Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

STRUMENTI PER L'ANALISI DEL TEMPO DI ESECUZIONE

Tesi di Laurea in Linguaggi di programmazione

Relatore:
Chiar.mo Prof.
Simone Martini

Presentata da:
Sara Bergonzoni

Sessione III
Anno Accademico 2010-2011

Indice

1 Introduzione

Il presente elaborato si pone lo scopo di analizzare alcuni strumenti automatici o semi-automatici per l'analisi del tempo di esecuzione.
Dopo una breve panoramica sui concetti alla base dell'analisi del tempo di esecuzione si procederà allo studio dettagliato di un esempio concreto quale è COSTA (*COSt and Termination Analyzer for Java Bytecode* [2]) un analizzatore di costi e terminazione, della quale si studierà in dettaglio il funzionamento delle sue componenti, quali:

- Acquisizione del codice Java Bytecode ed esecuzione del controllo di flusso;
- Salvataggio del contenuto delle stack all'interno di alcune variabili locali;
- Deduzione delle relazioni esistenti tra la dimensione dell'input e il numero di chiamate alle regole;
- Creazione degli alberi che descrivono i possibili sviluppi del codice Java Bytecode analizzato;
- Calcolo del numero di nodi massimo per questo albero;
- Stima dei costi di un nodo;
- Calcolo dell'upper bound delle *espressione di costo* che verranno restituite dal programma;

1.1 Definizione: analisi di costi

L'analisi dei costi si pone di quantificare la complessità computazionale di un algoritmo. La complessità computazionale di un algoritmo definisce con una espressione la quantità di risorse (in tempo o in spazio) utilizzate da un calcolatore per eseguire l'algoritmo stesso. Questa espressione è generalmente definita nei termini della dimensione dell'input.

1.2 Definizione e scopo dell'analisi statica

L'analisi statica del codice prevede l'analisi del software compiuta senza effettivamente eseguire il codice sorgente. Questo concetto generalmente definisce un insieme di controlli di consistenza effettuati al momento della compilazione. Tali controlli hanno lo scopo di verificare che un programma possa essere compilato con successo, ed individuano, prima dell'esecuzione, alcuni errori di programmazione effettuando un controllo sintattico ed un controllo di tipo. L'analisi statica non comprende unicamente semplici considerazioni sulle dichiarazioni, ma può arrivare a compiere un'analisi più approfondita del codice sergente. Le informazioni ottenute vengono utilizzate all'interno di metodi formali che dimostrino matematicamente le proprietà di un dato programma.

1.3 Tecniche di analisi dei costi

Un tipico approccio all'analisi automatica dei costi prevede due fasi. Nella prima fase, dato un programma e un'unità di misura di costo, si produce un insieme di equazioni che cattura il costo del programma in relazione alla dimensione dei dati in ingresso. Tali equazioni vengono generate convertendo i costrutti di iterazione del programma (loop e ricorsione) in equazioni di ricorrenza e deducendo le *relazioni di dimensione*, che approssimano la dimensione in relazione agli argomenti. Questo insieme di equazioni prende il nome di *rappresentazione ricorsiva* (RR in breve).
Lo scopo della seconda fase è di ottenere una rappresentazione non-ricorsiva delle equazioni, note come equazioni in *forma chiusa*. Nella maggior parte dei casi, non è possibile trovare una soluzione esatta e le equazioni in *forma chiusa* corrispondono ad un limite superiore.
Nella maggior parte degli analizzatori di costi disponibili, la prima fase è studiata in dettaglio, mentre la seconda fase ha ricevuto meno attenzione. Fondamentalmente, ci sono tre diversi approcci per la seconda fase. Un approccio, si basa su trasformazioni source-to-source che convertono i programmi ricorsivi in altri non ricorsivi. Il secondo approccio consiste nel costruire risolutori di ricorrenza limitati senza l'utilizzo di tecniche matematiche standard. Il terzo approccio consiste nel basarsi su sistemi esistenti

di *computer algebra systems* (CASs in breve[5])(di questa tecnica farà uso anche COSTA).[4]

Analizzatori di costo che si basano sull'approccio precedentemente descritto sono disponibili per una vasta gamma di linguaggi di programmazione, sia funzionali (esempi:ACE: an automatic complexity evaluator[8],Mechanical program analysis [13], Automatic complexity analysis [10], Strictness analysis aids time analysis [12], [11],Automated complexity analysis (ACA)[6]), che logici (esempi:Cost analysis of logic programs[7], Cost analysis of java bytecode[9]) ed imperativi (esempi:[1],Cost analysis of java bytecode[3]).

1.4 Differenza tra relazioni di costo e relazioni di ricorrenza

Le *relazione di costo* (CR) che verranno utilizzate all'interno di COSTA differiscono dalle normali relazioni di ricorrenza (RR). Le CR al contrario delle RR:

- Possono fare uso di non determinismo: quindi equazioni per la stessa relazione non devono essere necessariamente mutuamente esclusive. Anche se il linguaggio di programmazione è deterministico, l'astrazione della dimensione determina una perdita di precisione: alcune condizioni potrebbero non essere osservate quando si usa la dimensione degli argomenti invece dei valori reali.

- Possono contenere rapporti tra dimensioni e non unicamente uguaglianze. Quando si trattano linguaggi di programmazione realistici che contengono strutture dati non lineari, come gli alberi, spesso l'analisi delle dimensioni non produce risultati esatti. Ad esempio, l'analisi può dedurre che la dimensione di un struttura dati diminuisce tra un iterazione ed un altra, ma non è in grado di stabilire precisamente di quanto si riduce.

- Dipendono da molti argomenti, che possono crescere o decrescere ad ogni iterazione. Infatti, il numero di volte che viene eseguita una relazione può essere il risultato, non di un solo argomento ma di una combinazione di molti di questi.

[4]

1.5 Breve introduzione alla versione ridotta del linguaggio Java Bytecode utilizzata all'interno di COSTA

Il Java Bytecode è un linguaggio a basso livello per la programmazione object-oriented con controlli di flusso non strutturato e uno stack degli operandi che conserva i risultati intermedi della computazione. Inoltre, gli oggetti sono salvati nella memoria dinamica (lo heap).

Un programma Java Bytecode consiste in un set di *class file*, uno per ogni classe o interfaccia. Un *class file* contiene informazioni circa il suo *name* $c \in Class\_Name$, la classe che estende, l'interfaccia che la implementa, ed i campi e i metodi da essa definiti. In particolare, per ogni metodo, il *class file* contiene: la signature del metodo $m \in Meth\_Sig$ e il suo codice bytecode bc_m. La signature consiste nel nome $name(m) \in Meth\_Name$ e nei tipi che prende in input $type(m) = \tau_1, \ldots, \tau_n \to \tau \in Meth\_Type$ dove $\tau_i \in Type$. Il bytecode è indicato come $bc_m = \langle pc_0 : b0, \ldots, pc_{n_m} : b_{n_m} \rangle$, dove ogni b_i è un istruzione bytecode e pc_i è il suo indirizzo.

All'interno del progetto COSTA verrà considerato un sottoinsieme del linguaggio JVM, esso è in grado di gestire le operazioni su interi, la creazione e la manipolazione di oggetti, la chiamata ai metodi e la gestione delle eccezioni (sia generate da esecuzione anomala sia esplicitamente lanciata dal programma).

L'insieme di di istruzioni bytecode ($bcInst$) è: bcInst ::= push x | istore v | astore v | iload v | aload v | iconst a | iadd | isub | imul | idiv | if$\diamond$ pc | goto pc | new $Class\_Name$ | invokevirtual $Class\_Name.Meth\_Sig$ | invokespecial $Class\_Name.Meth\_Sig$ | athrow | ireturn | getfield $Class\_Name.Field\_Sig$ | putfield $Class\_Name.Field\_Sig$ dove $\diamond$ è un operatore di comparazione, v è una variabile locale, a è un intero, pc è l'indirizzo di una istruzione, ed x è un intero o il valore speciale `NULL`.

2 COSTA

2.1 Introduzione

COSTA acronimo di *COSt and Termination Analyzer for Java Bytecode* permette lo studio dei costi e della terminazione dei programmi scritti in Java Bytecode, attraverso un interpretazione astratta basata sull'analisi statica del codice. L'analisi dei costi fornita da COSTA stima staticamente l'ammontare delle risorse che potranno essere consumare a runtime, mentre l'analizzatore di terminazione si accerta, se possibile, della terminazione del programma su ogni input.
Esso è nato con l'obbiettivo di unire l'analizzatore di costi e analizzatore di terminazione nella stessa applicazione dal momento che condividono gran parte dei meccanismi di calcolo. Il progetto COSTA è basato su un approccio classico all'analisi statica dei costi e prevede due fasi. La prima fase prende in input un programma e il *modello di costo* e produce la *relazione di costo*, formata da un insieme di relazioni ricorsive. La seconda fase cerca una soluzione al problema della terminazione, appoggiandosi al progetto PUBS come verrà spiegato nel paragrafo 3.2.
Il *modello di costo* indica il tipo di informazioni che si vogliono ricavare dall'analisi del codice. Le notazioni di costo possibili sono: occupazione dello heap, numero di istruzioni bytecode eseguite e numero di chiamate ad uno o più metodi effettuate.
COSTA può analizzare sia codice Java derivante della Standard Edition sia della Micro Edition, ed è in grado di lavorare sia partendo da linguaggio Java che partendo da linguaggio Java Bytecode, che non è necessario derivi dalla compilazione di codice Java. COSTA può produrre risultati significativi e ragionevolmente precisi anche per programmi non banali, possibilmente utilizzando le librerie di Java [2].
Allo scopo di favorire l'usabilità di questa applicazione il sistema fornisce diverse interfacce utente:

- Una interfaccia a riga di comando (utilizzata durante lo sviluppo e ora non più disponibile).
- Un'interfaccia web.

- Un plugin per l'ambiente eclipse.

2.1.1 Utilizzo

I costi e i risultati di terminazione ottenuti dall'impiego di questo analizzatore permettono di:

- Aiutare il programmatore nello sviluppo dei processi, grazie anche al plugin per l'ambiente eclipse il quale permette di utilizzare l'analizzatore mentre si sviluppa il software.
- Fornire garanzie al cliente, che può così avere la certezza che il programma non consumerà più risorse di quanto dichiarato dall'analisi. In seguito questo porterà essere combinato con il paradigma Proof-carrying code.
- Ottimizzare i programmi, guidando la scelta sull'implementazione più efficiente tra varie alternative.

2.2 Interfacce utente

2.2.1 Interfaccia web

L'interfaccia web permette l'utilizzo di COSTA da locazione remota, senza installarlo localmente. Agli utenti è permessa l'analisi di una serie rappresentativa di esempi oltre alla possibilità di caricare codice sorgente Java o bytecode (estensioni .class e .jar). Il comportamento di COSTA può essere personalizzato utilizzando una serie di opzioni che permettono la selezione del modello di costo e del metodo da analizzare. Vi sono due modalità alternative di utilizzo dell'analizzatore: una automatica e una manuale. La versione automatica permette di selezionare solo il livello di precisione dell'analisi che varia da -3 (il più basso) a 4 (il più elevato), il vantaggio è che non richiede all'utente la conoscenza delle diverse opzioni implementate dal sistema e la loro implicazione nell'accuratezza dell'analisi. Il metodo manuale permette all'utente di accedere a tutte le opzioni, consentendo un controllo accurato dell'analizzatore. Alcune di queste opzioni consentono di decidere se analizzare le librerie standard di Java o no, se prendere in considerazione

le eccezioni, se scrivere o leggere i risultati dell'analisi da un file al fine di riutilizzarli successivamente, ecc.
Oltre a mostrare il risultato della terminazione e l'upper bound del costo di esecuzione, Costa mostra informazioni riguardanti il tempo richiesto dai passi intermedi eseguiti dall'analizzatore nelle fasi precedenti.[2]

2.2.2 Plugin Eclipse

Il plugin di Eclipse costituisce un'interfaccia per COSTA perfettamente integrata con l'ambiente di sviluppo. Questo plugin permette ai programmatori di analizzare metodi durante il processo di sviluppo. Esso carica il class path stabilito per il progetto e utilizza per l'analisi le stesse classi e librerie specificate dall'utente per la compilazione e l'esecuzione del programma. Come per l'interfaccia web gli utenti possono configurare un vasto set di opzioni [...] che sono salvate e caricate ad ogni esecuzione di Eclipse. Inoltre l'utente può scegliere tra l'analisi automatica e la modalità avanzata [...]che permette una personalizzazione molto più raffinata, come accade per l'interfaccia web. Con questo plugin si possono analizzare uno o più metodi di una classe o tutta la classe. I risultati dell'analisi sono riportati utilizzando marcatori nel codice sorgente. Tali marcatori sono diversi a seconda del modello dei costi utilizzato per l'analisi. Inoltre, il plugin mostra anche tutti i precedenti risultati dell'analisi in un ulteriore visita, chiamata the COSTA view. The COSTA view include anche un'icona di avviso per i metodi la cui terminazione non è dimostrata. [2]

3 Architettura di COSTA

Il progetto COSTA procede con l'analisi dei costi seguendo una serie di passaggi che comprendono:

1. Ricezione in input del codice bytecode da analizzare;
2. Esecuzione del controllo di flusso allo scopo di eliminare costrutti per il controllo di flusso non strutturato e invocazioni a metodi virtuali; questa operazione viene effettuata attraverso la costruzione di *control flow graph*(CFG);
3. rappresentazione del CFG come set di regole utilizzabili in una rappresentazione ricorsiva intermedia che salva il contenuto dello stack in una serie di variabili locali;
4. Deduzione delle relazioni esistenti tra dimensione delle variabili in ingresso e chiamate alle regole;
5. Approssimazione del numero delle variabili rilevanti per ogni regola;
6. Restituzione delle *relazioni di costo*;
7. Creazione dell'*albero di evoluzione* partendo dai sistemi di *relazioni di costo*;
8. Calcolo degli upper bound del numero di nodi dell'*albero di evoluzione* ;
9. Stima del costo del singolo nodo;
10. Calcolo dell'upper bound delle *espressioni di costo*;

I primi sei passi vengono eseguiti dal framework generico per l'analisi dei costi, come è spiegato nel paragrafo 3.1. I restanti passaggi sono compiuti come spiegato nel progetto PUBS 3.2, che è stato integrato nel progetto COSTA allo scopo di effettuare l'analisi della terminazione. É necessario notare che queste spiegazioni non si riferiscono direttamente a COSTA ma a documentazione a cui il progetto dice di ispirarsi. Tutto il processo verrà trattato riferendosi all'analisi del numero di istruzioni eseguite anche se gran parte delle operazioni sono comuni ad altri tipi di analisi.

3.1 Framework generico per l'analisi automatica dei costi nei programmi Java Bytecode

Oltre alle difficoltà, che normalmente si riscontrano nell'analisi dei costi, l'elaborazione di codice Java Bytecode e più in generale di codice di basso livello introduce ulteriori difficoltà, quali:

1. controlli di flusso non strutturati, esempio si fa uso di costrutti come il `goto` al posto di chiamate ricorsive.

2. invocazione di metodi virtuali, caratteristica tipica dei linguaggi object oriented, che possono modificarne il costo.

3. utilizzo di celle dello stack per la memorizzazione di valori intermedi dell'elaborazione.

3.1.1 Processo per la creazione delle *relazione di costo*

Il framework prende in input il bytecode corrispondente ad un metodo e produce le *relazioni di costo*, dopo aver eseguito i seguenti passaggi (alcuni dei quali permettono di risolvere le difficoltà evidenziate in precedenza):

1. Il codice bytecode in ingresso è trasformato in un *control flow graph*(CFG), che permette di: (1) risolvere la prima complicazione eseguendo un controllo di flusso non strutturato sul bytecode, (2) risolvere la seconda complicazione dal momento che le invocazioni di metodi virtuali e le eccezioni vengono considerate come nodi del grafo.

2. Il CFG è rappresentato con un set di regole, che vengono utilizzare all'interno di una *rappresentazione ricorsiva intermedia* con lo scopo di appiattire lo stack locale convertendolo in una serie di variabili locali, questo permette di risolvere la terza complicazione.

3. Vengono dedotte *relazioni di costo* esistenti tra le dimensioni delle variabili di input e le chiamate alle regole per mezzo dell'analisi statica. Le *relazioni di costo* sono rappresentate da vincoli sulla dimensione delle variabili (per gli interi) o dal valore del percorso più lungo raggiungibile (per i riferimenti).

4. Ogni regola della *rappresentazione ricorsiva* viene approssimata dall'insieme degli argomenti che sono rilevanti per il calcolo dei costi.

