($M+$D))
                let lp=lp+1
        done
        SUM=$(($M+$SUM))

    let RUN=RUN+1

done
```

Illustrazione 7.6

```
tot_num_files=$(($NUMBERRUNS*$LPS))
#average value of dimension
M=$(echo "scale=0;$SUM/$tot_num_files" | bc)

touch file_free_space.txt
free -b | awk '{print $4}' >file_free_space.txt
#free_space is a new variable
free_space=`head -2 file_free_space.txt | tail -1`

#number of parallel processes that have access to the secondary storage
number_parallel_processes=$(echo "scale=0;$free_space/$M" | bc)

if [ $number_parallel_processes -le 1 ]; then
number_parallel_processes=1
fi
```

Illustrazione 7.7

```
while [ $RUN -le $NUMBERRUNS ]; do

RUNNING=`ps aux 2> /dev/null | grep "get_coverage_script" | wc -l`
echo $RUN / $RUNNING

  if [ $RUNNING -le $CPUNUM ]; then
   mkdir -p $RESULTS_DIRECTORY/$TESTNAME/$RUN
   ./get_coverage_script "$TRACE_DIRECTORY/$TESTNAME/$RUN"
   $RUN $TESTNAME $LPS &
   let RUN=RUN+1
  else
   sleep 3
  fi
done
```

Illustrazione 7.8

Paragrafo 7.4
In questo paragrafo si riportano il risultato di un semplice test relativo alla porzione di codice presentata sopra.

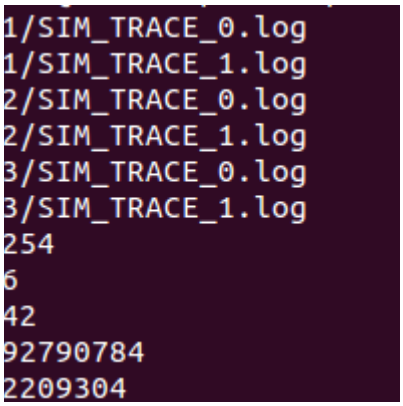


Illustrazione 7.9

```
leo@Kevin:~/Desktop$ free -b
             total       used       free     shared    buffers     cached
Mem:    1040388096  940744704   99643392          0   13955072  406650880
-/+ buffers/cache:  520138752  520249344
Swap:   1062203392   62439424  999763968
```

Illustrazione 7.10

Sono stati creati un totale di sei file in tre cartelle contenenti ciascuna due file. I nomi di questi file sono mostrati nell'illustrazione 7.9.
Le dimensioni di tali file, sono , in ordine : 2 byte, 8 byte, 135 byte, 88 byte, 21 byte, 0 byte.
La somma di queste dimensioni è quindi 254 byte, sempre indicato dall'illustrazione 7.9. Il numero successivo indica il numero dei file, si

tratta della stampa del valore della variabile tot_num_files, illustrazione 7.7. Il numero successivo è la media delle dimensioni dei file esclusi i valori dopo la virgola (il risultato comprendente le cifre decimali è 42.333333333). Il valore successivo è lo spazio libero in memoria misurato in byte, guardare in proposito l'illustrazione 7.10. Infine il numero di processi che possono essere eseguiti data la media dimensione di un trace file e la memoria libera, per quanto visto sopra.

CAPITOLO 8

TEST DELL'ALGORITMO

Paragrafo 8.0

In questo capitolo viene mostrata l'esecuzione di un test su trace file di dimesioni molto ridotte per mostrare quali siano i risultati prodotti dalla parte della tesi in linguaggio C, ovvero dal file get_coverage_and_delay.c.
Nei successivi paragrafi vengono mostrate le semplicissime strutture create, di cartelle e file, sui quali lavorerà l'algoritmo.
La parte testata è relativa al lancio del file get_coverage_script il quale lancia il file get_coverage_and_delay. Gli esempi mostrati successivamente sono due in totale. Nel primo verrà lanciata una sola istanza dello script get_coverage_script, in questo modo si focalizzerà l'attenzione sul risultato dell'algoritmo presentato nel file get_coverage_and_delay.c. Nel secondo esempio vengono lanciate più istanze dello script.

Paragrafo 8.1

In questo paragrafo viene presentata la struttura necessaria all'avvio del primo esempio.

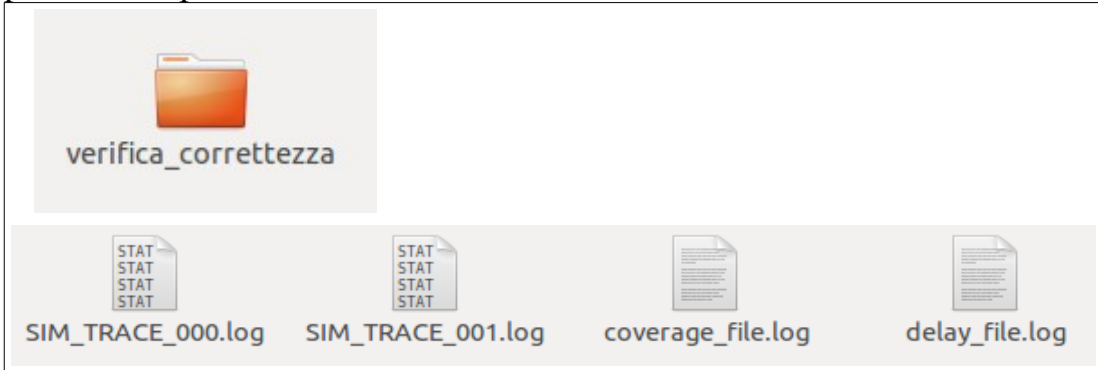


Illustrazione 8.0

