PREF, un algoritmo per il calcolo delle Preferred Extensions

M.Bono, P.Bertoni

19 gennaio 2014

Indice

Capitolo 1

Modellizzazione del problema

1.1 Strutture dati principali

Le strutture dati utilizzate dal programma sono basate sul progetto *SemOpt*. I singoli argomenti (implementati nella classe *Argument*) contengono, oltre a delle informazioni **uniche** del nodo stesso (come il *ID* ed il *numero*), anche i dati relativi alle interazioni che esso ha con gli altri argomenti suoi vicini: in particolare ogni argomento possiede due insiemi:

1. uno rappresentante gli argomenti attaccati dal nodo stesso (la lista *attacks*);
2. uno rappresentante gli argomenti che attaccano il particolare nodo (la lista *attackers*);

Per quanto concerne gli insiemi di argumenti, essi sono implementati nella classe *SetArguments*. In tale classe i singoli nodi dell'insieme sono immagazzinati in un albero di ricerca binario, implementato tramite la classe *std::map*[1]. All'interno dell'albero, tuttavia, non sono salvati i valori dei vari Argument, ma solo i relativi puntatori: in questo modo durante tutto l'arco di vita del programma, esisterà solo un'unica istanza rappresentante un particolare nodo. Inoltre è possibile aggiungere ed eliminare nodi dagli insiemi molto più velocemente in quanto è necessario aggiungere o eliminare solo dei puntatori, e non le intere strutture all'interno della classe Argument. Strutturando in tal modo argomenti ed insieme di argomenti, tuttavia, risulta più difficile considerare solo un sottoinsieme del

[1] la documentazione della struttura map è disponibile al link http://www.cplusplus.com/reference/map/map/

grafo iniziale: infatti, essendo le informazioni riguardanti gli attaccanti e gli attacchi memorizzate non nell'insieme in sé ma nei singoli argomenti, è possibile avere un *SetArguments* A, rappresentante per esempio un'intersezione tra due insiemi, in cui ci sono dei puntatori indicanti degli Argument non più contenuti nell'insieme stesso.
L'intero grafo, comprensivo di ogni argomento e di ogni attacco, è strutturato nella classe AF (Argument Framework): in tale classe è ovviamente contenuto un puntatore ad un *SetArguments* (rappresentante l'intero grafo). Oltre a ciò, la classe prevede altre funzioni, la più imporante delle quali è *readfile*, consentente di leggere un file rappresentante un grafo.
Per quanto concerne una *preferred extension*, essa è visibile come un insieme di *SetArguments*. Le uniche operazioni che devono essere eseguite su tale struttura dati sono:

- unione di due insiemi di *SetArguments*;
- aggiunta di particolari *Argument* in tutti i *SetArguments* all'interno dell'insieme.

Dato che le operazioni basilari non accedono alla struttura dati come in un dizionario ma piuttosto come in una lista si è pensato di implementare questo insieme di *SetArguments* utilizzando la classe *std::vector*: in questo modo l'esecuzione di alcune istruzioni su ciascun elemento dell'insieme di *SetArguments* è risultata più facile da realizzare. A livello implementativo, la classe che rappresenta un insieme di *SetArguments* è denominata *SetArgumentsVector*.

1.2 Strutture dati secondarie

Il programma necessita di un'altra struttura dati: infatti, come richiesto dall'algoritmo di Tarjan, è necessario decorare ogni nodo con 2 attributi interi. Per implementare ciò è stata definita la classe ExtendedArgument che rappresenta la decorazione stessa, ossia i 2 interi. Dato che in tale algoritmo ad ogni nodo vengono assegnati questi 2 numeri è necessario utilizzare una struttura dati che associ ad ogni argomento la relativa decorazione ExtendedArgument: tale associazione è resa possibile tramite una hashtable che associa ad ogni argomento (tramite il numero univoco posseduto da ciascun nodo) la relativa decorazione. È stata scelta una hashtable in quanto il tipo di accesso utilizzato per ottenere

le decorazioni non è sequenziale, bensì simile ad un dizionario; inoltre, il tempo di accesso costante della hashtable consente di velocizzare l'ottenimento dei valori interi richiesti dall'algoritmo. Per l'implementazione della hashtable è stata utilizzata la struttura unordered_map messa a disposizione da std [2] .