5. Partendo dalla *rappresentazione ricorsiva*, dagli argomenti rilevanti (trovati nel punto 4) e dalla *relazione di dimensione* viene prodotta la *relazione di costo* del metodo. La *relazione di costo* è rappresentata da una funzione degli argomenti dell'input.[3]

3.1.2 Generazione dei *control flow graph*

La generazione del CFG garantisce, che i controlli di flusso non strutturati presenti nel codice bytecode di un metodo vangano trasformati in ricorsione. All'interno del grafo del metodo ogni singolo nodo è identificato da una tupla $\langle id, G, B, D\rangle$ dove id è un identificatore del nodo, G ($guard$) indica le condizioni sotto cui il nodo è eseguito, B rappresenta una sequenza di istruzioni che vengono eseguite nel caso la condizione in G sia vera e D rappresenta la lista di adiacenza del nodo corrente. Le condizioni in G devono essere mutuamente esclusive in modo che solo un blocco possa essere eseguito. In seguito alla generazione dei CFG gran parte delle istruzioni Java Bytecode possiedono un solo successore, ma esistono tre tipi di istruzioni, che creano diramazioni:

Salti condizionali: hanno la forma "pc_i : if $\diamond$ pc_j." Dipendono dal valore di verità della condizione, l'esecuzione può saltare a pc_j oppure continuare la normale esecuzione eseguendo pc_{j+1}. Nel grafo questo nodo viene rappresentato facendo partire dal nodo corrente due archi, uno che porta al nodo destinazione del salto e l'altro che punta al nodo successivo del blocco corrente. Ognuno dei nodi destinazione inizierà con una condizione di base sotto la quale il blocco viene eseguito.

Dynamic dispatch : ha la forma "pc_i : invokevirtual $c.m$". Il tipo dell' oggetto o il cui metodo viene invocato non è noto staticamente (potrà appartenere a c o ad ogni sottoclasse di c), quindi non è possibile conoscere staticamente quale metodo verrà invocato. Verranno quindi esplicitate nel grafo tutte le possibili scelte effettuabili. Per trovare tutte le possibili scelte viene usata la funzione `resolve_virtual`(c, m), la qua-

le ritorna l'insieme $ResolvedMethods$ formato da coppie $\langle d, \{c_1, \ldots, c_k\}\rangle$, dove d è una classe che definisce una signature per il metodo m e ogni c_i è c oppure una sottoclasse di c che eredita il metodo specificato da d. Per ogni $\langle d, \{c_1, \ldots, c_k\}\rangle \in ResolvedMethods$, un nuovo blocco $Block_d^{pc_i}$ è generato con un'unica istruzione `invoke`$(d : m)$ che indica una invocazione non virtuale del metodo m che è definito nella classe d. In aggiunta, il blocco ha una guard della forma `instanceof`$(o, \{c_1, \ldots, c_k\})$ (o è un elemento dello stack) per indicare che il blocco è eseguibile solo quando o è un istanza di una delle classi $c_1, \ldots, c_k$. Un arco dal blocco contenente pc_i verso il blocco $Block_d^{pc_i}$ è aggiunto, insieme ad un arco che dal $Blockd_d^{pc_i}$ punta verso il blocco contenente la prossima istruzione pc_{i+1} (la quale descrive il resto dell' esecuzione dopo aver invocato m). I campi sono trattati in modo simile.

Eccezioni : Nel grafo CFG le eccezioni non vengono trattare in modo particolare. La possibilità che si sollevi un eccezione mentre si esegue uno statment bytecode b è semplicemente trattato come una diramazione aggiuntiva dopo b. Se il $Block_b$ termina con b; gli archi uscenti dal $Block_b$ sono quelli originati dal normale control flow, insieme a quelli che permettono di raggiungere i sotto grafi che corrispondono ai gestori delle eccezioni. Nel operazioni successive le eccezioni per semplicità vengono ignorate, dal momento che considerandole si introducono solamente più ramificazioni nel CFG e richiedono argomenti aggiuntivi nella rappresentazione ricorsiva.[3]

3.1.3 Rappresentazione ricorsiva con appiattimento dello stack

La rappresentazione ricorsiva con appiattimento dello stack consiste nella trasformazione di un metodo da iterativo a ricorsivo, e nella rappresentazione del contenuto dello stack come una serie di variabili locali. L'appiattimento dello stack è possibile dal momento che l'altezza massima dello stack è sempre decidibile staticamente.
Dato un metodo m definito in una classe c, le variabili locali sono indicate con $\overline{l_k} = l_0, ..., l_k$ dove l_0 contiene un riferimento all'oggetto stesso,

$l_1, ..., l_n$ rappresentano gli n argomenti di input del metodo, mentre le variabili $l_{n+1}, ..., l_k$ corrispondono alle $k - n$ variabili locali dichiarate in m. Le variabili $\overline{s_t} = s_0, ..., s_{t-1}$ vengono usate per indicare gli elementi dello stack (s_0 indica la posizione più bassa mentre s_{t-1} quella più alta).
La rappresentazione ricorsiva di un metodo è definita da un insieme di regole `head`$\leftarrow$ `body` ottenute, dal grafico CFG del metodo, nel modo seguente:

- La regola del method entry è `c:m(`l_n`,ret)` $\leftarrow$ `c:`m_0 `(`l_k`, ret)`. Questo significa, che alle variabili di ingresso del metodo vengono aggiunte le variabili dichiarate all'interno del metodo e la variabile in cui viene salvato il valore di ritorno, ovvero `ret`.

- Per ogni nodo $\langle id, G, B_p, id_1, \ldots, id_j \rangle \in G_m$, c'è la regola
  `c:m`^{id}`(`$\overline{l_k}$`,` $\overline{s_t}|_{h_{id}}$`, ret)` $\leftarrow$ `G',` $\overline{B'}_p$ `(call`$_{id_1}$ `;...; call`$_{id_j}$`)`
  dove {`G'`}$\cup\overline{B'}_p$ è ottenuta da {`G`}$\cup\overline{B}_p$, e `call`$_{id_1}$`; ...; call`$_{id_j}$ sono le possibili chiamate ai nodi (; indica la disgiunzione). Il metodo definito nel blocco rappresentato come parametri, le variabili locali, gli argomenti del metodo e la cella in cui salvare il valore di ritorno, viene rimpiazzato da una *gard* e dal puntatori ai blocchi successivi a cui sono state già aggiunte le variabili locali e le variabili di stack

Ogni $b_i \in$ {`G'`} $\cup\overline{B'}_p$ è traslato in b_1' dove vengono esplicitamente aggiunte le variabili (sia locali, sia dello stack) che b_i usa come argomento [3].

3.1.4 Relazione di dimensione

Le *relazione di dimensione* esistenti tra gli stati delle diverse parti del programma sono indispensabili per la creazione della *relazione di costo* , dal momento, che a tale scopo è importante stabilire il costo di un nodo in relazione al successivo. Le *relazione di dimensione* vengono determinate in due modi a seconda del tipo di dato: le variabili intere sono vincolate dai possibili valori della variabile, mentre nei riferimenti a variabili il vincolo è costituito dalla massima lunghezza del percorso raggiungibile.
Al fine di creare le *relazioni di costo,* è necessario che per ogni regola della *rappresentazione ricorsiva* avvenga una chiamata `calls-to-size-relations` tra le variabili nella head della regola e le variabili usate nelle chiamate

(alle regole) che si verificano nel body. Data una regola $\mathtt{p}(\overline{x})\leftarrow\mathtt{G}$, $\overline{B}_k$, ($\mathtt{q}_1$; ...; $\mathtt{q}_n$), dove ogni $b_1 \in \overline{B}_k$ è un istruzione bytecode o una chiamate ad un altra regola (che deriva dalla traduzione di un altro metodo invocato). $\mathtt{calls}(\overline{B}_k)$ indica l'insieme di tutti i b_i che corrispondono al metodo chiamato, e $\mathtt{bytecode}(\overline{B}_k)$ denota l'insieme di tutti i b_i corrispondente ad altro codice bytecode.

Definizione 1 (Calls-to size-relations) *Se R_m è la rappresentazione ricorsiva del metodo m, dove ogni regola prende la forma $p(\overline{x}) \leftarrow G$, $\overline{B}_k$ ($q_1(\overline{y})$; ...; $q_j(\overline{y})$). La calls-to size-relations di R_m è una tripla della forma*

$$\langle p(\overline{x}), p'(\overline{z}), \varphi\rangle \text{ dove } p'(\overline{z}) \in calls(\overline{B}_k) \cup \{p\_cont(\overline{y})\}$$

descrivendo, per tutte le regole, la relazione di dimensione tra $\overline{x}$ e $\overline{z}$ quando viene effettuata la chiamata $p'(\overline{z})$ dove $\{p_{cont}(\overline{y})\}$ si riferisce al program pointer subito dopo $\overline{B}_k$. La relazione di dimensione φ è data come una congiunzione di vincoli lineari tra $a_0 + a_1v_1 + ... + a_nv_n$ op 0, dove op $\in \{=, \leqq <\}$ per ogni a_i è una costante e $v_k \in \overline{x} \cup \overline{z}$ per ogni k.

[3]

3.1.5 Deduzioni sulle relazioni di dimensione

L'analisi sulla *relazioni di dimensione* è effettuata un modo semplice ma efficiente compilando il bytecode, imponendo dei vincoli lineari sulle variabili e calcolando un bottom-up fixpoint sulle regole di compilazione usando un algoritmo fixpoint bottom-up standard.
La compilazione dei vincoli lineari è fatta con un funzione astratta α_{size} che fondamentalmente rimpiazza le variabili nella guardia e nel bytecode con i vincoli imposti sulle variabili corrispondenti. In generale, in ogni esecuzione bytecode (lineare) le operazioni aritmetiche sono rimpiazzate dalla loro costante lineare corrispondente, e ogni istruzione bytecode che manipola oggetti è compilata con la variabile che indica la costante lineare del più lungo cammino raggiungibile dall'oggetto.

3.1.6 Costo Computazionale di un metodo

Il costo computazione di un metodo è generato dalla *rappresentazione ricorsiva* di un metodo e dall'utilizzo delle informazioni dedotte dal analisi delle dimensioni. Al fine di ottenere una *relazione di costo* ottimale è necessario cercare gli argomenti della *relazione di costo* che possono essere ignorati.
Considerando un nodo nel CFG rappresentato dalla formula $\texttt{c:m}^{id}(\overline{l_k}\texttt{, ret}) \leftarrow \texttt{G, } \overline{B}_h \texttt{ (call}_{id_1}\texttt{; ...; call}_{id_j}\texttt{)}$ nella quale le variabili locali e lo stack non sono più distinguibili. La *funzione di costo* per il nodo ha la forma $C_{id} : (\mathbb{Z})^n \rightarrow \mathbb{N}_\infty$ con $n \leq k$ numero degli degli argomenti, $\mathbb{Z}$ insieme di interi e $\mathbb{N}_\infty$ insieme dei numeri naturali aumentati dal simbolo speciale ∞, che denota l'infinito.
Il numero n degli argomenti da prendere in considerazione nella *funzione di costo* viene limitato compiendo alcune osservazioni, ad esempio la variabile di output `ret` non può influenzare il costo del nodo e viene quindi ignorata nella *funzione di costo* , inoltre è possibile ignorare alcuni argomenti di input come, i parametri di accumulazione, che non influenzano ne il controllo di flusso ne il costo del programma.
In generale, data una regola, gli argomenti che non hanno un impatto sul costo del programma sono quelli che non hanno effetti diretti o indiretti sulla guardia del programma, o sono usati come argomenti di input su altri metodi esterni il cui costo, a sua volta può dipendere dalla dimensione dell'input. Calcolare una sicura approssimazione nell'insieme delle variabili che interessano è un problema molto studiato in analisi statica. Per fare questo, è necessario seguire le dipendenze dei dati nei confronti del controllo di flusso, questo comporta il calcolo del punto fisso. Il nostro problema risulta più semplice rispetto al calcolo del punto fisso dal momento che non dobbiamo togliere le variabili ridondanti dal programma, dobbiamo solo individuare gli argomenti rilevanti. Formalmente, data una regola $\texttt{p}(\overline{x}) \leftarrow \mathit{body}$, $\hat{\texttt{l}}_p \subseteq \overline{x}$ è la sotto-sequenza delle variabili rilevanti in p. La sequenza $\hat{\texttt{l}}_P$, è ottenuta dall'unione delle sequenze $\{\hat{\texttt{l}}\}_{p \in P}$ con un set P di regole, mantengono l'ordinamento delle variabili.

3.1.7 Relazioni di costo

La *funzione di costo* $C_{id} : (\mathbb{Z})^n \to \mathbb{N}_\infty$ per un $Block_{id}$ viene definita per mezzo di una *relazioni di costo*, le quali consistono in un insieme di equazioni di costo. Queste equazioni di costo saranno utilizzate per ragionare sul costo computazionale che comporta l'esecuzione del $Block_{id}$. Intuitivamente, data una regola $\mathtt{p}(\overline{x})\leftarrow \mathtt{G},\ B,\ (\mathtt{q}_1;\ \ldots;\ \mathtt{q}_j)$ associata al $Block_{id}$, vengono generate:

- una equazione di costo, che definisce il costo di p come il costo delle dichiarazioni in B, più il costo della sua continuazione denotata da `p_count`.
- un'altra equazione di costo definisce `p_count` o come il costo di q_1 se la guardia è soddisfatta, ..., o come il costo di q_n se la guardia è soddisfatta.

Il costo della continuazione è specificato in un'equazione separata perchè le condizioni per determinare quale dei cammini alternativi q_i verrà eseguito sarà noto solo alla fine dell'esecuzione di B. Nella definizione seguente noi usiamo la definizione α_{guard} per rimpiazzare quelle guardie che indicano il tipo di oggetto corretto.

Definizione 2 (Cost relation) *Se R_m è la rappresentazione ricorsiva del metodo m, dove ogni regola prende la forma $\mathtt{p}(\overline{x}) \leftarrow \mathtt{G}_p, B, (\mathtt{q}_1(\overline{y});\ \ldots;\ \mathtt{q}_j(\overline{n}))$ e $\hat{l}_p$ e la sua sequenza di variabili rilevanti. Se φ è una chiamata alla relazione di dimensione per R_m dove ogni relazione di dimensione è della forma $\langle\ \mathtt{p}(\overline{x}),\ \mathtt{p'}\ (\overline{z}),\ \varphi^{\mathtt{p}(\overline{x})}_{\mathtt{p'}(\overline{z})}\ \rangle$ per ogni $\mathtt{p'}\ (\overline{z}) \in \mathtt{calls}(B) \cup \{\mathtt{q}(\overline{y})\}$ tale che $\mathtt{q}(\overline{y})$ si riferisce al program point immediatamente dopo B. Allora noi generiamo l'equazione di costo per ogni nodo della forma superiore in R_m come segue:*

$$C_p(\hat{l_p}) = \sum_{b \in bytecode(B)} T_b + \sum_{r(\overline{z}) \in calls(B)} C_r(\hat{l_r}) + C_{p\_count}(\cup_{i=1}^{n} \hat{l}_{q_i}) \bigwedge_{r(\overline{z}) \in calls(B)} (\varphi_{r(\overline{z})}^{p(\overline{x})}) \wedge \varphi_{q(\overline{y})}^{p(\overline{x})}$$

$$C_{p\_count}(\cup_{i=1}^{n} \hat{l}_{q_i}) = \begin{cases} C_{q_1}(\hat{l_{q_1}}) & \alpha_{guard}\ (G_{q_1}) \\ \dots & \\ C_{q_n}(\hat{l_{q_n}}) & \alpha_{guard}\ (G_{q_n}) \end{cases}$$

dove T_b è l'unità di costo associata al bytecode b. La relazione di costo associata a R_m e φ è definita come un set di equazioni per questo nodo.

Nella definizione precedente dobbiamo notare che:

1. I collegamenti presenti nella relazione delle dimenzioni che si crea tra le variabili di input provenienti dalla analisi delle dimensioni e l'equazioni dei costi avvengono attraverso p.

2. La guard non ha effetto sul costo, dal momento che definisce solo l'applicabilità delle condizioni delle equazioni.

3. Gli argomenti dell'equazioni dei costi sono solo gli argomenti rilevanti nel nodo.

4. Nella equazione è necessario includere l'unione di tutti gli argomenti rilevanti per ognuno dei successivi blocchi q_i.

Il costo T_b di una istruzione b dipende dalla scelta del cost model. Nel caso della ricerca della complessità o della approssimazione del numero di righe di codice che saranno eseguite, allora t_b può essere la stessa per tutte le istruzioni. D'altra parte, è possibile usare cost model più raffinati con lo scopo di stimare il tempo di esecuzione di un metodo. Tali modelli possono assegnare costi diversi a differenti istruzioni. [3]

3.1.8 Soluzione ed approssimazione delle funzioni di costo

Le *relazioni di costo* permettono di ragionare sul costo computazionale dei metodi a patto che la l'analisi delle dimensioni sia stata efficiente. Tuttavia, le *relazioni di costo* generalmente dipendono dal costo delle altre chiamate. Questo rende conveniente ottenere *soluzioni in forma chiusa* per le funzioni che corrispondono al costo del metodo. Questo può essere fatto in due fasi. La prima riguarda l'eliminazione delle variabili esistenziali, esempio quelle che appaiono sul lato sinistro, ottenendo così equazioni ricorsive. Il secondo passo riguarda l'uso degli strumenti esistenti per risolvere le equazioni di ricorrenza e/o calcolare il loro upper o lower bound.