```
messaggi: 1,2,3,44,9,7,230,70
nodi:42,30,20,41,43
matrice
     1     2     3       44       9       7      230     70

43   -1    -1    -1       -1      -1      -1       8     -1

41   -1    -1    -1       -1      -1      -1       0     -1

20   -1    -1    -1        1      -1      -1      -1     -1

30   20    10    -1       -1      80      -1      -1      -1

42   0     -1    -1       -1      -1      -1      -1      -1
```

Illustrazione 8.1

```
copertura: 1+3+1+1+1/(8*5)=7/40=0.175 *100=17.5 %
ritardo:0+20+10+80+1+0+8=20+10+80+1+8=119-> ritardo medio 119/7=17.00
```

Illustrazione 8.2

L'illustrazione 8.0 mostra una cartella ed il suo contenuto.
L'illustrazione 8.1 mostra la totalità di informazioni che verranno calcolate, la matrice non avrà necessariamente le righe nell'ordine proposto. Le prime due righe elencano gli id di messaggi e nodi.
L'illustrazione 8.2 mostra invece i valori che ci si aspetta di ricevere dall'algoritmo, quindi sono i calcoli di coverage e delay che verranno memorizzati sugli appositi file di output.

Paragrafo 8.2
In questo paragrafo viene descritto cosa accade nel primo esempio svolto.

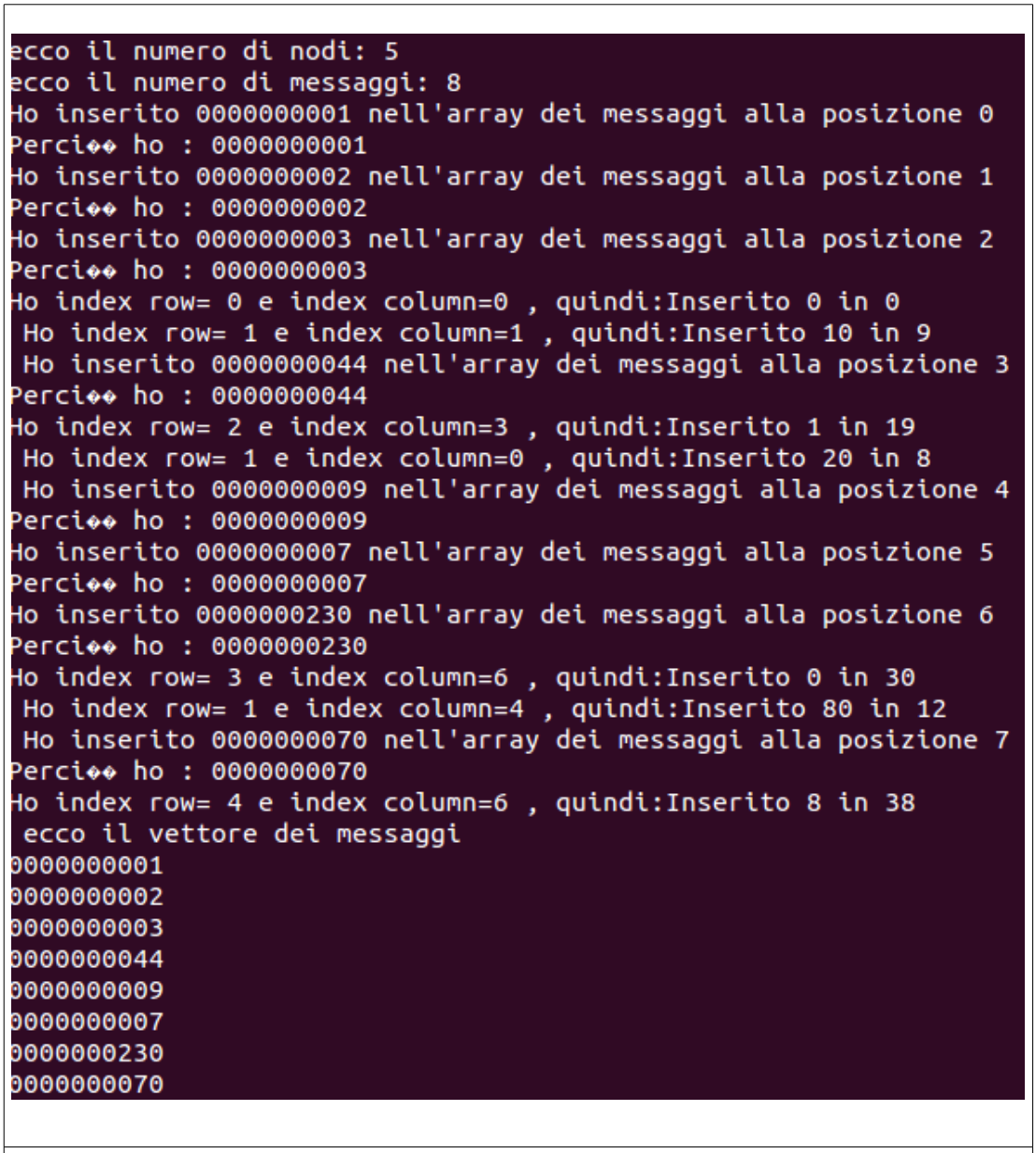


Illustrazione 8.3

L'illustrazione 8.3 mostra alcuni dati stampati dall'applicazione: l'inserimento dei vari elementi all'interno della matrice ed il vettore dei

messaggi.

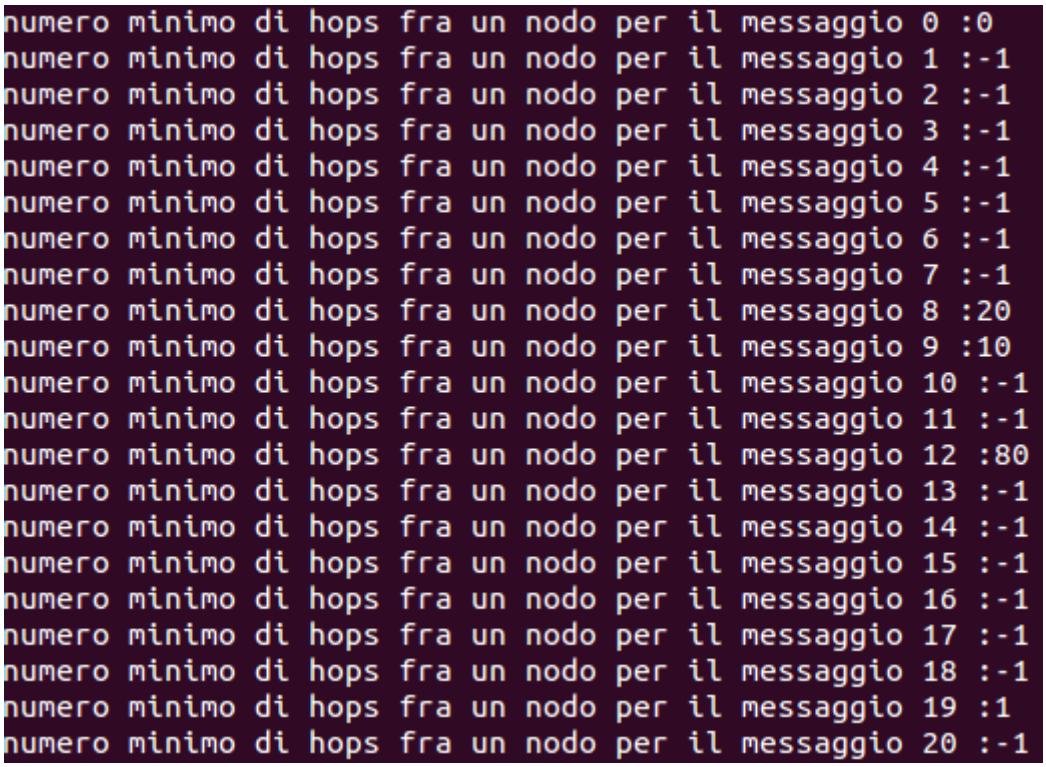


Illustrazione 8.4

L'illustrazione 8.4 e 8.5 mostrano i valori inseriti. Sono tutti i valori a partire dal primo elemento della prima riga della matrice, fino all'ultimo elemento dell'ultima riga della matrice.
Le illustrazioni 8.6 e 8.7 mostrano i valori calcolati per il ritardo e per la copertura, memorizzati nei relativi file.