1.3 classi utilizzate

Il progetto definisce varie classi utilizzate per calcolare la soluzione. Mentre alcune di esse rappresentano concetti chiave utilizzati lungo tutto il programma, altre servono semplicemente come *wrapper*: esse infatti servono solo per isolare del particolare codice in modo da aumentare la leggibilità del progetto stesso. Altre classi, infine, sono delle classi di facciata (classi *facet*): il loro scopo è fornire una più semplice interfaccia per sfruttare le funzioni offerte dal programma, senza dover scrivere complessi prototipi o utilizzare lunghi pezzi di codice. Le classi definite all'interno del progetto sono le seguenti:

AF	classe del progetto SemOpt
Argument	classe del progetto SemOpt
Boundcond	wrapper per l'algoritmo boundcond
ExtendedArgument	decorazione richiesta dall'algoritmo Tarjan
Grounded	wrapper per l'algoritmo ground
Labelling	classe del progetto SemOpt
OrClause	classe del progetto SemOpt
PrefAlgorithm	classe facet per fornire una facile interfaccia dell'algoritmo
Preferred	classe del progetto SemOpt
SATFormulae	classe del progetto SemOpt
SCCSEQ	classe wrapper per l'algoritmo SCCSEQ
SetArguments	classe del progetto SemOpt
SetArgumentsVector	un vettore di insiemi di argomenti

[2] La documentazione relativa a unordered_map è disponibile al link http://www.cplusplus.com/reference/unordered_map/unordered_map/

1.4 estensioni delle classi originali in SemOpt

Alcune classi ereditate dal progetto SemOpt sono state modificate per fornire delle funzioni più avanzate rispetto a quelle base. Tali modifiche sono tuttavia solo delle aggiunte: per preservare ogni retrocompatibilità, non sono state modificate funzioni preesistenti né tanto meno sono stati eliminati metodi.

Una delle aggiunte più comuni che sono state eseguite è stata l'overloading dell'operatore "»" nelle classi in cui non fosse già presente: questa aggiunta ha permesso di semplificare il rilevamento dei vari bug incontrati durante lo sviluppo del programma.

Overloading dell'operatore "»" a parte, una delle classi in cui sono stati aggiunti il maggior numero di funzioni è stata la classe *SetArguments*: in essa sono stati aggiunti i metodi sotto elencati:

- *SetArguments(AF af, string names[], int length)*;
- *vector<SetArguments*> getSingletons()*;
- *SetArguments* getArgumentAttackersInSet(Argument*)*;
- *SetArguments* getArgumentAttacksInSet(Argument*)*;
- *SetArguments* getAllAttackers()*;
- *SetArguments* getAllAttacks()*;
- *SetArguments* getAllAttackersInSet()*;
- *SetArguments* getAllAttacksInSet()*;
- *SetArguments* getSubsetAttackedBy(SetArguments*)*;
- *SetArguments* getSubsetAttacks(SetArguments*)*;
- *SetArguments* getExternalAttackers(void)*;
- *SetArguments* getExternalAttacks(void)*;

- *void setRestrictDeprecated(SetArguments*, SetArguments*)*;
- *void setSafeRestrict(SetArguments*, SetArguments*)*;
- *void merge(SetArguments*, SetArguments *)*;
- *void remove_All_Arguments()*;
- *void clean()*;
- *bool equal(const SetArguments& other) const*;
- *bool operator!=(const SetArguments& other) const*;
- *void printNodeRelations(void)*;
- *void printGraph(void)*;
- *bool graphHasEdges(void)*.

Per i dettagli sul funzionamento delle funzioni (parametri, precondizioni e/o postcondizioni) è possibile visionare la documentazione del codice, allegata al progetto.