Ottenere equazioni ricorsive Inizialmente, consideriamo le relazioni di costo che contengono solo le uguaglianze. Data una variabile esistenziale y, una *relazione di dimensione* φ e una sequenza di variabili di input x, indichiamo come risoluzione (y, φ, x) l'operazione che ritorna un espressione e, con $Vars(e) \subseteq x$ tale che $\varphi \models (y = e)$. Il risultato può essere y stessa se non è stato trovato altro e.
Esistono molte situazioni complicate, nella quale l'analisi delle dimensioni ha bisogno di approssimare informazioni ed è solo in grado di fornire intervalli nella quale una variabile può variare, invece di eguaglianze. Data una variabile y, una *relazione di dimensione* φ e una sequenza di variabili $\overline{x}$, l'intervallo di funzionamento $(y, \varphi, \overline{x})$ ritorna:

- un intervallo $[e_1, e_2]$ con $(Vars(e_1) \cup Vars(e_2)) \subseteq \overline{x}$ tale che $\varphi \models (e_1 \leq y \leq e_2)$
- altrimenti la stessa variabile y.

Ad esempio, considerando la *relazione di costo* $C_p(\overline{x}) = \sum T_b + C_q(y)$ φ, where $[e_1, e_2] = \texttt{interval}(\varphi, y, (\overline{x}))$. Come y può variare all'interno di un intervallo, è possibile stimare l'upper o il lower bound per $C_p(\overline{x})$. Per farlo, è necessario analizzare tutte le possibili varianti di y. Per fare questo

generiamo la seguente relazione:

$$C_p(\overline{x}) = \begin{cases} \sum T_b \ + C_q(e_1) \\ \sum T_b \ + C_q(e_2) \end{cases}$$

e la massimizzazione o minimizzazione delle *relazioni di costo*, dipende dal desiderio di ottenere approssimazioni rispettivamente all'upper o al lower bound, come spiegato in seguito. In casi molto complessi il costo di q può dipendere da una sequenza di variabili $\overline{y}$ piuttosto che da una sola y e l'analisi delle dimensioni potrebbe fornire intervalli (non solo uguaglianze) per molti di essi. Questo caso porta ad una formalizzazione più complessa che non verrà trattata in seguito. [3]

Approssimazione di equazioni di ricorrenza Gli algoritmi di approssimazione delle equazioni ricorsive non possono sempre trovare una *soluzione in forma chiusa* per un insieme di equazioni di ricorrenza. Comunque è spesso possibile trovare una forma chiusa che, non è soluzione di un insieme di equazioni ma, è garantito essere un upper (o lower) bound delle *funzioni di costo*. In molti casi trovare un upper (o un lower) bound può essere sufficiente. In particolare nelle *relazioni di costo* è interesante calcolare l'upper o il lower bound in due casi:

- quando ci sono rami alternativi corrispondenti alla seconda equazione di costo nella def2 (che rappresentano un invio dinamico o una diramazione condizionale)
- quando ci sono intervalli (piuttosto che uguaglianze) per la *relazione di dimensione* di alcune variabili.

Per trovare un upper bound è sufficiente modificare la definizione formale di *relazione di costo* trasformandola da:

$$C_p(\overline{x}) = \begin{cases} C_{q_1}(\overline{x}) & (G_{q_1}) \\ \dots & \\ C_{q_n}(\overline{x}) & (G_{q_n}) \end{cases}$$

in:

$$C_p(\overline{x}) = max \begin{cases} C^{up}_{q_1}(\overline{x}) & (G_{q_1}) \\ \dots & \\ C^{up}_{q_n}(\overline{x}) & (G_{q_n}) \end{cases}$$

Per ottenere il lower bound è sufficiente sostituire a max con min, e up con low.

3.2 PUBS

PUBS [4] acronimo di *Pratical Upper Bounds Solver* produce un upper bound in forma chiusa della *relazione di costo* che riceve in ingresso.

3.2.1 Relazioni di costo

Una *espressione di costo* di base è un espressione simbolica che indica le risorse accumulate e i blocchi fondamentali non ricorsivi per la definizione delle *relazioni di costo*.

Definizione 3 (Espressioni di costo di base) *Le espressione di costo di base sono della forma:* $exp ::= a|nat(l)|exp+exp|exp*exp|exp^a|log_a(exp)|a^exp|max(S)|\frac{exp}{a}|exp\text{-}a$, *dove* $a \geq 1$, l *è un espressione lineare,* S *è un insieme non vuoto di espressione di costo ,* $nat:\mathbb{Z} \to \mathbb{Q}^+$ *è definita come* $nat(v)=max(\{v,0\})$, *e* exp *soddisfa che per qualsiasi assegnamento* $\overline{v}$ *per* $vars(exp)$ *si ha* $exp[vars(exp/\overline{v})] \in \mathbb{R}^+$

Le espressioni di costo di base godono di due proprietà fondamentali: (1) come si vede dalla definizione esse sono sempre valutate per valori non negativi; (2) rimpiazzando una sotto-espressione $\texttt{nat}(l)$ con $\texttt{nat}(l')$ tale che $l' \geq l$, il risultato è un upper bound per l'espressione originale.
Una *relazione di costo* C di arietà n è un sottoinsieme di $\mathbb{Z}^n * \mathbb{R}^+$. Questo significa che per ogni tupla $\overline{v}$ di interi ci possono essere più equazioni che risolvono $C(\overline{v})$. C e D vengono usati per indicare le *relazione di costo* . L'*analisi dei costi* di un programma di solito produce multiple *relazioni ci costo* interconnesse, che prendono il nome di *sistema di relazioni di costo* (CRS in breve).

Definizione 4 (Cost Relation System) *Un sistema di relazioni di costo S è un set di equazioni della forma $\langle C(\overline{x}) = \texttt{exp} + \sum_{i=0}^{k} D_i(\overline{y}_i), \varphi \rangle$ con $k \geq 0$, dove C e tutti i D_i sono relazione di costo ; tutte le variabili $\overline{x}$ e $\overline{y}_i$ sono variabili distinte; $\texttt{exp}$ è una espressione di costo di base; e φ è una relazione di dimensione tra $\overline{x}$ e $\overline{x} \cup vars(\ \texttt{exp}\ ) \cup \overline{y}_i$.*

[4] Dato S *sistema di relazioni di costo* , $rel(S)$ indica l'insieme delle *relazione di costo* definite in S, $def(S, C)$ indica il sottoinsieme si equazioni in S il cui lato sinistro è della forma $C(\overline{x})$. Senza perdita di generalità si suppone che tutte le equazioni in $def(S, C)$ abbiano variabili con lo stesso nome nella parte sinistra. Si assume inoltre che ogni CRS S è autonoma, nel senso che ogni *relazione di costo* che appare nella parte sinistra dell'equazione S deve essere in $rel(S)$.
Semantica per CRS. Data una CRS S, una *call* è della forma $C(\overline{v})$, dove $C \in rel(S)$ e $\overline{v}$ sono valori interi. Le *call* sono valutate in due fasi. La prima fase permette di costruire un *albero di evoluzione* per la *call* . Dalla seconda fase si ottine un valore di $\mathbb{R}^+$ da aggiungere alla costante che appare nell' *albero di evoluzione* . Gli *alberi di evoluzione* sono ottenuti espandendo ripetutamente i nodi che contengono chiamate alle relazioni. Ogni espansione è eseguita in riferimento a un istanziatore appropriata della parte destra di un'equazione applicabile. Se tutte le foglie dell'albero contengono un *espressione di costo* di base allora non ci sono nodi sinistri da espandere e il processo viene terminato. Gli *alberi di evoluzione* vengono rappresentati utilizzando termini nidificati della forma $node(Call, Local\_Cost, Children)$ dove $Local\_Cost$ è una costante in $\mathbb{R}^+$ e $Children$ sono una sequenza di *alberi di evoluzione.*

Definizione 5 (Evolution tree) *Data una CRS S e una call $C(\overline{v})$, un albero $node(C(\overline{v})\ , e\ , \langle T_1, ..., T_k \rangle)$ è un albero di evoluzione per $C(\overline{v})$ in S, indicato con $\texttt{Tree}\ (C(\overline{v}), S)$ se: 1) c'è una denominazione parziale dell'equazione $\langle C(\overline{x}) = \texttt{exp} + \sum_{i=0}^{k} D_i(\overline{y}_i), \varphi \rangle \in S$ tale che φ è soddisfacibile in $\mathbb{Z}$ con $\varphi' = \varphi[\overline{x}/\overline{v}]$, e 2) esiste un assegnamento di $\overline{w}$, $\overline{v}_i$ per $vars(\texttt{exp}), \overline{y}_i$ rispettivamente tali che $\varphi'[vars(\texttt{exp})/\overline{w}, \overline{y}_i/\overline{v}_i]$ è soddisfacibile in $\mathbb{Z}$, e 3) $e = \texttt{exp}[vars(\texttt{exp})/\overline{w}]$, T_i è un albero di evoluzione $\texttt{tree}(D_i(\overline{v}_i), S))whiti = 0, ...k..$*

Al passo uno si cerca un equazione ε utile a risolvere $C(\overline{v})$, è importante notare che ci possono essere diverse equazioni applicabili. Al passo 2 si cerca un assegnamento per le variabili nella parte destra di ε che soddisfa la *relazione di dimensione* associata a ε. Questo passo è non deterministico dal momento che ci sono moltissimi assegnamenti che soddisfano tutte le *relazione di dimensione* . Al passo tre gli assegnamenti sono applicati ad `exp` e si continua ricorsivamente valutando le *call* . $Trees(C(\overline{v}), S)$ viene usato per denotare l'insieme di tutti gli *alberi di evoluzione* per $C(\overline{v})$. $Answers$ $(C(\overline{v}), S)$ è definita come $\{Sum(T)|T \in Trees(C(\overline{v}), S)\}$, dove $Sum(T)$ attraversa tutti i nodi in T e calcola la somma delle loro *espressione di costo* . [4]

3.2.2 Closed Form Upper Bounds for Cost Relation

Se C è una relazione su $\mathbb{Z}^n * \mathbb{R}^+$. Una funzione $U : \mathbb{Z}^n \to \mathbb{R}^+$ è un upper bound di C se e solo se $\forall \overline{v} \in \mathbb{Z}^n$, $\forall a \in Answers(C(\overline{v}), S)$, $U(\overline{v}) \geqslant a$. C^+ viene usato per indicare l'upper bound di C. Una funzione $f : \mathbb{Z}^n \to \mathbb{R}^+$ è in *forma chiusa* se $f(\overline{x}) =$`exp`, e `exp` è una *espressione di costo* di base tale che $vars($`exp`$) \subseteq \overline{x}$.

Un importante caratteristica del CRS, è la loro composizionalità permette il calcolo dei limiti superiori delle CRS, concentrandosi su una relazione alla volta. Il principio di composizionalità risulta un meccanismo efficace se tutte le ricorsioni dirette (esempio, tutti i cicli hanno lunghezza uno), in questo caso è possibili calcolare l'upper bound per le *relazioni di costo* che non dipendono dalle altre relazioni, chiamate *relazioni di costo indipendenti* e proseguire rimpiazzando l'upper bound calcolato sulle equazioni che chiamano queste relazioni.

Gli approcci per il calcolo degli upper bound e della complessità asintotica delle relazioni ricorsive, sono basati su ragionamenti che tengono conto di dimensione, profondità, numero di nodi ecc. di un *albero di evoluzione* . Essi tipicamente considerano due categorie di nodi: (1) nodi interni, i quali corrispondono all'applicazione di equazioni ricorsive e (2) le foglie dell'albero, le quali corrispondono all'applicazione di un caso base. L'idea è quella di contare, o ottenere un upper bound, del numero delle foglie e del numero dei

nodi interni per poi fornire per ciascuno di essi un upper bound del costo del caso base o del caso ricorsivo.
É possibile estendere lo schema di approssimazione appena menzionato per considerare tutto i possibili *alberi delle evoluzioni* che esistono per una *call* . Successivamente $|S|$ verrà usato per indicare la cardinalità dell'insieme S. Dato un *albero di evoluzione* T, $leaf(T)$ denota l'insieme dei nodi interni di T.

Proposizione 1 (node-count upper bound) *Se C è un relazione di costo e $C^+(\overline{x}) = internal^+$*
$(\overline{x}) * costr^+(\overline{x}) * leaf^+(\overline{x}) * costnr^+(\overline{x})$, dove $internal^+(\overline{x})$,
$costr^+(\overline{x})$, $leaf^+(\overline{x})$ e $costnr^+(\overline{x})$ sono le close form delle funzioni definite su $\mathbb{Z}^n \to \mathbb{R}^+$. Allora, C^+ è un upper bound di C se per ogni $\overline{v} \in \mathbb{Z}^n$ e per ogni $T \in Trees(C(\overline{v}), S)$, esso vale : (1) *$internal^+(\overline{x}) \geqslant |internal(T)|$ e $leaf^+(\overline{x}) \geqslant |leaf(T)|$;* (2) *$costr^+(\overline{v})$ è un upper bound di $\{e \mid node(\_, e, \_) \in internal(T)\}$ e* (3) *$costnr^+(\overline{v})$ è un upper bound di $\{e \mid node(\_, e, \_) \in leaf(T)\}$.*

[4]

3.2.3 Upper bound sul numero di nodi

Il meccanismo automatico, che punta ad ottenere le funzioni $internal^+(\overline{x})$ e $leaf^+(\overline{x})$ valide per un qualsiasi assegnamento di $\overline{x}$, cerca prima di tutto b e $h^+\overline{x}$, che sono rispettivamente l'upper bound del fattore di diramazione e dell'altezza di tutti gli *alberi di evoluzione*, partendo dal numero di nodi interni e dal numero di foglie. Allora,

$$leaf^+(\overline{x}) = b^{h^+\overline{x}} \qquad internal^+(\overline{x}) = \begin{cases} h^+\overline{x}, & b = 1 \\ \left(\frac{b^{h^+\overline{x}}-1}{b-1}\right) & b \geq 2 \end{cases}$$

Per una *relazione di costo* C, il fattore di diramazione b in ogni *albero di evoluzione* risultante dalla *call* $C(\overline{v})$ è limitato dal massimo numero di chiamate ricorsive che occorrono in una singola equazione per C. L'upper bound per l'altezza h^+, dato un *albero di evoluzione* $T \int Trees(C(\overline{v}), S)$ per una *relazione di costo* C può essere limitato dal numero delle chiamate ricorsi-

ve consecutive. Questo dal momento i nodi consecutivi in ogni ramo di T rappresentano le successive chiamate ricorsive che occorrono durante la valutazione di $C(\overline{v})$. La notazione di *loop* in una *relazione di costo* , introdotta in seguito, è usato per "adattare" modellare le chiamate consecutive.

Definizione 6 *Se* $\mathcal{E} = \langle C(\overline{x}) = \mathit{exp} + \sum_{i=1}^{k} C(\overline{y}_i), \varphi \rangle$ *è un equazione per la relazione di costo* C. *Allora,* $Loops(\mathcal{E}) = \{\langle C(\overline{x}) \to C(\overline{y}_i),\ \varphi' \rangle \mid \varphi' = \overline{\exists \overline{x}} \cup \overline{y}_i.\varphi,\ i = 1...k\}$ *è il set di loops inclusi in* $\mathcal{E}$. *Allo stesso modo,* $Loops(C) = \cup_{\mathcal{E} \in def(S,C)}\ Loops(\mathcal{E})$.

Il limite del numero delle chiamate ricorsive successive è essenzialmente usato nel contesto dell'analisi della terminazione. Esso dimostra che c'è una funzione f che partendo dagli argomenti dei cicli portano a un ordine parziale valido che decresce ogni due chiamate consecutive e garantisce l'assenza di percorsi infiniti e quindi termina. Questa funzione è normalmente chiamata *funzioni di rango* e verrà utilizzata per generare la funzione h^+.
Per generare la renking function, idea è questa: Dato un loop del programma della quale vogliamo trovare la renking function lineare , costruiamo un sistema corrispondente alle diseguaglianze lineari razionali. le soluzioni di questo sistema codificano la renking function lineare del ciclo del programma. Cioè siamo in grado di controllare l'esistenza di una renking function lineare risolvendo il vincolo. se esiste una funzione di renking lineare può essere costruita una soluzione al sistema di diseguaglianze lineari, una soluzione che si ottiene dal vincolo di soluzione. Se il sistema non ha soluzione allora la funzione di renking lineare non esiste. La *funzioni di rango* è definita nel modo seguente: una funzione $f : \mathbb{Z}^n \to \mathbb{Z}$ è una *funzioni di rango* per il ciclo $\langle C(\overline{x}) \to C(\overline{y}), \varphi \rangle$ se $\varphi \vDash f(\overline{x}) > f(\overline{y})$ e $\varphi \vDash f(\overline{x}) \geq 0$. L'utilizzo delle *funzioni di rango globali* permette di delimitare il numero delle possibili iterazioni compiute dai CRS non deterministiche con argomenti multipli. Dal momento che la *funzioni di rango* può ritornare un valore negativo quando è applicata a valori che corrispondono al caso base (livello delle foglie dell'albero), h^+ è definito come $h^+(\overline{x}) = \mathtt{nat}(f_C(\overline{x}))$. La funzione `nat` garantisce che i valori negativi vengano portati a 0 e, quindi forniscano una corretta approssimazione per l'altezza dell'*albero di evoluzione* con un singolo nodo. Anche se la *funzioni di rango* fornisce un upper bound per l'altezza del-

l'albero corrispondente, in alcuni casi si può ulteriormente raffinare questo risultato ed ottenere un upper bound più stretto. [4]

3.2.4 Stima del costo dei nodi

Dal momento che tutte le espressioni nei nodi dell' *albero di evoluzione* sono istanze delle espressioni che appaiono nelle equazioni corrispondenti, il calcolo di $costr^{+}(\overline{x})$ e $costnr^{+}(\overline{x})$ può essere fatto prima di trovare un upper bound a tali espressioni. Viene calcolata prima l'invariante per i valori che possono essere assunti come valori iniziali, e la si usa per calcolare l'upper bound di tali espressioni, che a loro volta verranno usate per calcolare il valore dell'upper bound di ogni espressione.