```
numero minimo di hops fra un nodo per il messaggio 21 :-1
numero minimo di hops fra un nodo per il messaggio 22 :-1
numero minimo di hops fra un nodo per il messaggio 23 :-1
numero minimo di hops fra un nodo per il messaggio 24 :-1
numero minimo di hops fra un nodo per il messaggio 25 :-1
numero minimo di hops fra un nodo per il messaggio 26 :-1
numero minimo di hops fra un nodo per il messaggio 27 :-1
numero minimo di hops fra un nodo per il messaggio 28 :-1
numero minimo di hops fra un nodo per il messaggio 29 :-1
numero minimo di hops fra un nodo per il messaggio 30 :0
numero minimo di hops fra un nodo per il messaggio 31 :-1
numero minimo di hops fra un nodo per il messaggio 32 :-1
numero minimo di hops fra un nodo per il messaggio 33 :-1
numero minimo di hops fra un nodo per il messaggio 34 :-1
numero minimo di hops fra un nodo per il messaggio 35 :-1
numero minimo di hops fra un nodo per il messaggio 36 :-1
numero minimo di hops fra un nodo per il messaggio 37 :-1
numero minimo di hops fra un nodo per il messaggio 38 :8
numero minimo di hops fra un nodo per il messaggio 39 :-1
◆◆ stato rimosso filename1
```

Illustrazione 8.5

Illustrazione 8.6

Illustrazione 8.7

Paragrafo 8.3

In questo paragrafo viene descritto cosa accade nel secondo esempio svolto. Il secondo esempio anch'esso molto semplice, consiste nel lanciare due istanze in modo parallelo. In questo caso le istanze lavoreranno sugli stessi elementi.

Illustrazione 8.8

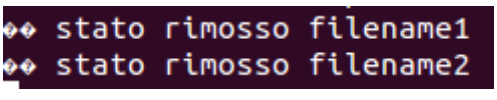


Illustrazione 8.9

A differenza dell'illustrazione 8.5 nella quale il file cancellato è uno solo, l'illustrazione 8.9 mostra come i file cancellati ora siano due. Uno viene cancellato dal primo processo lanciato e l'altro dal secondo.

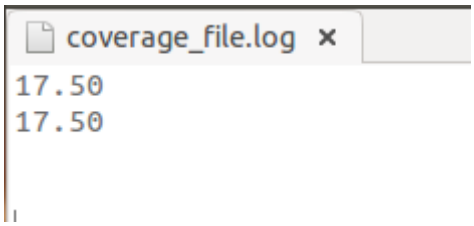


Illustrazione 8.10

Le stime presenti nei file ora sono due per ciascuno. Infatti ogni stima viene aggiunta a quelle precedenti. Nel caso di questo semplice test, le precedenti misure del primo esempio erano state cancellate. Infatti le misure sono due invece di tre.

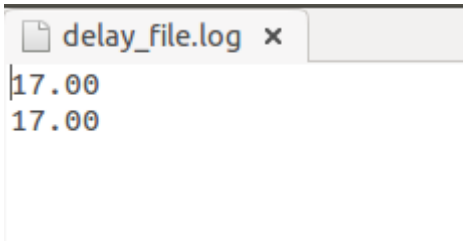


Illustrazione 8.11

Oltre ai dati riportati in queste illustrazioni vi sono, per ciascun processo, le stampe dei dati inseriti nella matrice, dei valori del vettore come nell'esempio precedente. In questo caso però saranno il doppio essendo due i processi lanciati.

CONCLUSIONI

Ciò che è stato fatto in questa tesi è scrivere un algoritmo per un determinato applicativo e regolare l'avvicendarsi di processi paralleli (eseguibili del codice contenente l'algoritmo citato sopra).
Dell'algoritmo proposto è presente il costo in termini di grandezza dell'input. Tale algoritmo è stato scritto in linguaggio C. La successiva parte relativa alla gestione dei processi paralleli stabilisce un numero massimo di processi da lanciare in parallelo. Tale numero massimo è però relativo alla prima parte dell'algoritmo, la quale accede frequentemente alla memoria secondaria in cui risiedono i file di traccia che devono essere elaborati. Una volta finita tale fase, la seconda non presenta limiti relativi al numero di processi in esecuzione parallela.
Quindi vengono ora raccolte osservazioni in merito ad un ulteriore miglioramento del codice scritto.