Un'ultima aggiunta che è stata fatta alle classi originarie del progetto SemOpt è l'introduzione di alcuni iteratori: essi, tuttavia, non offrono sensibili miglioramenti per quanto concerne le performance dell'algoritmo, ma servono soltanto per aumentare la leggibilità del codice, come per esempio durante dei cicli.

1.5 funzioni disponibili della classe SetArgumentsVector

Durante la codifica del progetto è stata realizzata anche una classe rappresentante un insieme di *SetArguments* (quindi un insieme di insiemi di *Argument*). Benché la classe contenga poche funzioni, essa è stata indispensabile per realizzare un codice flessibile e ben leggibile.
Il concetto di insiemi di insiemi di *Argument* è utile nell'algoritmo computante le Preferred Extensions: in esso si richiede di utilizzare l'operatore $\otimes$, operatore che elabora appunto un

insieme di insiemi di *Argument*. Per realizzare tale struttura dati, si è utilizzata la struttura *vector* disponibile nella libreria *std*: abbiamo scelto tale struttura in quanto l'accesso ai dati in essa contenuti è di tipo sequenziale; dato che anche all'interno dell'algoritmo un insieme di insiemi di Arguments viene scandagliato elemento per elemento l'uso la classe *vector* è sembrato appropriato.

All'interno della classe *SetArgumentsVector* sono individuabili alcuni metodi, di seguito riportate:

- *static void swap(SetArgumentsVector& first, SetArgumentsVector& second)*;
- *SetArgumentsVector()*;
- *SetArgumentsVector(vector<SetArguments*>& _list)*;
- *virtual ~SetArgumentsVector()*;
- *void clone(SetArgumentsVector* list)*;
- *SetArguments* at(unsigned int i) const*;
- *void push_back(SetArguments* lab)*;
- *void clearAll()*;
- *unsigned int size() const*;
- *SetArgumentsVectorIterator begin() const*;
- *SetArgumentsVectorIterator end() const*;
- *SetArgumentsVector& operator=(SetArgumentsVector rhs)*;
- *bool operator==(const SetArgumentsVector &other) const*;
- *bool operator!=(const SetArgumentsVector &other) const*;
- *bool exist(const SetArguments* lab) const*;
- *void addSetArgumentsToAll(const SetArguments* set)*;

- *bool empty()*;
- *void pop_back()*;
- *void printLabellingList()*;

Per i dettagli sul funzionamento delle funzioni (parametri, precondizioni e/o postcondizioni) è possibile visionare la documentazione del codice, allegata al progetto.

Oltre ai già accennati metodi, la classe *SetArgumentsVector* dispone anche di un overloading sull'operatore "+": tale overloading serve per unire 2 SetArgumentsVector; si è preferito effettuare un overloading di un'operatore invece di creare un metodo separato per aumentare la leggibilità del codice.

Capitolo 2

Pseudocodice del programma

Analizziamo ora gli algoritmi utilizzati per realizzare il progetto.

```
Input: Γ, C
Data: S, E*, e, O, I, E'_P
Result: E_P
(e, I) ← Grounded(Γ, C)
E_P ← {e}
if I.isEmpty then
 |  return Ep
end
Γ ← Γ ↓ I
S ← SCCSSEQ(Γ)
foreach S_i ∈ S do
 |  E'_P ← φ
 |  foreach e ∈ E_P do
 |  |  (O, I) ← boundcond(Γ, S_i, e)
 |  |  if O.isempty then
 |  |  |  if I.isempty then
 |  |  |  |  E* ← φ
 |  |  |  end
 |  |  |  else
 |  |  |  |  E* ← SATPref(Γ ↓ S_i, I ∩ C)
 |  |  |  end
 |  |  end
 |  |  else
 |  |  |  E* ← Pref(Γ ↓ (S_i\O), I ∩ C)
 |  |  end
 |  |  E'_P ← E'_P ∪ (e ⊗ E*)
 |  end
 |  E_P ← E'_P
end
return E_P
```

Algorithm 1: Algoritmo Pref.