3.2.5 Invariante

Calcolare l'invariante (in termini di vincoli lineari) che, contiene tutte le chiamate ai contesti di una relazione C tra gli argomenti di una chiamata iniziale e ogni chiamata durante la valutazione che può essere fatta usando $Loops(C)$. Intuitivamente, se è presente un vincolo lineare ψ tra gli argomenti della chiamata iniziale $C(\overline{x}_0)$, e quelli della chiamata ricorsiva $C(\overline{x})$, e questo viene indicato con $\langle C(\overline{x}_0) \rightsquigarrow (\overline{x}, \psi)\rangle$, e se esiste il ciclo $\langle C(\overline{x}_0) \rightsquigarrow (\overline{y}), \varphi\rangle \in Loops(C)$, allora è possibile applicare il ciclo a uno o più step e prende un nuovo calling context $\langle C(\overline{x}_0) \rightsquigarrow (\overline{y}), \overline{\exists}\overline{x} \cup \overline{y}_i.\psi \wedge \varphi\rangle$

Definizione 7 (loop invariants.) *Per una relazione C, se T è un operatore definito come:*

$$T(X) = \begin{cases} \langle C(\overline{x}_0) \rightsquigarrow C(\overline{y}), \psi'\rangle | \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \psi\rangle \in X,\ \langle C(\overline{x}) \rightarrow C(\overline{y}), \varphi\rangle \in \\ Loops(C), \\ \psi = \exists\overline{x} \cup \overline{y}_i.\psi \wedge \varphi\rangle. \end{cases}$$

che deriva un insieme di context, da un dato context X, mediante l'applicazione di tutti i loop, allora la loop invariant I è $lfp \cup_{i\geq 0} T^{i}(I_0)$ *dove* $I_0 = \{\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \{\overline{x}_0 = \overline{x}\}\rangle\}$.

3.2.6 Upper bound delle cost expression

Una volta che le invarianti sono disponibili, la ricerca dell'upper bound delle *equazioni di costo* può essere fatta massimizzando la loro parte *nat* indipendentemente. Questo è possibile grazie alla proprietà di monotonia delle *espressione di costo* . Data una cost equation $\langle\, C(\overline{x}) = \texttt{exp}{+}\sum_{i=0}^{k} C(\overline{y}_i),\ \varphi\,\rangle$ e un invariante $\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \Psi\rangle$ una funzione calcola un upper bound $f^{'}$ per ogni f che occorre nell'operatore `nat` e allora rimpiazza in `exp` per tutte quelle espressioni f per cui non si trova un upper bound la funzione restituisce ∞. Se questa funzione è completa ovvero se Ψ e φ implicano che c'è un upper bound per un dato $\texttt{nat}(f)$, allora possiamo trovare uno *syntactically* su $varPsi^{'}$

Teorema 1 *Se $S{=}S_1 \cup S_2$ è una relazione di costo dove S_1 e S_2 sono rispettivamente l'insieme di non ricorsive e ricorsive equazioni per C e se $mathcalI = \langle C(\overline{x}_0) \rightarrow C(\overline{x}, varPsi)$ è un loop invariant per C; $E_i = \{ub_exp(\texttt{exp}, \overline{x}_0, \varphi, \Psi)|$
$\langle C(\overline{x}) = \texttt{exp}{+}\sum_{i=0}^{k} C(\overline{y}_i),\ \varphi\rangle \in S_i\}$; $contnr^{+}(\overline{x}_0) = max(E_1)$ e $costr^{+}(\overline{x}_0) = max(E_2)$. Allora per ogni call $C(\overline{v})$ e per ogni $T \in Trees(C(\overline{v2}), S)$:(1) $\forall node(\_, e, \_) \in internal(T)$ abbiamo $costr^{+}(\overline{v}) \geq e$; e (2) $\forall node(\_, e, \_) \in leaf(T)$ abbiamo $costr^{+}(\overline{v}) \geq e$.*

[4]

3.2.7 Migliorare l'accuratezza nei programmi divide et impera

Per alcuni CRS, è possibile ottenere un più accurato upper bound approssimando il costo dei livelli invece di approssimare il costo dei nodi, come indicato dalla Proposizione1. Dato un *albero di evoluzione* T, $\texttt{Sum\_Level}(T, i)$ indica la somma dei valori di tutti i nodi in T che sono alla stessa altezza.

Proposizione 2 (level-count upper bound) *Se C è una relazione di costo e se C^{+} è una funzione definita come: $C^{+}(\overline{x}) = l^{+}(\overline{x}) * costl^{+}(\overline{x})$, dove $l^{+}(\overline{x})$ e $costl^{+}(\overline{x})$ sono funzioni in forma chiusa definite su $\mathbb{Z}^n \rightarrow \mathbb{R}^{+}$. Allora C^{+} + un upper bound di C se per ogni $\overline{v} \in \mathbb{Z}^n eT \in Tree(C\overline{v}.S)$, esso detiene: (1) $l^{+}(\overline{v}) \geq depth(T){+}1$; e (2) $\forall i \in \{0, \ldots, depth(T)\}$ si ha $costl^{+}(\overline{v}) \geq$ `Sum_Level(T,i)`.*

La funzione l^+ può essere semplicemente definita come $l^+(\overline{x}) = nat(f_C(\overline{x}))+1$ (vedi 3.2.3). Generalmente non è facile trovare una $costl^+$ accurata, come fatto nella proposizione 2, non è ampliamente applicabile come nella 1. Comunque, *albero di evoluzione* per i programmi divide et impera accerta che $\texttt{Sum\_Level}(T,k) \geq \texttt{Sum\_Level}(T,k+1)$, ovvero il costo per ogni livello non aumenta da un livello all'altro. In questo caso, si prende il costo del nodo root come upper bound di $costl^+(\overline{x})$. Una condizione sufficiente affinché una *relazione di costo* entri nella classe dei divide et impera è che ogni *espressione di costo* che ha contribuito a maggiorare un equazione oppure equivale alla somma delle *espressione di costo* che a contribuito dalle corrispondenti immediate chiamate ricorsive. [4]

3.2.8 Ricorsione diretta usando Partial Evaluation

La trasformazione diretta dei CRS in forma ricorsiva è basata sulla *valutazione parziale* (PE) e viene eseguita rimpiazzando le chiamate alle relazioni intermediate con la loro definizione usando l'*unfolding*. Il primo passo in questa trasformazione è rappresentato dalla ricerca di un *binding time classification* (BTC in breve) che indichi quali sono le relazioni *residual*, ovvero quali sono le risorse che devono rimanere nei CRS. Le relazioni rimanenti sono considerate *unfoldable*, quindi esse verranno eliminate. Per il calcolo dei BTC, ad ogni CRS S viene associato un *grafo delle chiamate*, indicato $G(S)$, che è il grafo direttamente ottenuto dalla S assumendo $rel(S)$ come l'insieme di nodi ed includendo un arco (C,D) se e solo se D appare nella parte destra dell'equazione per C. La seguente definizione fornisce le condizioni sufficienti su un BTC per garantire l'ottenimento di un direttamente di un CRS ricorsivo.

Definizione 8 *Se G(S) è il call graph di S e se SCC sono le fue componenti fortemente connesse. Un BTC* `btc` *per S è direttamente ricorsiva se per tutti gli S∈SCC le seguenti condizioni sono soddisfatte: (1) se* $s_1, s_2 \in S$ *e se* $s_1, s_2 \in$ `btc`*, allora* $s_1 = s_2$*; e (2) se S ha un ciclo, allora esiste s∈S tale che s∈btc.*

La condizione 1 garantisce che tutte le le ricorsioni nel Crs trasformato siano dirette, come c'è solo una relazione residual per SCC. La condizione 2 garan-

tisce che i processi *unfolding* terminino in quanto vi è una relazione residual per ciclo. Nel BTC sono inclusi solo i *covering point* (ovvero un nodo che è parte di tutti i cicli) di SCC che contengono cicli, ma nessun nodo viene incluso se SCC non presenta cicli. Questa strada per calcolare BTC, oltre a garantire la ricorsione diretta, elimina anche tutte le relazioni che non fanno parte di un ciclo.
L'unfolding è guidato ad ogni passo da un BTC e ogni passo combina una *espressione di costo* e una *relazione di dimensione* . Definizione dell unfolding nel contesto delle CRS.

Definizione 9 (unfolding) *Data una CRS S, una call* $C(\overline{x}_0)$ *tale che* $C\in$ *rel(S), a relazione di dimensione* $\varphi_{\overline{x}_0}$ *su* $\overline{x}_0$*, e un BTC* `btc` *per S, una coppia* $\langle E, \varphi_{\overline{x}_0}\rangle$ *è un unfolding per* $C(\overline{x}_0)$ *e* $\varphi_{\overline{x}_0}$ *in S un riferimento a* `btc`*, denotando* `Unfold`$(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle$*, S,* `btc`$)\rightsquigarrow \langle E, \varphi\rangle$*, se valgono entrambe le seguenti condizioni:*

(res) $C\in$ `btc`$\wedge \neq$`true`$\wedge\langle E, \varphi\rangle = \langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle$

(unf) $C\notin$ `btc` $\wedge \neq$`true`$\wedge\langle E, \varphi\rangle = \langle (exp + e_1 + \ldots + e_k,), \varphi^{'} 2 \bigwedge_1^k \varphi_i\rangle$

dove $\langle C(\overline{x}) =$ `exp` $+ \sum_{i=1}^{k} D_i(\overline{y}_i), \varphi_C\rangle$ *è una rinominazione parziale in S tale che* $\varphi^{'} = \varphi_{\overline{x}_0}\wedge\varphi_C[\overline{x}/\overline{x}_0]$ *è soddisfacibile in* $\mathbb{Z}$ *e* $\forall 1 \leq i \leq k$ `Unfold`$(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle$*,* *S,* `btc`$)\rightsquigarrow \langle E, \varphi\rangle$.

Il primo caso, (res), è richiesto al fine di provare la terminazione. Quando viene chiamata una relazione C che è segnata come residual, si ottine come valore di ritorno la chiamata iniziale $C(\overline{x}_0)$ e la *relazione di dimensione* $\varphi_{\overline{x}_0}$ fino a quando la *relazione di dimensione* corrente $\varphi_{\overline{x}_0}$ non raggiunge il valore iniziale (`true`). Quest'ultima condizione è aggiunta al fine di forzare il passo iniziale unfolding per le relazioni segnate come residual. In tutte le successive chiamate a `Unfold` diverse da quella iniziale, la *relazione di dimensione* è non assume valore `true`. Il secondo caso (unf) corrisponde al calcolo del processo di unfolding. Il punto 1 è non deterministico, dal momento che spesso le *relazione di costo* contengono varie equazioni. Dal momento che le espressioni cono transitivamente unfolding, il passo 2 può anche fornire molteplici soluzioni. Inoltre, se la *relazione di dimensione* finale φ è insoddisfacibile, semplicemente non considerano $\langle E, \varphi\rangle$ come un unfolding valido.

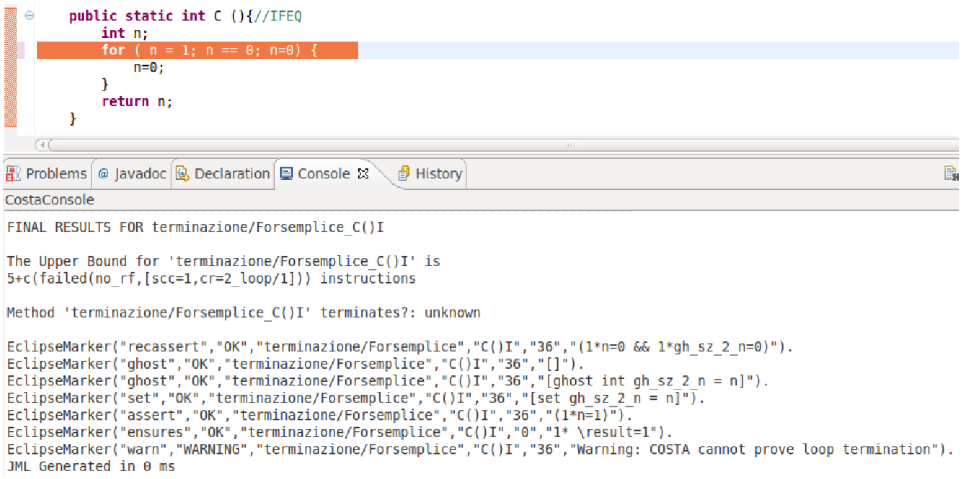
Da ogni risultato di unfolding possiamo costruire una *residual equation*. Dato l'unfolding `Unfold`($\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle$, S, `btc`)$\rightsquigarrow \langle E, \varphi\rangle$ `Unfold`() la sua corrispondente *residual equation* è $\langle C(\overline{x}_0) = E, \varphi\rangle$. Come di consueto in PE, una *partial evaluation* di C si ottiene attraverso la raccolta di tutte le residual equations per la chiamata $\langle C(\overline{x}_0), \texttt{true}\rangle$.
Correttezza di PE assicura che le soluzioni di CRS vengono mantenute. Per quanto riguarda la completezza, possiamo ottenere la ricorsione diretta se tutti SCC nel cakk graph hanno punto/i di copertura. È importante sottolineare che i cicli strutturati (for, while, ecc) e i modelli ricorsivi che troviamo nei programmi danno risultato a CRS che soddisfano questa proprietà.
Inoltre, prima di applicare PE, si verifica che la CRS termini rispetto alla query iniziale, altrimenti si potrebbe compromettere la non-terminazione e ottenere così upper bound non corretti. Questo controllo non è necessario quando le CRS sono generate da programmi imperativo.

4 Sperimentazione

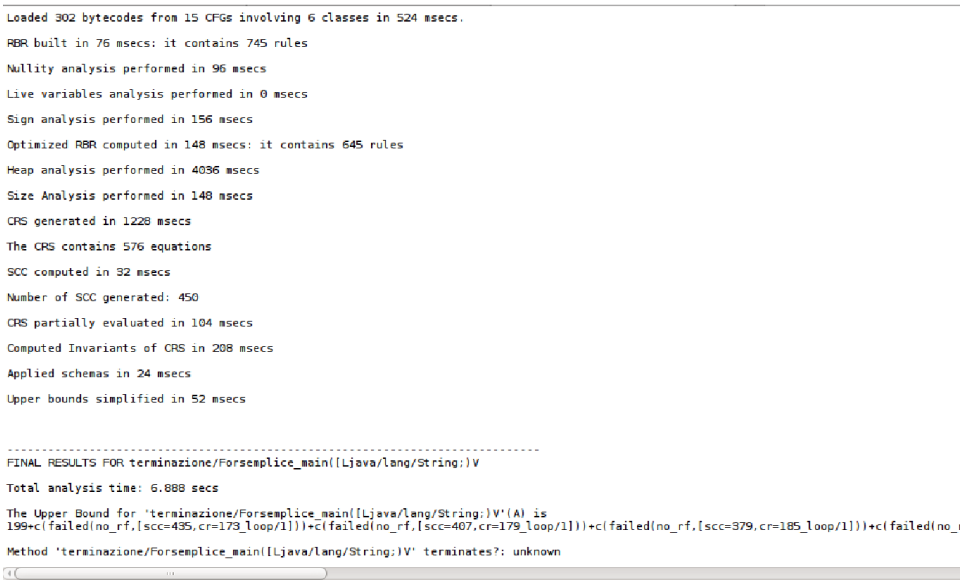
4.1 Scoperte effettuate sulle funzionalità di COSTA

L'analizzatore COSTA non è dotato di una guida che ne faciliti l'utilizzo. La sequenza di test che è stata condotta ha portato alla luce alcune problematiche relative sia all'interfaccia web sia al plugin di eclipse, oltre a rilevare alcune differenze tra le due interfacce.
Il pulgin di eclipse risulta estremamente comodo in quanto consente un'analisi immediata dei metodi senza ricorre ad caricamento dei file da analizzare. Durante l'esecuzione del plugin non è però possibile procedere a nessun tipo di operazione su eclipse, e l'analisi non è interrompibile, quindi la processazione di algoritmi complessi può causare l'interruzione di altre attività. Se il metodo contiene richiami a librerie (anche solo printf) l'analisi può andare avanti per ore.
La presentazione dei risultati da parte delle due interfacce utente risulta molto diversa. Il plugin restituisce oltre al risultato finale e ad una serie di regole relative ai passi compiuti durante l'analisi (purtroppo di nessuna utilità del momento che non ne è spiegato il significato) anche gli eventuali warning riscontrati durante l'analisi che potrebbero aver compromesso il risultato finale. Questa interfaccia fornisce solo il tempo totale occorso al completamento delle operazioni. L'interfaccia web invece produce un listato dei tempi occorsi all'esecuzione delle varie operazioni e il numero di regole intermedie ottenute ma oltre ai risultati finali (se presenti) non segnala nessun tipo di warning.
Oltre a questa differenza nell'immagine sottostante (figura 1) si vede anche la scarsa leggibilità di alcuni risultati.