Ci sono tre parti di codice scritto contenute in tre differenti file. La prima è quella contenuta nei file sim-metrics-*-corpus. Tale parte di codice è stata affrontata nel capitolo otto, per quanto già scritto permette di scegliere il numero di processi da lanciare in parallelo in un determinato momento. Tale numero rimane poi fisso riguardo all'esecuzione delle varie parti dell'applicativo. La memoria libera però, che è il parametro attraverso il quale si stima il numero di processi da lanciare, è una variabile non una costante. La disponibilità di memoria infatti varia continuamente in relazione a ciò che viene eseguito sul terminale. Per rendere il tutto più dinamico si potrebbe quindi procedere ad inserire il calcolo del numero di istanze da lanciare in parallelo durante l'esecuzione dello script get_coverage_script. In questo modo, la scelta di tale numero dipenderà dalle condizioni subito precedenti l'eventuale ingresso di un nuovo processo all'interno dell'esecuzione. Non vi è conflitto fra le varie istanze. Infatti per evitare conflitti è fondamentale che non vi siano un numero massimo di K istanze mandate in esecuzione. Ognuno dei processi, a seconda del suo limite massimo, cercherà di creare una

cartella con un determinato indice. Tale cartella si troverà però nella path in comune con altri processi e quindi vi sarà sempre una “visuale ” abbastanza ampia da sapere quali cartelle possano essere create. Tale metodo tende ad avere un numero di processi paralleli in esecuzione maggiore del numero stabilito. Si pensi a cosa accadrebbe nel caso in cui un processo venisse lanciato dopo una lunga serie di processi. Il penultimo processo aveva stimato uno spazio in memoria per cinque esecuzioni parallele. Vi sono quattro processi in esecuzione quindi viene lanciato anch'esso. Il processo successivo ha una stima di K pari a due. Supponendo che l'indice numero uno si sia liberato, in quanto il primo processo ha terminato l'esecuzione, il nuovo processo prenderà quell'indice per poter eseguire in parallelo. In questo caso nonostante il massimo dall'ultimo processo sia due, il numero di processi in esecuzione sarà cinque. Questo è vero se si mantiene la struttura identica a quella creata nel dile get_coverage_script e si aggiunge la parte di codice aggiunta in sim-metrics-*-corpus nello script get_coverage_script.
Un'ulteriore modifica può essere fatta al file get_coverage_and_delay.c.
In questo caso la modifica proposta è l'aggiunta di un ordinamento del vettore dei messaggi con conseguente velocizzazione della ricerca dell'indice corretto all'interno della matrice, per quanto scritto nel capitolo sul calcolo dei costi.

BIBLIOGRAFIA:

[1]http://arxiv.org/pdf/1105.2447v2.pdf.
[2]Fondamenti di informatica per la progettazione multimediale. Dai linguaggi formali all'inclusione digitale, Marco Padula, Amanda Reggiori. Pagina 54.
[3]Fondamenti di informatica per la progettazione multimediale.Dai linguaggi formali all'inclusione digitale, Marco Padula, Amanda Reggiori. Pagina 53.
[4]http://gdangelo.web.cs.unibo.it/pool/papers/gdangelo-DISIO-2010-NONUFFICIALE.pdf
[5]C pocket. Telecomunicazioni, didattica, software aerospaziale e processi industriali: con C questo e molto altro, Enrico Amedeo, Apogeo Editore ,2007, pagina 44 capitolo 3.
[6]C pocket. Telecomunicazioni, didattica, software aerospaziale e processi industriali: con C questo e molto altro, Enrico Amedeo, Apogeo Editore, 2007, capitolo 3, pagina 37.