Un altro punto dolente per entrambe le interfacce riguarda la segnalazione degli errori. La maggior parte degli errori evidenziati non presenta nessuna spiegazione.Inoltre a volte vengono presentati i risultati della complessità con (maximize_failed) senza segnalare a cosa questo si riferisca.
Capita a volte che l'analisi con il plugin non inizi nemmeno e appaia solo una finestra di errore (figura 2), oppure che dopo aver caricato il file sull'interfaccia web questa non si riesca a trovare i metodi che è possibile analizzare. Questo ultimo errore a volte è legato all'illegalità del no-



(a) Risultato plugin eclipse



(b) Risultato interfaccia web

Figura 1: Segnalazione di warning

me del metodo o del package, ad esempio quando viene inserito un underscore o un carattere accentato. É importante infatti segnalare che in

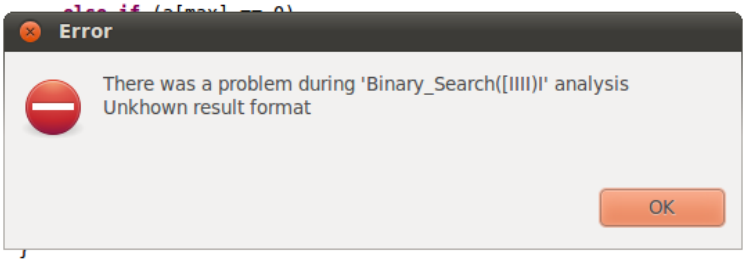
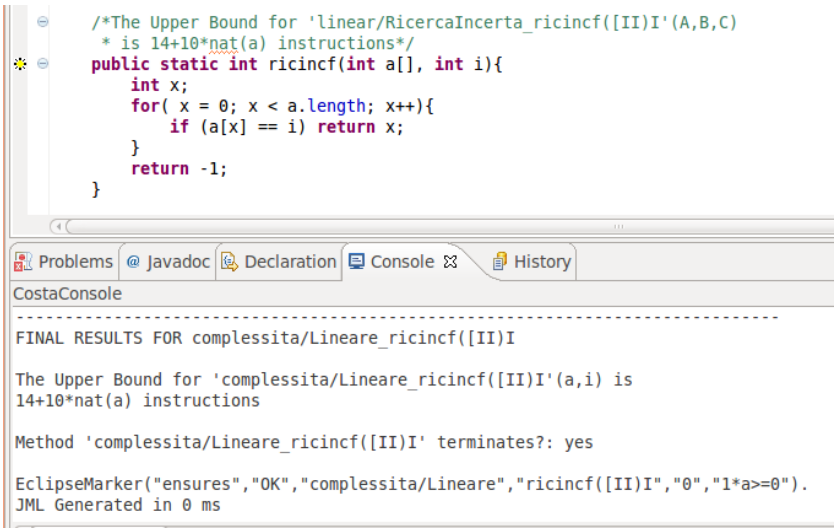


Figura 2: Finestra di errore

nessun caso è possibile inserire un carattere underscore (_) all'interno del nome della classe dal momento che questo carattere viene utilizzato della sintassi dell'analizzatore per indicare la gerarchia classe metodo. Infatti i risultati dell'analizzatore sono legati al metodo attraverso questa sintassi ″NomeDelPackage/NomeDellaClasse_ Metodo‶.
Durante la fase di test si è scelto di procedere utilizzando il plugin (a meno che non diversamente indicato), dal momento che con opportuni plugin su eclipse è possibile visualizzare direttamente il codice bytecode dei metodi. Questa scelta viene specificato dal momento che eclipse e l'interfaccia web utilizzano due compilatori diversi e questo fa mutare anche i risultati finali (figura 3).

4.2 Compilazione da parte di eclipse

Il compilatore Java di eclipse restituisce un compilato Java Bytecode diverso dal compilatore javac utilizzato dall'interfaccia web. Il risultato della compilazione rimane identico per i costrutti `do...while`, `if` e `swich`. Per quest'ultimo varia leggermente solo la sintassi. La compilazione dei costrutti `while` e `for` restituisce risultati differenti a seconda del compilatore utilizzato.
Il compilatore di eclipse compila, con le caratteristiche che verranno spiegate in seguito. I costrutto if durante la compilazione vede complementare la sua condizione. La tabella mostra come venga eseguito un salto ogni volta che la condizione all'interno dell'if non è verificata.



(a) Risultato plugin eclipse

```
FINAL RESULTS FOR complessita/Lineare_ricincf([II)I

Total analysis time: 0.104 secs

The Upper Bound for 'complessita/Lineare_ricincf([II)I'(A,B) is
13+11*nat(A) instructions

Method 'complessita/Lineare_ricincf([II)I' terminates?: yes
```

(b) Risultato interfaccia web

Figura 3: Confronto tra risultati

Condizione	Salto	Condizione	Salto
i == 0	IFNE	i == int	IF_ICMPNE
i ! = 0	IFEQ	i ! = int	IF_ICMPEQ
i > 0	IFGT	i > int	IF_ICMPGT
i >= 0	IFGE	i >= int	IF_ICMPGE
i < 0	IFLT	i < int	IF_ICMPLT
i <= 0	IFLE	i <= int	IF_ICMPLE

Questo permette di mantenere nel codice Java Bytecode lo stesso flusso presente nel codice Java. Allo stesso modo, lo swich punta a preservare il flusso del codice, anche se questo risulta essere un caso a se stante dal momento che non dovendo essere valutata una condizione, viene utilizzata un istruzione dedicata TABLESWITCH (Codice 1). Questa istruzione viene chiamata

dopo aver caricato sullo stack la variabile su cui viene eseguito lo swich. L'istruzione è poi seguita da coppie di valori composte da $< Valore : label >$ dove valore indica uno dei possibili valori assunti dalla variabile (più un eventuale caso di default) e label indica dove saltare nel caso in cui la variabile assuma il valore corrispondente.

Codice 1: SWITCH in Java Bytecode

```
ILOAD 1
    TABLESWITCH
      0: L4
      1: L5
      2: L6
      3: L7
      default: L8
```

Il costrutto Do... while è l'unico che mantiene le stesse caratteristiche del codice Java originario infatti mantiene la stessa condizione e lo stesso flusso. Come si vede nel codice sottostante.

Codice 2: Do... While generico

```
public static int B (){
    int n=1;
    do{
        n = 0;
    }while (n != 0);
    return n;
}
```

Codice 3: Do... While, Java Bytecode

```
public static B()I
   L0
    LINENUMBER 30 L0
    ICONST_1 //inizializzo
    ISTORE 0 //n ad 1
   L1
    LINENUMBER 32 L1
   FRAME APPEND [I]
    ICONST_0 //inizializzo
    ISTORE 0 //n a 0
   L2
    LINENUMBER 33 L2
    ILOAD 0 //carico n sullo stack
    IFNE L1 //se e' diversa da 0 salto
   L3
    LINENUMBER 34 L3
    ILOAD 0 //carico n sullo stack
    IRETURN
```

I costrutti for e while che risultano riconducibili l'uno all'altro, viene mantenuta la stessa condizione ma il flusso delle istruzioni varia (codice seguente). In pratica in corrispondenza del costrutto iterativo si compie un salto a dopo la fine del corpo dell' iterazione, nel punto in cui si inizializzano le variabili per poi eseguire la condizione.

Codice 4: For generico

```
public static int C (){
    int n;
    for ( n = 1; n == 0; n=0) {
        n=0;
    }
    return n;
}
```

Codice 5: While generico

```
public static int C (){
    int n = 1;
    while (n==0) {
        n=0;
        n=0;
    }
   return n;
}
```

Codice 6: For Java Bytecode

```
public static C()I
 L0
  LINENUMBER 36 L0
  ICONST_1 //inizializzo
  ISTORE 0 // n ad 1
 L1
  GOTO L2 //salto a condizione
 L3
  LINENUMBER 37 L3
 FRAME APPEND [I]
  ICONST_0 //inizializzo
  ISTORE 0 //n a 0
 L4
  LINENUMBER 36 L4
  ICONST_0 //inizializzo
  ISTORE 0 //n a 0
 L2
 FRAME SAME
  ILOAD 0 //carico n sullo stack
  IFEQ L3 //se e' uguale a 0 salto
 L5
  LINENUMBER 39 L5
  ILOAD 0 //carico n sullo stack
  IRETURN
```

4.3 Descrizione dei test effettuati

I test effettuati puntavano a valutare la capacità di concludere un risultato e la correttezza di questi ultimi, da parte di COSTA. I test possono essere suddivisi i due famiglie: quelli destinati a valutare l'effettiva terminazione di un algoritmo e quelli destinati a valutare l'accuratezza nel calcolo della complessità degli algoritmi, presentata in termini di numero di istruzioni eseguite.

Gli algoritmi che puntano a valutate la terminazione sono spesso presentati in forma iterativa e ricorsiva, o con lievi differenze al fine di valutarne le ripercussioni sui risultati ottenuti. I metodi che valutano la complessità sono suddivisi in base alla complessità dell'algoritmo che si vuole analizzare.

4.4 Test relativi alla complessità

I test sulla complessità possono essere suddivisi a seconda della complessità dell'algoritmo che viene sottoposto all' analizzatore. Le classi di complessità analizzate sono:

- Logaritmica
- Lineare
- Quadratica e Pseudo-lineare
- Cubica e Esponenziale

4.4.1 Complessità logaritmica

Gli esempi trattati per la classe di complessità logaritmica sono:

- Binary_Search:ricerca di un elemento in un array ordinato
- Binary_Search2:ricerca di un elemento in un array ordinato, distingue tra array vuoto ed elemento assente e presenta lievi ottimizzazioni.
- Conta_Zeri: restituisce l'indice dell'ultimo 0 di un array di n 0 seguiti da m 1.

Questa serie di test ha anche mostrato come le condizioni all'interno dei costrutti devono essere semplificate al massimo. Inizialmente all'interno della Bynary_Search2 la condizione iniziale che valuta se la lunghezza dell'array è maggiore di due era espressa come '**if** ((min == max) || (min == (max−1)))' ma in questo modo risultava che l'algoritmo non fosse terminante.
Tutti questi algoritmi sono stati studiati sia nella versione iterativa sia in quella ricorsiva ed in tutti i casi l'analizzatore riesce a concludere che il metodo è terminante e restituisce un upper bound per il metodo in linea con i risultati attesi come si vede dalla tabella.

Metodo	Versione ricorsiva
Conta_Zeri	$19 + 25 * log(2, 1 + nat(-2 * min + 2 * max - 3))$
Binary_Search	$21 + 27 * log(2, 1 + nat(-2 * f + 2 * l + 1))$
Binary_Search2	$21 + 31 * log(2, 1 + nat(-2 * min + 2 * max - 3))$
	Versione iterativa
Conta_ZeriIte	$26 + 20 * log(2, 1 + nat(2 * a - 5))$
Binary_SearchIte	$31 + 24 * log(2, 1 + nat(2 * a - 1))$
Binary_Search2Ite	$28 + 25 * log(2, 1 + nat(2 * a - 5))$

Analisi di Binary_Search2 e Binary_Search2Ite L'analisi dettagliata per questa classe di complessità verterà unicamente sullo studio dei metodi di Binary_Search2 dal momento che i metodi Binary_Search risultano una semplificazione di questi ultimi e i metodi Conta_Zeri sono pressoché identici a meno di qualche variabile.

La complessità calcolata dall'analizzatore per questo metodo è pari a $21 + 31 * log(2, 1 + nat(-2 * min + 2 * max - 3))$ istruzioni ovvero $21 + 31 * \log_2(1 + (-2 * min + 2 * max - 3))$ istruzioni. Questo indica che vi sono 21 istruzioni che vengono eseguite in ogni caso. Vi sono poi 31 istruzioni che vengono eseguite ad ogni ciclo. Il logaritmo in base due indica che il numero di istruzioni che vengono eseguite cresce logaritmicamente in relazione ai dati forniti in input. La base del logaritmo è in relazione al numero di parti in cui è suddiviso ad ogni passa l'array.

L'argomento del logaritmo $1 + (-2 * min + 2 * max - 3)$ fa si che la condizione '((max−min)<2)' venga rispettata facendo in modo che il logaritmo influenzi il numero delle istruzioni eseguite sono nel caso la distanza tra i due indici sia almeno di due. L'argomento del logaritmo risulterà essere maggiore di 0 solo nel caso la condizione non venga rispettata come si vede nell'equazione 1.

Nell'argomento troviamo poi min che indica l'indice più basso dell'array da analizzare, quindi alla prima chiamata è pari a 0 (pensando di far partire la ricerca su tutto l'array); mentre max indica l'indice dell'ultimo elemento dell'array e quindi normalmente punta a length-1. Un array di dieci elementi, il valore del logaritmo sarà pari a 4 che rappresenta il numero massimo di ricorsioni che vengono effettuate. sarebbero tre in tutto con un array di 15 elle menti il valore è 4, 8 e il suo massimo è quattro, siamo un pelino in

rialzo ma ci siamo.

$$\begin{aligned} 1 + (-2 * min + 2 * max - 3) &> 0 \\ -2 * min + 2 * max &> 2 \\ -min + max &> 1 \end{aligned} \tag{1}$$

Codice 7: Binary_Search2

```
private static int Binary_Search2
                    (int[] a, int min, int max, int cerca) {
    int tmp;
            // controllo array non vuoto
    if (max == -1) return -2;
            //uno o due elementi nell'array
    if ((max-min)<2){
        if (a[min] == cerca)
            return min;
        else
            if (a[max] == cerca)
                return max;
            // elemento assente
        else return -1;
    } else{
        tmp = (min+ max)/2;
        if (a[tmp]== cerca)
            return tmp;
        else if (a[tmp]< cerca)
                //ricerca nella seconda meta' dell'array
            return Binary_Search2 (a,tmp++, max, cerca);
        else
                //ricerca nella prima meta' dell'array
            return Binary_Search2 (a,min,tmp--,cerca);
    }
        }
```

La complessità calcolata per il metodo Binary_SearchIte2 è pari a $28 + 25 * log(2, 1 + nat(2 * a - 5))$ istruzioni, ovvero $28 + 25 * \log_2(1 + (2 * a - 5))$ come prima 28 indica il numero di istruzioni che verranno sempre eseguite e che non sono influenzate dai dati in input, 25 è il numero di istruzioni

eseguite ad ogni iterazione. In questo metodo dal momento che l'assenza della chiamata ricorsiva ha eliminato la necessità di passare in input gli indici dell'array da analizzare, la complessità è espressa sono in relazione alla lunghezza dell'array a.

Codice 8: Binary_SearchIte2

```
private static int Binary_SearchIte2 (int[] a, int cerca) {
    int tmp;
    int min = 0;
    int max = a.length-1;
                // controllo array non vuoto
    if (max == -1) return -2;
                //verifico ci siano almeno tre elementi nell'array
    while ((max-min)>1){
        tmp = (min+ max)/2;
        if (a[tmp]== cerca)
            return tmp;
        else
            if (a[tmp]< cerca)
                //ricerca nella seconda meta' dell'array
                min = tmp++;
        else
                //ricerca nella prima meta' dell'array
            max = tmp--;
    }
    if (a[min] == cerca)
        return min;
    else
        if (a[max] == cerca)
            return max;
    else return -1;
}
```

Allo stesso modo dell'esempio precedente l'argomento del logaritmo si annulla solo se non viene rispettata la condizione intera al while '((max−min)>1)'. La disequazione che viene impostata è la 2, che viene però espressa nei termini di a ovvero della lunghezza dell'array in ingresso, quindi un array per effettuare il ciclo deve avere lunghezza pari ad almeno 3 elementi. Questo

risultato risponde alla condizione del while che per essere eseguito necessita almeno di $(max - min = 2)$ ovvero quando nell'array ci sono almeno tre elementi.

$$\begin{aligned} 1 + (2 * a - 5) &> 0 \\ 2 * a &> 4 \\ a &> 2 \end{aligned} \tag{2}$$

Dal momento che il numero di istruzioni per questi esempi cresce in maniera logaritmica si può dire che la complessità asintotica è pari a $O(\log n)$. Come indicato all'inizio si può vedere anche dalla complessità che Conta_Zeri è pressoché identico a Binary_Search2 infatti a meno dei due indici, che indicano il numero di istruzioni utilizzate, la complessità in entrambi i casi è pari a $k * \log_2^{1+nat(-2*min+2*max-3)}$ per il caso iterativo e $k * \log_2^{1+nat(2*a-5)}$ per il caso ricorsivo.
Allo stesso modo si possono vedere le lievi ottimizzazioni introdotte da Binary_Search2 rispetto a Binary_Search dal momento che anche la sua complessità è inferiore. Binary_Search2 mostra una complessità di $21 + 31 * log(2, 1 + nat(-2 * min + 2 * max - 3))$ mentre Binary_Search mostra una complessità di $21 + 27 * log(2, 1 + nat(-2 * f + 2 * l + 1))$ Lo stesso vale per i corrispondenti metodi iterativi.

4.4.2 Complessità lineare

Gli algoritmi lineari trattati riguardano:

- RicercaIncerta: ricerca di un elemento su un array non ordinato, in cui la presenza dell'elemento cercato non è assicurata.
- Increment: Incremento di tutte le celle di un array.

Tutti questi metodi risultano terminanti e forniscono un risultato soddisfacente

RicercaIncerta Questo tipo di algoritmo è stato proposto in tre varianti una ricorsiva e due iterative, dove l'iterazione è stata eseguita in un caso con il while e nell'altro con il for. Nei due casi iterativi COSTA non rivela nessuna discrepanza nei risultati di complessità, in ambo i casi infatti la complessità

risulta pari a $14 + 10 * nat(a)$ istruzioni con a dimensione dell'array in input al metodo. Il risultato si può ritenere corretto dal momento che sono 14 le istruzioni che vengono eseguite nel caso l'array sia vuoto mentre 10 sono le istruzioni compiute per ogni elemento array per compiere la ricerca.

Codice 9: RicercaIncerta con while

```
public static int ricincw(int a [], int i){
    int x = 0;
    while (x < a.length){
        if (a[x] == i) return x;
        x ++;
    }
    return −1;
}
```

Increment Di questo tipo di algoritmo sono state proposte 6 varianti 4 iterative e due ricorsive. Le varianti iterative si differenziano per il costrutto che controlla l'iterazione (while o for) e per il passaggio della lunghezza dell'array. Le due varianti ricorsive che si differenziano per il passaggio della lunghezza dell'array. La lunghezza dell'array viene passato o come parametro oppure viene ottenuto attraverso array.length.
I risultati forniti da COSTA sono espressi nella tabella sottostante. Dalla tabella si nota che il numero di istruzioni cresce in modo lineare con la lunghezza dell'array. Array e length si riferiscono direttamente alla lunghezza dell'array mentre $a - i$ o $l - i$ indicano la lunghezza dell'array meno l'indice da cui si parte ad incrementare che normalmente è 0.
Si può notare come il numero di istruzioni cali leggermente con l'introduzione del parametro in tutti i casi e che l'esecuzione di un for abbai lo stesso costo lo stesso numero di istruzioni di un while e che entrambi meno costose della ricorsione. Queste differenze si possono notare anche visualizzando il codice bytecode. Quando non abbiamo la lunghezza passata per parametro siamo costretti a ricavarci la lunghezza dell'array e quindi invece di un semplice ILOAD per caricare il parametro nello stack viene sostituito da un ALOAD seguita da un ARRAYLENGTH, la prima per caricare l'array sullo stack la seconda per ricavarne la lunghezza. Questo spiega l'aumento di due

Metodo	Numero di istruzioni	Descrizione
IncWhitFor	$11 + 15 * nat(array)$	Incremento iterativo con utilizzo di for
IncWhitForwp	$9 + 13 * nat(length)$	Incremento iterativo con utilizzo di for e lunghezza passata come parametro
IncWhitWhile	$11 + 15 * nat(array)$	Incremento iterativo con utilizzo di while
IncWhitWhilewp	$9 + 13 * nat(length)$	Incremento iterativo con utilizzo di while e lunghezza passata come parametro
IncRicwp	$6+18*nat(l-i)$	Incremento ricorsivo con lunghezza passata come parametro
IncRic	$8+19*nat(a-i)$	Incremento ricorsivo

istruzioni tra il codice con parametro e quello senza nei casi while e for dal momento che all'interno della condizione del costrutto la lunghezza viene utilizzata due volte. Nel caso ricorsivo l'if viene contato sia nel caso base che in quello ricorsivo e sono presenti due richiami alla lunghezza dell'array (all'interno della condizione) questo quindi spiega l'aumento di due istruzioni nel caso base (da 6 a 8). Ugualmente nel caso ricorsivo è necessario aggiungere un parametro alla chiamata, questo fa si che non ci sia un guadagno di due istruzioni ma di una soltanto (da 18 a 19).
Una volta evidenziate la differenza tra passaggio con parametro o meno, si può passare alla differenza tra caso iterativo e ricorsivo, per comodità verranno analizzati solo il caso del while e quello ricorsivo entrambi con parametro. La differenza tra i due casi, nella parte ricorsiva, si basano ovviamente sulle due differenti chiamate mentre nel caso iterativo è necessario esclusivamente effettuare un incremento della variabile j più il salto alla condizione (costo due istruzioni) per la ricorsione oltre ad incrementare la variabile è necessario caricare sullo stack i tre parametri per effettuare la chiamata ricorsiva (costo 1 istruzione incremento più tre istruzioni per caricare i parametri sullo stack e una istruzioni per invocare la chiamata ricorsiva) La differenza nella parte di base in cui il caso iterativo conta tre istruzioni in più di quello ricorsivo ricorsivo è da riferirsi alla creazione e al salvataggio della variabile j e al salto

alla condizione del while (inserimento costante 0 sullo stack più salvataggio più goto). Quindi posso dire che in questo caso i risultati sono corretti.

Codice 10: Incremento array attraverso while con parametro

```
public static void IncWhitWhilewp (int array[], int length){
    int j=0;
    while (length != 0 && j < length) {
        array[j]++;
        j++;
    }
}
```

Codice 11: Incremento array con ricorsione e parametro

```
public static void IncRicwp(int a[], int l, int i){
    if (l != 0 && i < l) {
        a[i]++;
        IncRicwp(a, l, ++i);
    }
}
```

Codice 12: Incremento array attraverso while con parametro

```
public static IncWhitWhilewp ([II)V
 0  L0
    LINENUMBER 118 L0
    ICONST_0
    ISTORE 2
   L1
 5  LINENUMBER 119 L1
    GOTO L2
   L3
    LINENUMBER 120 L3
   FRAME APPEND [I]
10  ALOAD 0
    ILOAD 2
    DUP2
    IALOAD
    ICONST_1
15  IADD
    IASTORE
   L4
    LINENUMBER 121 L4
    IINC 2 1
20 L2
    LINENUMBER 119 L2
   FRAME SAME
    ILOAD 1
    IFEQ L5
25  ILOAD 2
    ILOAD 1
    IF_ICMPLT L3
   L5
    LINENUMBER 123 L5
30 FRAME SAME
    RETURN
```

Codice 13: Incremento array con ricorsione e parametro

```
public static IncRicwp([III)V
 0  L0
    LINENUMBER 130 L0
    ILOAD 1
    IFEQ L1
    ILOAD 2
 5  ILOAD 1
    IF_ICMPGE L1
   L2
    LINENUMBER 131 L2
    ALOAD 0
10  ILOAD 2
    DUP2
    IALOAD
    ICONST_1
    IADD
15  IASTORE
   L3
    LINENUMBER 132 L3
    ALOAD 0
    ILOAD 1
20  IINC 2 1
    ILOAD 2
    INVOKESTATIC
      complessita/Lineare.IncRicwp([III)V
   L1
25  LINENUMBER 134 L1
   FRAME SAME
    RETURN
```

4.5 Complessità pseudo lineare e quadratica

In questa sezione sono stati trattati i principali algoritmi di ordinamento. Questi algoritmi sono stati implementati prima mediante array ed in seguito, nel caso questa implementazione sia andata a buon fine, anche attraverso liste mediante la libreria java.util.* (java.util.List, java.util.ArrayList, java.util.Arrays) 4.6.

4.5.1 Implementazione mediante array

Gli algoritmi di ordinamento che sono stati implementati sono:

- bubblesort
- insertionsort
- mergesort
- quicksort
- selectionsort

Questa lista di algoritmi può essere suddivisa in due a seconda se l'analizzatore riesce ad ottenere un risultato o meno. L'analisi degli algoritmi quali quicksort e mergesort non fornisce un risultato di complessità. Gli algoritmi si bubblesort, insertionsort e selectionsort risultano tutti terminanti e restituiscono una *espressione di costo* valida.

Quicksort L'implementazione di questo algoritmo prevede l'utilizzo di un unico metodo più uno si swap tra due elementi, il quicksort così ottenuto risulta di terminazione ignota, anche l'indicazione dei warning non ciè di nessun aiuto visto che segnala come non terminante l'intero algoritmo e non un particolare ciclo.

Mergesort L'implementazione di questo algoritmo prevede l'utilizzo di due metodi uno principale ed uno ausiliario. Entrambe i metodi risultano terminanti e non sono segnalati warning ma all'interno dell'equazioni di costo appaiono dei maximize_failed ad indicare che non si è trovato il massimo

numero di cicli compiuti per questa operazione. All'interno dell'equazione di costo della merge il problema è dovuto alla chiamata alla funzione ausiliaria mentre, nella funzione ausiliaria l'errore maximize_failed è da riscontrare all'interno dell'ultimo for prima dell'uscita (indicato con la lettera [A]). Costa è in grado di determinare la lunghezza di quel for dal momento che non riesce a stabilire per quale porzione dell'array verrà effettuata la ricopiatura, effettivamente questo valore è possibile stabilirlo soltanto durante l'esecuzione.

L'*espressione di costo* per i due metodi sono $(nat(-2*low+2*high+-1)+1-1)*(59+22*nat(-low+high)+11*nat(-low/2+high/2+1)+10*c(maximize_failed))+4*(1+nat(-2*low+2*high-1))$ estrusioni per la mergesort e $34+22*nat(-low+high)+11*nat(-low+middle+1)+10*c(maximize_failed)$ istruzioni per la merge. Si vede con molta chiarezza come l'*espressione di costo* della merge sia inserita all'interno di della della mergesot che la richiama. Analizzando l'equazione della mergesort (tralasciando l'errore) si nota che il numero di istruzioni cresca in maniera quadratica rispetto alla dimensione dell'array $((-low+high)$ o $(-2*low+2*high))$, infatti risulta avere complessità $(n*3n)$ con n dimensione dell'array.

Mergesrt

```
public static void mergesrt(int array[], int low, int high) {
    int middle;
    if (low < high) {
        middle = (low + high) / 2;
                //la prima meta' dell'array viene divisa
        mergesrt(array,low, middle);
                //la seconda meta' dell'array viene divisa
        mergesrt(array, middle + 1, high);
                //i sotto array chiamano merge
        merge(array, low, middle, high);
    }
}
```

Funzione ausiliaria

```
public static void merge(int array[], int low, int middle, int high) {
    int[] aa = new int[array.length];
    int l=low, m=middle+1, aux=low;
            //mentre una delle due meta' dell'array non e' esaurita
    while(l<= middle && m <= high){
            //confronta i primi due elementi non copiati in aa[]
            //e aggiungi il minore in aa[]
        if (array[l] <= array[m]){
            aa[aux] = array[l];
            l++;
        }
        else {
            aa[aux] = array[m];
            m++;
        }
        aux++;
    }
    m = aux;
            //se la prima meta e' esaurita ignoro altrimenti copio
            //i suoi elementi sul fondo dell' array
    for (int h = l; h <= middle; h++) {
        array[m]= array[h];
        m++;
    }
            //Copia l'array ausiliario nel principale
[A] for (int q = low; q < aux; q++)
        array[q]=aa[q];
}
```

Bubblesort Questo algoritmo di ordinamento è stato studiato unicamente nella sua versione ricorsiva, oltre a quelle qui presentata ne esisteva un'altra versione che prevedeva il richiamo di un metodo esterno per lo swap tra elementi, l'unica differenza riscontrata però era un aumento del numero di istruzioni per il secondo ciclo for (che passava da 30 a 33), causato dal caricamento dei valori di input e dalla successiva chiamata a questo metodo.

Babblesort

```
public static void BabbleSrt(int array[]){
    int i,j;
    int tmp;
    for ( i = 0; i < array.length−1; i++)
        for (j = array.length−1; j > 0; j−−)
            if (array[j] < array[j−1]){
                tmp = array[j];
                array[j] = array[j−1];
                array[j−1] = tmp;
            }
}
```

Il numero di istruzioni necessari e per eseguire questo metodo risulta uguale a $10 + nat(array - 1) * (15 + 30 * nat(array - 1))$ istruzioni. Il valore $array{-}1$ si riferisce alla condizione intera ai cicli for, COSTA ovviamente non riesce a comprendere che, il -1 sia necessario allo scopo di evitare overflow e quindi lo ritiene una limitazione allo scorrimento dell'array in input. Le dieci istruzioni iniziali vengono sempre eseguite anche se l'array è composto da un solo elemento o meno esse infatti comprendono, l'inizializzazione delle variabili, la valutazione del primo for e l'istruzione di ritorno. Ognuno dei due fattori del prodotto si riferisce a un differente ciclo for. COSTA rivela quindi che il numero di istruzioni eseguite per questo metodo crescono in maniera quadratica con la dimensione dell'array di input.

Insertionsort Le versioni dell'Insertionsort analizzate sono tre una ricorsiva e due iterative, la prima che usa un costrutto while annidato all'interno di un costrutto for e la seconda che prevede l'annidamento di due costrutti for. I risultati dell'analisi sono riportati nella tabella sottostante. Si può no-

Metodo	Numero di istruzioni	Descrizione
Insertion_srt	$8 + nat(array - 1) * (25 + 18 * nat(array - 1))$	Ciclo while annidato nel for
Insertion_srt_wf	$8 + nat(array - 1) * (25 + 18 * nat(array - 1))$	Due cicli for annidati
InsertionSrt_R	$7 + nat(array - x) * (37 + 18 * nat(array - x))$	Versione ricorsiva

tare come l'impiego di costrutti diversi nell'iterazione non provochi nessun mutamento al numero di istruzioni eseguite. La differenza tra caso iterativo e ricorsivo è lieve, ma in entrambi i casi il numero di istruzioni cresce in maniera quadratica con la dimensione dell'array in input. Il caso ricorsivo risulta però leggermente più preciso, infatti mentre $array - x$ indica la dimensione dell'array meno l'indice dal quale si inizia a scorrere, che inizialmente è 0, quindi la lunghezza dell'array viene contata tutta nel caso iterativo la lunghezza dell'array è diminuita di un elemento.

Isertionsort con utilizzo di while

```
public static void insertion_srt(int array[]){
    int tmp;
    int i,j;
    for ( i = 1; i < array.length; i++){
        j = i;
        tmp = array[i];
        while ((j > 0) && (array[j-1] > tmp)){
            array[j] = array[j-1];
            j--;
        }
        array[j] = tmp;
    }
}
```

Insertionsort Ricorsiva

```
public static void InsertionSrt_R(int array[], int x){
    int i, j, tmp;
    if (x <= array.length-1){
        i = x;
        j = x;
        tmp = array[x];
        while (x <= array.length-1){
            if (tmp > array[x]){
                tmp = array[x];
                j = x;
            }
            x = x+1;
        }
        array[j] = array[i];
        array[i] = tmp;
        InsertionSrtRicorsiva(array, i+1);
    }
}
```

Selectionsort L'algoritmo Selectionsort è stato presentato in tre varianti una iterativa e due ricorsive, una delle quali sfrutta un metodo ausiliario. I risultati dell'analisi sono riportati nella tabella sottostante. La differenza

Metodo	Numero di istruzioni	Descrizione
Selection_srt	$10 + nat(array - 1) * (32 + 14 * nat(array - 1))$	Versione iterativa
Selection_srt_r	$7 + nat(array - x - 1) * (36 + 14 * nat(array - x - 1))$	Versione ricorsiva
Selection_srt_r2	$7 + nat(array - x - 1) * (44 + 20 * nat(array - x - 1))$	Versione ricorsiva con metodo ausiliario

tra caso iterativo e ricorsivo è lieve, ma in entrambi i casi il numero di istruzioni cresce in maniera quadratica con la dimensione dell'array in input. In questo caso per tutte e tre le versioni dell'algoritmo la lunghezza dell'array è diminuita di un elemento.

Valutando le differenze tra i due casi ricorsivi si nota come, giustamente, la chiamata alla funzione ausiliaria si rivolga unicamente al secondo elemento

della moltiplicazione che si rivolge nel primo caso al costrutto for e nel secondo alla chiamata al metodo ausiliario.

Selectionsort ricorsivo con metodo ausiliario

```
public static void Selection\_srt\_2r(int array[], int x){
    int least, tmp, j;
    if (x < array.length-1){
        j=x+1;
        least = x;
        sortaux(array, j, least);
        least = c;
        tmp = array[least];
        array[least] = array[x];
        array[x] = tmp;
        x++;
        Selection\_srt\_2r(array, x);
    }
}
public static void sortaux(int a[], int j, int least){
    if (j < a.length){
        if (a[j] < a[least])
            least=j;
        j++;
        sortaux (a, j, least);
    }else{
    c = least;
    }
}
```

Selectionsort ricorsivo

```
public static void Selection\_srt\_r(int array[], int x){
    int least , tmp, j;
    if (x < array.length-1){
        for (j = x+1, least = x; j <array.length; j++)
            if (array[j] < array[least])
                least =j;
        tmp = array[least];
        array[least] = array[x];
        array[x] = tmp;
        x++;
        Selection\_srt\_r(array, x);
    }
}
```

4.6 COSTA e le librerie

L'analizzatore dovrebbe riuscire ad analizzare anche metodi che sfruttano librerie per questo motivo i precedenti algoritmi quadratici, per cui COSTA riusciva ad ottenere un risultato finale sono stati implementati attraverso le liste. L'analisi è stata fatta attraverso l'interfaccia web visto che il plugin in restituisce l'errore mostrato in figura 2 I metodi analizzati sono la versione ricorsiva della bubblesort, l'insertionsort implementata attraverso un while annidato in un for e la selectionsort iterativa. Nessuno dei tre metodi una volta analizzato restituisce un risultato comprensibile. Tralasciando la insertionsort per cui non si riesce a raggiungere neanche un risultato. Un risultato così espresso non è di alcun aiuto e anche nel caso si decidesse di capire quale dei tre metodi richiede il minor numero di istruzioni il confronto è complesso.