[7]Dizionario di informatica: inglese-italiano (Google eBook), Angelo Gallippi, Tecniche Nuove 2006, pag 533.
[8]Eucip. Il manuale per l'informatico professionista. Certificazione Core Level, Antonio Teti, Egidio Cipriano, HOEPLI Editore.
[9]PHP 5 - Guida completa, Andi Gutmans, Stig Bakken, Derick Rethans, pagina 23.
[10]Introduzione al Calcolo Scientifico, Alfio Quarteroni, Fausto Saleri, Springer 2007, pagina 25.
[11]Introduction to algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press,capitolo 8 , pag 191.
[12]C. Corso completo di programmazione, Harvey M. Deitel, Paul J. Deitel, Apogeo Editore 2007, capitolo 6, pag 218.
[13]Programmazione ad oggetti e tipi di dati astratti con il C++,Franco Crivellari, Franco Angeli, 1997, pagina 289.
[14]Algoritmi e strutture dati in Java, Adam Drozdek, Apogeo Editore 2001, capitolo 6, pag 238.

[15]Objective-C & Cocoa: Il manuale di programmazione per Max OS X, By Angelo Iacubino, HOEPLI EDITORE
Primo capitolo, paragrafo “I parametri standard della funzione *main*”.
[16]Introduction To Algorithms, Cormen, MIT Press 2001, pagina 221, capitolo 11.
[17]Linux. La guida. By Matt Welsh, Matthias Kalle Dalheimer, Lar Kaufman, Apogeo Editore 2000, capitolo 1, pagina 15.
[18] Problem solving e programmazione in C, Jeri R. Hanly, Elliott B. Koffman, Apogeo Editore, capitolo 1, pagina 9.
[19]Learning the bash Shell: Unix Shell Programming, By Cameron Newham, O'Reilly Media, Inc., Feb 9, 2009, capitolo 8, pagina 197.
[20]Introduction To Algorithms, Cormen, MIT Press 2001, pagina 221, capitolo 11.
[21]Operating Systems, Sibsankar Haldar, Alex Alagarsamy Aravind, pagina 76.
[22]Guida a Unix con Linux, Jack Dent, Tony Gaddis, Apogeo Editore 2001, capitolo10, pagina 378.
[23]The C Programmer's Companion: ANSI C Library Functions, R.S Jones, Silicon Press 1991, pagina 130.
[24]Problem solving e programmazione in C, By Jeri R. Hanly, Elliott B. Koffman, Apogeo Editore, capitolo6, pagina 278.
[25]C for U Including C and C Graphics,Veerana V K, Jankidevi S J, Firewall Media 2007, pagina 335.
[26]Beginning C, 5th Edition, Ivor Horton, Apress, Feb 27, 2013, pagina 59.
[27] Gocce di Java. Un'introduzione alla programmazione procedurale ed orientata agli oggetti, Pierluigi Crescenzi, pagina 25.
[28]C pocket. Telecomunicazioni, didattica, software aerospaziale e processi industriali: con C questo e molto altro, Enrico Amedeo, pagina 123.
[29]C. Corso completo di programmazione, Harvey M. Deitel, Paul J. Deitel, Apogeo Editore 2007, pagina 185.
[30]Guida a Unix con Linux ,By Jack Dent, Tony Gaddis, capitolo

2,pagina 40.
[31]Informatica.Concetti e sperimentazioni. By M. Rita Laganà, Marco Righi, Francesco Romani, Glossario, Albero.
[32]Gocce di Java. Un'introduzione alla programmazione procedurale ed orientata agli oggetti, Pierluigi Crescenzi,2011, pagina 24.
[33]ECDL Syllabus 4.0. Sabrina Bertolacci, Francesco Grossi, Apogeo Editore, 2004,1.2.2.1 Concetti di base della Tecnologia dell'informazione.
[34]Guida a Unix con Linux, Jack Dent, Tony Gaddis, Apogeo Editore,2001, pagine 35, 36.
[35]Sistemi operativi, Harvey M. Deitel, David R. Choffnes, Paul J. Deitel, pagina 327.
[36]Sistemi operativi. Concetti ed esempi, Abraham Silberschratz, Peter Baer Galvin, Greg Gagne, pagina 333.
[37]www.linuxmanpages.com.