Dal momento che i risultati ottenuti erano poco confortanti si è pensato di implementare le liste tramite un interfaccia, ma a quanto pare nessuna delle interfacce utente sembra essere totalmente in grado di gestire questo tipo di struttura. Ovvero, l'implementazione dei metodi risulta analizzabile e il suo risultato è corretto in quasi tutti i casi, analizzando la classe che contiene il metodo main in cui si richiamano alcuni metodi su una lista,

Metodo	Numero di istruzioni
Babblesort	$49+nat(B-1)*(48+nat(B-1)*(40+c(valueOf(I))+4*c(get(I))+3*c(intValue())+2*c(set(IObject)))+c(valueOf(I))+c(set(IObject))+4*c(get(I))+3*c(intValue()))+nat(B-1)*(40+c(valueOf(I))+4*c(get(I))+3*c(intValue())+2*c(set(IObject)))+c(valueOf(I))+c(set(IObject))+4*c(get(I))+3*c(intValue())$
Insertionsort	$3+c(failed(no_rf,[scc=10,cr=2_loop/4]))$
Selectionsort	$43+nat(B)*(43+c(valueOf(I))+nat(B-1)*(18+2*c(get(I))+2*c(intValue()))+4*c(get(I))+3*c(intValue())+2*c(set(IObject)))+c(valueOf(I))+nat(B-1)*(18+2*c(get(I))+2*c(intValue()))+c(set(IObject))+4*c(get(I))+3*c(intValue())$

fornisce un risultato contenente (maximize_failed) anche nel caso si provi a rimuovere un elemento da una lista.

4.7 Complessità cubica ed esponenziale

In questa parte i metodi testati sono quattro.

Tutti e quattro i metodi, piuttosto semplici, risultano terminanti e restitui-

Metodo	Numero di istruzioni	Descrizione
Fibonacci	$-13+18*pow(2,nat(n-2))$	Calcolo n-esimo numero di fibonacci versione iterativa
Esponenziale	$28*((pow(3,nat(1*in+-3))-1)/2)+12*pow(3,nat(in-3))$	
Cubow	$14+nat(n)*(8+nat(n)*(8+6*nat(n)))$	Metodo formato da tre cicli annidati (utilizzo while)
Cubof	$10+nat(n)*(10+nat(n)*(10+6*nat(n)))$	Metodo formato da tre cicli annidati (utilizzo for)

scono un risultato plausibile.

I due esempi di complessità cubica, restituiscono risultati molto simili, le lievi variazioni delle costanti sono dovute all'inizializzazione delle variabili in diversi punti del programma. Dove viene utilizzato il costrutto while

l'inizializzazione è all'inizio, mentre per il costrutto for l'inizializzazione è all'interno del costrutto stesso. I risultati forniti dall'analizzatore mostrano come effettivamente i due metodi crescono in maniera cubica rispetto alla dimensione dell'input n che determina il numero di iterazione che deve essere compiuto da ogni costrutto

Codice 14: Complessità cubica (while)

```
public static int Cubow(int n){
    int i1 = 0, i2 = 0, i3 = 0, r=-1;
    while (i1 < n){
        i1++;
        while (i2 < n){
            i2++;
            while (i3 < n){
                r = 0;
                i3++;
            }
        }
    }
    return r;
}
```

Complessità cubica (for)

```
public static int Cubof (int n) {
    int i1, i2, i3, r = -1;
    for (i1 = 0; i1 < n; i1++)
        for (i2 = 0; i2 < n; i2++)
            for (i3 = 0; i3 < n; i3++)
                r = 0;
    return r;
}
```

I due esempi di complessità esponenziale, hanno lo scopo di testare l'analizzatore su algoritmi non polinomiali.

Fibonacci

```
public static int Fibonacci(int n) {
    if (n < 1) return 1;
    else return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

Funzione mistero

```
public static int Esponenziale(int[] n, int in) {
    //int c;
    if (n.length==0 || in<4 || in>n.length) return 2;
    else {
        return Esponenziale(n, in−1) ∗ Esponenziale(n, in−2)∗
                                        Esponenziale(n, in−3);
    }
}
```

L'analizzatore COSTA restituisca anche in questo caso risultati validi infatti il metodo fattoriale restituisce un valore la cui complessità asintotica è pari a 2^n, mentre il metodo esponenziale restituisce un valore la cui complessità asintotica è pari a 3^n.

4.8 Test relativi alla terminazione

I test di terminazione riguardano la funzione di Mc Carthy 91 e alcuni algoritmi che puntano ad escludere il ciclo indefinito.

4.8.1 Funzione 91 di Mc Carthy

In matematica discreta, la Funzione 91 di Mc Carthy è una funzione ricorsiva che ritorna 91 per tutti gli argomenti $n \leq 101$ e ritorna $n - 10$ per $n > 101$. Limitatamente all'insieme degli interi minori di 102 essa, quindi, è un'endofunzione avente un unico punto fisso. Tale funzione fu ideata dall'informatico John Mc Carthy.
La Funzione 91 di Mc Carthy è definita come segue:

$$M(n) = \begin{cases} n - 10, & \text{se } n > 100 \\ M(M(n + 11)), & \text{se } n \leq 100 \end{cases}$$

La funzione 91 è stato introdotta nei documenti pubblicati da Zohar Manna, Amir Pnueli e John Mc Carthy nel 1970. Questi documenti rappresentano i primi sviluppi verso l'applicazione di metodi formali di verifica dei programmi. La funzione 91 è stata scelta come modello complesso di ricorsione (in contrasto con i modelli più semplici, come la definizione di f (n) per mezzo di

f(n - 1)). L'esempio è stato reso popolare dal libro di Manna, Teoria matematica di calcolo (1974). Esso è visto come un problema sfida per programmi automatici di verifica, che spesso ragionano più facilmente su computazioni non ricorsive. A prova di ciò, il libro di Manna include un algoritmo non-ricorsivo che simula la versione originale (ricorsiva) 91 funzione. Molti dei documenti che parlano di verifica automatica (o la prova di terminazione) segnalano di essere in grado di gestire solo la versione non-ricorsiva della funzione 91.[14] La funzione di Mc Carthy, nel test effettuato, è stata calcolata sia con un algoritmo ricorsivo sia con uno iterativo. I risultati ottenuti mostrano come COSTA riesca a provare che l'algoritmo ricorsivo termini mentre non è in grado di provare la terminazione dell' algoritmo iterativo. Questo è un segnale del fatto che la trasformazione, effettuata dall' analizzatore per passare dell'algoritmo ricorsivo a quello iterativo non è perfetta. Sicuramente l'analizzatore è mandato in crisi dall'introduzione dell'indice che tiene conto del numero di iterazioni della funzione di Mc Cartney che devono essere ancora eseguite probabilmente mentre nel caso ricorsivo l'analizzatore vede o che il valore di x è cresciuto a sufficienza da far terminare la funzione oppure continuare ad aumentare questo lo interpreta come segnale di terminazione sicura. Nell'altro caso l'introduzione dell'indice che aumenta all'aumentare del valore di x lo sconvolge.

4.8.2 Esclusione di cicli

Questa parte è composta dal alcuni algoritmi che puntano ad escludere un ciclo infinito presente all'interno del metodo stesso. Questa serie di test ha due finalità:

- verificare se COSTA riconosce questi cicli come terminanti, se evidenzia parti di codice di cui non riesce a provare la terminazione oppure se fallisce non riuscendo a valutare la complessità dell'algoritmo.
- verificare se tra una sequenza di costrutti COSTA riesce ad individuare quello infinito.

Il secondo scopo era inizialmente quello primario, infatti, dal momento che normalmente un analizzatore statico non permette l'individuazione l'esclusione di cicli infiniti, si pensava di valutare la sua efficacia a tale scopo

risultando un ulteriore debugger utile nella ricerca di errori sfuggiti al compilatore. Dal momento che COSTA analizza codice Java Bytecode la fase di test punterà anche a capire quali costrutti condizionali, sono "riconosciuti' in maniera più efficace:

- Costrutti che compiono raffronti con zero:

  `IFEQ`, effettuare salto a label se il valore sulla cima dello stack è uguale a 0.

  `IFGE`, effettuare salto a label se il valore sulla cima dello stack è maggiore o uguale a 0.

  `IFGT`, effettuare salto a label se il valore sulla cima dello stack è maggiore di 0.

  `IFLE`, effettuare salto a label se il valore sulla cima dello stack è minore o uguale a 0.

  `IFLT`, effettuare salto a label se il valore sulla cima dello stack è minore di 0.

  `IFNE`, effettuare salto a label se il valore sulla cima dello stack è diverso da 0.

- Costrutti che compiono raffronti tra interi:

  `IF_ICMEQ`, effettua un salto a label se i due valori sulla cima dello stack risultano uguali.

  `IF_ICMGE`, effettua un salto a label se il secondo valore sulla cima dello stack risulta maggiore o uguale al primo valore sulla cima dello stack.

  `IF_ICMGT`, effettua un salto a label se il secondo valore sulla cima dello stack risulta maggiore del primo valore sulla cima dello stack.

  `IF_ICMLE`, effettua un salto a label se il secondo valore sulla cima dello stack risulta minore del primo valore sulla cima dello stack.

  `IF_ICMLT`, effettua un salto a label se il secondo valore sulla cima dello stack risulta minore o uguale al primo valore sulla cima dello stack.

`IF_ICMNE`, effettua un salto a label se i due valori sulla cima dello stack risultano diversi.

I test puntano a verificare l'abilità di COSTA nell'individuare cicli infiniti oppure nell'ignorarli, quando questi sono esclusi dal flusso di esecuzione. Questi costrutti saranno utilizzati attraverso i tipici costrutti condizionali (if...then...[else...] oppure swich...case) o iterativi (while, do...while, for...) del linguaggio Java. In particolare sarà effettuata un'analisi che prevede:

- l'utilizzo dei costrutti precedentemente indicati singolarmente
- l'utilizzo di sequenze di costrutti

In seguito verranno poi studiati metodi che contengono sia costrutti nella cui condizione saranno presenti sia raffronti con zero che raffronti tra interi.

Utilizzo di costrutti singoli In questa parte il costrutto verrà inserito nel metodo in modo che contenga un costrutto, la cui la condizione risulti sempre falsa ed il corpo del costrutto contenga un ciclo infinito. Ogni costrutto, condizionale o iterativo, è stato testato con ognuno dei costrutti condizionali del Java Bytecode .

I costrutto if è stato sottoposto a due tipi di test, uno che puntava a fare in modo che la condizione fosse sempre falsa ed inseriva il ciclo nel corpo dell'if ed un'altra, che prevedeva che la condizione fosse sempre vera e il ciclo infinito si trovasse nel ramo else. Il ciclo infinito in questo caso è sempre e solo ottenuto attraverso la ricorsione.
Quando la condizione sfrutta costrutti condizionali che prevedono il raffronto con zero, in ogni test effettuato risulta sempre terminante.

Utilizzo IFEQ

```
public static int B (int y){
    int r = 1;
    if (r != 0)
        return r;
    else B(r);
    return −1;
}
```

Utilizzo IFEQ

```
public static int B (int y){
int r = 0;
if (r != 0)
    B(r);
return r;
}
```

Quando la condizione sfrutta costrutti che prevedono il confronto tra interi, il metodo risulta generalmente terminante tranne in due casi. Il primo caso prevede l'utilizzo del costrutto IF_CMPEQ mentre il ciclo è presente nel corpo dell'if, il secondo caso prevede l'utilizzo del costrutto IF_CMPNE mentre il ciclo risulta essere nel ramo else.
Il risultato ottenuto ci ha suggerito che la non terminazione non dipendesse strettamente dall'istruzione utilizzata. Infatti si nota che COSTA si trovi in difficoltà non in corrispondenza di queste due istruzioni ma nel caso debba raffrontare l'uguaglianza tra due elementi, dal momento che il ciclo nel corpo dell'if verrebbe eseguito se i due interi fossero uguali nel primo caso e il corpo dell'else verrebbe eseguito se i due interi fossero uguali.

Utilizzo IF_CMPEQ

```
public static int BB (int y){
    int p = 2;
    if (p != 2)
        BB(p);
    return p;
}
```

Utilizzo IF_CMPNE

```
public static int CC (int y){
    int p = 1;
    if (p == 1)
        return p;
    else CC(p);
    return 12;
}
```

Questo ha portato a compiere altri test per approfondire la problematica. Da questi test è uscito che non è l'istruzione a condizionare il risultato ma l'intero che viene utilizzato, infatti dagli esempi successivi si vede che modificando il valore delle variabili ed arrivando a produrre un confronto indiretto con zero il metodo risulta terminante malgrado il codice Java Bytecode risulti invariato.

Utilizzo IF_CMPEQ

```
public static int BB (int y){
    int p = 0;
    int x = 0;
    if (p != x)
        BB(p);
    return p;
}
```

Utilizzo IF_CMPNE

```
public static int CC (int y){
    int p = 1;
    if (p == 1)
        return p;
    else CC(p);
    return 12;
}
```

Java Bytecode per l'if

```
public static BB(I)I
   L0
    LINENUMBER 77 L0
    ICONST_0
    ISTORE 1
   L1
    LINENUMBER 78 L1
    ICONST_0
    ISTORE 2
   L2
    LINENUMBER 79 L2
    ILOAD 1
    ILOAD 2
    IF_ICMPEQ L3
   L4
    LINENUMBER 80 L4
    ILOAD 1
    INVOKESTATIC terminazione
                /IfSemplice.BB(I)I
    POP
   L3
    LINENUMBER 81 L3
   FRAME APPEND [I I]
    ILOAD 1
    IRETURN
```

Java Bytecode per l'if... else

```
public static CC(I)I
  L0
   LINENUMBER 93 L0
   ICONST_0
   ISTORE 1
  L1
   LINENUMBER 94 L1
   ICONST_0
   ISTORE 2
  L2
   LINENUMBER 95 L2
   ILOAD 1
   ILOAD 2
   IF_ICMPNE L3
  L4
   LINENUMBER 96 L4
   ILOAD 1
   IRETURN
  L3
   LINENUMBER 97 L3
  FRAME APPEND [I I]
   ILOAD 1
   INVOKESTATIC terminazione
            /IfElseSemplice.CC(I)I
   POP
  L5
   LINENUMBER 98 L5
   BIPUSH 12
   IRETURN
```

Il codice Java Bytecode rimane invariato dal momento che il compilatore non è in grado di individuare che almeno una delle due variabili è zero. Al contrario COSTA quando stabilisce gli intervalli dei possibili valori assumibili dalle variabili, riconosce che è zero, da questo ragionamento si deduce che l'analizzatore compie maggiore fatica nel valutare condizioni che non contengono almeno uno zero anche se indirettamente.

Il costrutto while è stato sottoposto a due tipi di test uno che prevedeva la creazione di un ciclo attraverso l'iterazione e l'altro prevede la creazione del ciclo attraverso la ricorsione. Il ciclo è sempre inserito all'interno del corpo del while e la condizione è sempre falsa.
Nel caso il ciclo sia ottenuto attraverso l'iterazione per qualsiasi costrutto Java Bytecode la terminazione non risulta provata. Nel caso di ciclo ottenuto con la ricorsione i metodi che fanno uso di costrutti che prevedono il raffronto con zero risultano terminanti mentre gli altri no.
Dopo le scoperte effettuate durante i test sull'if è stata la volta di una serie di test che prevedeva il confronto indiretto con zero. Mentre nei casi iterativi non si nota alcun miglioramento, nel caso ricorsivo i metodi che usano IF_ICMPNE, IF_ICMPLT e IF_ICMPGE risultano terminanti anche se non si comprende esattamente quale sia la motivazione di questo fatto.

Utilizzo IF_ICMPNE

```
public static int BB (){
    int n = 2;
    while (n != 1)
        BB();
    return n;
}
```

Utilizzo IF_ICMPNE

```
public static int BB (){
    int n = 0;
    int x = 0;
    while (n != x)
        BB();
    return n;
}
```

Utilizzo IF_ICMPLT

```
public static int EE (){
    int r = 2;
    while (r < 1)
            EE();
    return r;
```

Utilizzo IF_ICMPLT

```
public static int EE (){
    int r = 1;
    int x = 0;
    while (r < x)
            EE();
    return r;
}
```

Utilizzo IF_ICMPGE

```
public static int FF (){
    int r = 1;
    while (r>=2)
            FF();
    return r;
}
```

Utilizzo IF_ICMPGE

```
public static int FF (){
    int r = −1;
    int x = 0;
    while (r>=x)
            FF();
    return r;
}
```

Il costrutto for dal momento che come è stato mostrato in precedenza risulta molto simile al while, i test effettuati risultano gli stessi. Contrariamente a quelle che potevano essere le aspettative, in nessun caso, né iterativo né ricorsivo, né per nessun costrutto Java Bytecode. Guardando il codice Java Bytecode dei metodi, malgrado in nessuno dei casi il corpo del costrutto non viene mai eseguito e nel caso del for questo implica che l'aggiornamento dell'indice non venga mai effettuato, questo aggiornamento mette in crisi l'analizzatore.
La differenza tra il costrutto while ed il costrutto for è visibile nell'esempio sottostante che fa uso di IFNE.

Esempio while

```
public static int B (){
    int n = 0;
    while (n != 0)
            B();
    return n;
}
```

Esempio for

```
public static int B (){
    int n;
    for ( n = 0; n != 0; n=1) {
        B();
    return n;
}
```

Java Bytecode per il while

```
 public static B()I
   L0
    LINENUMBER 28 L0
    ICONST_0
    ISTORE 0
   L1
    LINENUMBER 29 L1
    GOTO L2
   L3public static int C (int y){//IFEQ
            int n = 1;
            int q=0;
            do{ n++;
                    n%=2; //0 //1 esco
            }while (n==q);
//Ciclo infinito ritenuto infinito
            return n;
        }
    LINENUMBER 30 L3
   FRAME APPEND [I]
    INVOKESTATIC terminazione
        /WhileSempliceRicorsivo.B()I
    POP
   L2
    LINENUMBER 29 L2
   FRAME SAME
    ILOAD 0
    IFNE L3
   L4
    LINENUMBER 31 L4
    ILOAD 0
    IRETURN
```

Java Bytecode per il for

```
 public static B()I
   L0
    LINENUMBER 30 L0
    ICONST_0
    ISTORE 0
   L1
    GOTO L2
   L3
    LINENUMBER 31 L3
   FRAME APPEND [I]
    INVOKESTATIC terminazione
        /ForSempliceRicorsivo.B()I
    POP
   L4
    LINENUMBER 30 L4
    ICONST_1 //assegnazione
    ISTORE 0 //n=1
   L2
   FRAME SAME
    ILOAD 0
    IFNE L3
   L5
    LINENUMBER 33 L5
    ILOAD 0
    IRETURN
```

Il costrutto Do... while è stato sottoposto ad una serie di test lievemente differente rispetto a quella dei costrutti precedenti dal momento che il corpo del while viene sempre eseguito. Il corpo del costrutto è stato strutturato in modo che le istruzioni al suo interno rendano la condizione la prima volta vera e la seconda volta falsa, in modo che il corpo venga eseguito sempre due volte. In questa serie di test non si è fatto uso di ricorsione. Le condizioni che non prevede raffronti con zero risultano sempre non terminanti, neanche se si esegue un raffronto indiretto con zero. Le uniche condizioni terminanti che prevedono il raffronto con zero sono quelle che utilizzano le istruzioni `IFEQ` e `IFLE` anche se non si comprende esattamente quale sia la motivazione di questo fatto.

Utilizzo IFEQ

```
public static int C (int y){
    int n = 1;
    int q=0;
    do{ n++;
        n%=2; //0 --> 1 esco
    }while (n==q);
    return n;
}
```

Utilizzo IFLE

```
public static int G (int y){
    int r  = 1;
    do{ r++;
        r%=2; //0 --> 1 esco
    }while (r<=0);
    return r;
}
```

Sequenze di costrutti Le sequenze di costrutti prevedono una successione dello stesso tipo di costrutto, posto sia a formare cicli che no allo scopo di verificare se COSTA riesce a rilevare quali costrutti sono non terminanti. Ovviamente anche in questo caso come nei precedenti i cicli non terminanti saranno posti in modo da non essere mai eseguiti. In questa parte si farà ricorso solamente a codice iterativo dal momento che il codice ricorsivo tende a restituire warning su tutto il metodo (o non restituisce warning malgrado la terminazione sia sconosciuta) e non su una iterazione specifica. Questo ovviamente esclude cicli di soli if, o swich...case.

Sequenza di while ha previsto l'analisi di un metodo che può essere espresso come una sequenza di quattro blocchi:

1. Due while annidati di cui il più interno non terminante
2. Due while annidati entrambi terminanti
3. Un while non terminante
4. Un while non terminante con al suo interno due cicli while uno terminante e uno no

```
public static int WhileUse(int x){
    int i = 0;
    int j = 1;
    int k = 5;
    int p = 3;

    while(k != 10){      //Punto 1
        j ++;
        k++;
        while (i == 1)
            i = 1;
    }

    while(k > 0){        //Punto 2
        j *= 10;
        j %= 5;
        k--;
        while(p > 0){
            j *= 10;
            j %= 5;
            p--;
        }
    }

    while (i == 1)       // Punto 3
        i = 1;

    p = 10;
    while (i == 1){      // Punto 4
        i = 1;
        while(p > 0){
            j *= 10;
            j %= 5;
            p--;
        }
        while(i == 1)
            i = 1;
    }
    return j;
}
```

Si ricorda che i cicli while ottenuti con iterazione risultano non terminanti per qualsiasi condizione di ingresso. Quindi i risultati del test non sono influenzati dalla condizione del while. L'analisi di questo metodo, restituisce terminazione sconosciuta, ma segnala cinque warning che possono essere ricondotti ad entrambi i while del primo punto, al while non terminante del terzo punto e ai 2 while non terminanti del quarto punto. Questo risultato

ci mostra come COSTA punti ad analizzare l'intero metodo, e non si fermi nel caso trovi un punto che ritiene non terminante. I due casi da segnalare riguardano il primo e l'ultimo punto. Nel primo punto si nota come la non terminazione del ciclo interno si estenda anche al ciclo esterno viene giustamente riconosciuto anche esso come non terminante. Nel quarto punto si nota invece come la non terminazione del ciclo esterno non sottintenda la non terminazione dei cicli interni.

Sequenza di for ha previsto l'analisi di un metodo estremamente simile a quello del caso precedente, ed anche i risultati rilevati sono identici a quelli del caso precedente. L'unica differenza sostanziale che si può notare riguarda il punto uno, quello che tratta due costrutti annidati di cui quello più interno è non terminante. Il primo punto è stato modificato e si presenta come segue:

```
for (k = 0; k<=10; k++){
    j++;
    for (i = 0; i==1; i=1)      //Warning
        i=0;
}
for (k = 0; k<=10; k++){
    j++;
    for (i = 0; i==1; i=1)
        i = 1;                  //Warning
}
```

Questa parte è stata sdoppiata introducendo una lieve modifica tra i due casi (l'assegnazione all'intero del ciclo più interno). L'analisi su questa parte di questa porzione di codice rivela due warning in due porzioni differenti del codice Java Bytecode . COSTA segnala infatti la prima istruzione che rende il ciclo non terminante, questa istruzione nel primo caso si trova nell'espressione di aggiornamento del for nel secondo caso è all'interno del corpo del for.

Sequenza di Do...While ha previsto l'analisi di un metodo che analizza i punti dei casi precedenti ma esattamente come per l'analisi di costrutti

semplici il corpo del while viene eseguito due volte. Il metodo che può essere espresso come una sequenza di quattro blocchi:

1. Due do... while annidati di cui il più interno non terminante
2. Due do... while annidati entrambi terminanti
3. Un do... while non terminante
4. Un do... while non terminante con al suo interno due cicli do... while uno terminante e uno no

```
public static int DWhileUse(int x){
    int i = 0;
    int j = 1;
    int k = 5;
    int p = 3;
    do{
[A]:    j ++;
        k++;
        i = 6;
        do{              //Punto 1
[B]:        i ++;
            i%=2;
        }while (i == 1);
    }while (k != 10);

    do {                 //Punto 2
        j *= 10;
        j %= 5;
        k--;
        do{
            j *= 10;
            j %= 5;
            p--;
        }while (p>10);
    }while (k >0);

        i = 6;
        do{              //Punto 3
[C]:    i ++;
        i%=2;
        }while (i == 1);

        p = 10;
        i = 7;
        do{              //Punto 4
            do{
[D]:            j *= 10;
                j %= 5;
                p--;
            }while (p>10);
            do{
[E]:            i ++;
                i%=5;
            }while (i == 3);
                i%=2;
        }while(i == 1);

        return j;
}
```

Si ricorda che i cicli do... while ottenuti con iterazione utilizzando costrutti che prevedono raffronti tra interi risultano tutti non terminanti.

L'analisi di questo metodo, restituisce terminazione sconosciuta, ma segnala cinque warning che possono essere ricondotti possono essere ricondotti ai punti indicati con le lettere all'intero del codice soprastante. Questo risultato ci mostra come COSTA segnali il warning sempre sulla prima istruzione del corpo del do... while. I due casi da segnalare riguardano il primo e l'ultimo punto. Nel primo punto si nota come la non terminazione del ciclo interno si estenda anche al ciclo esterno che viene giustamente riconosciuto anche esso come non terminante. Nel quarto punto si nota invece come la non terminazione colpisca non solo i due cicli non terminanti ma anche quello terminante (lettera D).

5 Conclusioni

5.1 Ricorsione contro iterazione

La sperimentazione ha sollevato alcune differenze tra l'analisi di algoritmi ricorsivi e l'analisi algoritmi iterativi.
I risultati mostrano, che un algoritmo espresso in modo ricorsivo viene riconosciuto molto più facilmente come terminante rispetto allo stesso algoritmo espresso in maniera iterativa. Questo fatto da attribuire al funzionamento di COSTA, che trasforma ogni algoritmo iterativo in ricorsivo prima di procedere all'analisi dello stesso. La trasformazione sicuramente introduce un ulteriore difficoltà alle operazioni di vera e propria analisi, e la sua implementazione non risulta perfetta.
Il punto precedente deponeva sicuramente a favore degli algoritmi ricorsivi, ma la situazione si ribalta osservando la segnalazione dei warning all'interno del codice. Nel caso di metodi la cui terminazione è sconosciuta appaiono warning in corrispondenza del problema, nel caso in cui COSTA riesca a scoprire a chi attribuirlo. Se il metodo in questione è ricorsivo l'eventuale warning farà riferimento all'istruzione successiva alla dichiarazione del metodo (molto spesso una dichiarazione di variabile), e non risulta quindi utile nella eventuale operazione di debugging. Nel caso di metodo iterativo, il warning verrà restituito in corrispondenza del ciclo che non ritiene terminante (esclusi casi precedentemente evidenziati nella parte dedicata alle sequenze di costrutti 4.8.2), facilitando ricerca e correzione di eventuali errori.

5.2 Analizzatore come ulteriore strumento di debug

I risultati dei test di terminazione provano come COSTA, possieda un intuito superiore a quello di un semplice analizzatore statico. Capita infatti che inserendo in un metodo parti di codice non terminante, che però non viene mai eseguito, esso venga ritenuto comunque terminante. Questa straordinaria capacità è offuscata dell'imprevedibilità dei risultati forniti. COSTA non riesce infatti a far valere questa sua dote su tutti i costrutti Java Bytecode e anche su codice estremamente simile i risultati ottenuti possono essere contrastanti. Malgrado si possa segnalare come le condizioni di if e

while che prevedono raffronti con zero e i metodi ricorsivi risultino più spesso dimostrati come terminanti non è possibile stabilire regole che fissino come ottenere un risultato terminante e quando no.
É importante sottolineare come ogni risultato fornito dall'analizzatore risulti corretto, ovvero non esistono casi in cui un metodo non terminante venga ritenuto terminante e i casi in cui una parte di codice terminante viene ritenuto non terminante avvengono solo nel caso altre parti di codice circostante siano non terminanti. Questo indica che la segnalazione di un warning prevede sempre la presenza di un errore, anche se questo non influisce sul funzionamento del metodo.
Ritengo importante segnalare, che i realizzatori di questo progetto non si sono interessati a sviluppare COSTA come strumento di supporto alle operazioni di debugging. Lo si può osservare guardando i risultati forniti in output dall'analizzatore. L'interfaccia web non permette di visualizzare il codice Java Bytecode analizzato e non restituisce nessun risultato che sia riconducibile al codice. Nel plugin di eclipse i warning che vengono segnalati fanno riferimento unicamente al codice Java Bytecode è quindi necessario di un altro plugin per visualizzare il Java Bytecode del metodo. La COSTA view prevista per eclipse è inoltre estremamente limitata dal momento che permette di visualizzare unicamente errori, warning e informazioni sui risultati ottenuti, mostrati sotto forma di tabella. I riferimenti presenti nella tabella si rifanno solo all'intero metodo a cui sono riferite, e non indicano nemmeno l'istruzione Java Bytecode a cui si rivolge, che permetterebbe di risalire alla porzione di codice corrispondente. Inoltre in nessun caso è possibile visualizzare risultati intermedi come ad esempio le *sistema di relazioni di costo* .

5.3 Complessità

I test effettuati sulla parte di complessità evidenziano alcuni punti a favore o contro l'utilizzo dell'analizzatore. Pro (nel caso vengano restituiti risultati coerenti):

- I risultati forniti risultano sempre in linea con la classe di complessità che rappresentano;

- I risultati forniti risultano estremamente precisi, e lievi modifiche al metodo vengono correttamente rilevate all'interno delle *espressione di costo* ;
- La crescente complessità computazionale di un algoritmo, non costituisce di per se, un ostacolo al calcolo della complessità;

Contro:

- In molti casi vengono restituiti risultati non completi dal momento che all'interno dell'*espressione di costo* si segnala che la massimizzazione di una parte di codice è fallita. Questo indica che è ancora presto per utilizzare questo strumento lontano da semplici fasi di test;
- L'analizzatore risulta poco efficace nel caso un metodi risulti contorto;

5.4 L'analizzatore e le librerie

Nella presentazione dell'analizzatore COSTA, si faceva riferimento alla sua capacità di analizzare codice in cui venivano utilizzate librerie, e si segnalava la sua capacità di calcolarne o tralasciarne la complessità che introducevano. Le sequenze di test effettuate, rivelano che questa prospettiva non risulti veritiera o per meglio dire non restituisce risultati che permettano un effettivo calcolo del numero di istruzioni necessarie all'esecuzione dell'algoritmo. Nel caso i risultati restituiti fossero esattamente quelli che gli sviluppatori del progetto volevano ottenere, essi possono essere utilizzati esclusivamente come raffronto tra metodi, anche se nel caso di risultati molto complessi anche questa operazione è resa estremamente difficile.

5.5 Parere complessivo sul progetto

Consci della complessità degli argomenti di cui il progetto tratta e quindi della complessità dell'analisi che COSTA è portato a compiere. Comprendendo quindi che alcuni errori sono pressoché inevitabili, si ritiene però che alcune parti del progetto siano meglio sviluppate di altre. In particolare la parte di progetto basata su PUBS sembra essere ad un ottimo stato di avanzamento. Al contrario la l'interfaccia tra il byte-code e PUBS (la parte che dal codice

ricava le *relazione di costo*) necessita di migliorie. Andrebbe migliorata la sua capacità di conversione da codice iterativo a ricorsivo.
Altri difetti riscontrati nell'analizzatore sono:

- La scarsa capacità di riscontrare warning all'interno di codice ricorsivo. Sarebbe opportuno provare a spostare la segnalazione dall'inizio del codice alla chiamata ricorsiva che lo provoca;
- L'incapacità di gestire interfacce ed extends, non permettendo quindi a chi utilizza l'analizzatore di sfruttare a pieno le caratteristiche di incapsulamento tipiche del linguaggio Java.

Riferimenti bibliografici

[1] Akeo Adachi, Takumi Kasai, and Etsuro Moriya. A theoretical study of the time analysis of programs. In *MFCS'79*, pages 201–207, 1979.

[2] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. Ramírez, G. Román, and D. Zanardini. Termination and cost analysis with costa and its user interfaces. *Electron. Notes Theor. Comput. Sci.*, 258:109–121, December 2009.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 157–172, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proceedings of the 15th international symposium on Static Analysis*, SAS '08, pages 221–237, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Cost relation systems: A language-independent target language for cost analysis. *Electron. Notes Theor. Comput. Sci.*, 248:31–46, August 2009.

[6] Ralph Benzinger. Automated higher-order complexity analysis. *Theor. Comput. Sci.*

[7] Saumya K. Debray and Nai-Wei Lin. Cost analysis of logic programs. *ACM Trans. Program. Lang. Syst.*, 15:826–875, November 1993.

[8] Daniel Le Métayer. Ace: an automatic complexity evaluator. *ACM Trans. Program. Lang. Syst.*, 10:248–266, April 1988.

[9] Jorge Navas, Edison Mera, Pedro López-García, and Manuel V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP'07*, pages 348–363, 2007.

[10] Mads Rosendahl. Automatic complexity analysis. In *Proceedings of the fourth international conference on Functional programming languages*

and computer architecture, FPCA '89, pages 144–156, New York, NY, USA, 1989. ACM.

[11] D. Sands. A naive time analysis and its theory of cost equivalence. *Journal of Logic and Computation, 5(4)*.

[12] P. Wadler. Strictness analysis aids time analysis. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 119–132, New York, NY, USA, 1988. ACM.

[13] Ben Wegbreit. Mechanical program analysis. *Commun. ACM*, 18:528–539, September 1975.

[14] Wikipedia. Mccarthy 91 function — wikipedia, the free encyclopedia, 2011. [Online; accessed 12-January-2012].