Input: Γ, C
Data: N, ANC, ANI
Result: e, I
$e \leftarrow \emptyset$
$I \leftarrow \Gamma.A$
`/* N sono tutti e soli i nodi in C non attaccati da nodi in I */`
while $N \neq \emptyset$ **do**
  $e \leftarrow e \cup N$
  $ANC \leftarrow$sottoinsieme di C contenente tutti i nodi attaccati dai nodi in N
  $ANI \leftarrow$sottoinsieme di I contenente tutti i nodi attaccati dai nodi in N
  $C \leftarrow C \backslash (N \cup ANC)$
  $I \leftarrow I \backslash (N \cup ANI)$
end
return (e, I)

Algorithm 2: Algoritmo Grounded.

Input: Γ, S_i, e
Result: O, I
$O \leftarrow \{n |\ n \in S_i, \quad n \text{ is attacked by } m \in e\}$
$temp \leftarrow \{n |\ n \in \Gamma \setminus (S_i \cup e), \quad n \text{ is attacked by } m \in e\}$
$I \leftarrow \{n |\ n \in S_i \setminus O,$

$$(\nexists n \in I : n \text{ is attacked by } m \in \Gamma \setminus S_i) \ \oplus$$
$$(\nexists n \in I : n \text{ is attacked by } m \notin temp)$$

return (O, I)

Algorithm 3: Algoritmo Boundcond.

Input: $G = (V, E)$
Result: insieme di componenti fortemente connesse
index $\leftarrow 0$
Stack $\leftarrow \emptyset$
foreach $v \in V$ **do**
  if *v.index = undefined* **then**
    strongconnect(v);
  end
end
return Stack

Algorithm 4: Algoritmo di Tarjan

```
Input: un singolo nodo
v.index ← index
v.lowlink ← index
index ← index + 1
S.push(v)
//prendo i successor di v
foreach ((v, w) ∈ E) do
    if w.index = undefined then
        // il successore w non e' ancora stato visitato
        strongconnect(w)
        v.lowlink ← min(v.lowlink, w.lowlink)
    end
    else
        if w ∈ S then
            // il successore w si trova nello stack e quindi anche nel SCC corrente
            v.lowlink ← min(v.lowlink, w.index)
        end
    end
end
// se v si configura come un nodo radice, effettua un pop dallo stack e genera un
nuovo SCC
if (v.lowlink = v.index) then
    newSCC ← createSCC()
    repeat
        w ← Stack.pop()
        newSCC.add(w)
    until (w = v);
    SCClist.add(newSCC)
end
```

Algorithm 5: Funzione strongconnect utilizzata dall'algoritmo di Tarjan

```
Input: SCCset
Result: {V, E}
foreach (nodei ∈ SCCset) do
    i_attacks_j ← false
    while (nodej ≠ nodei ∧ i_attacks_j = false) do
        if nodei.getSubsetAttackedBy(nodej)← ∅ then
            addEdge({nodej → nodei}, E)
            i_attacks_j ← true
        end
        nodej ← nextNode(SCCset)
    end
end
return G={SCCset, E}
```

Algorithm 6: Algoritmo linkMacroGraph utilizzato per creare un grafo partendo dai singoli SCC

```
Input: {V,E}
Result: {V, E}
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while (S ≠ ∅) do
    n ← pop(S)
    addToTail(n, L)
    foreach node m with an edge e = (n, m) do
        removeEdge(e, E) if (m has no other incoming edges) then
            push(m, S)
        end
    end
end
if (E ≠ ∅) then
    return error //il grafo ha almeno un ciclo
end
else
    return L //una lista di nodi ordinati in modo topologico
end
```

Algorithm 7: Algoritmo di Kahn utilizzato per ordinare topologicamente i nodi

Capitolo 3

Ottimizzazione del codice

3.1 Miglioramenti suggeriti

È stata implementata una serie di regole utili ad evitare delle chiamate alle procedure "pesanti" usate dall'algoritmo *Pref*, che sono qui elencate. Il codice qui mostrato è, per comodità, espresso in pseudocodice e non in standard C_{++}.

3.1.1 Sulla chiamata a Grounded e Pref

La regola espressa informalmente è: *la chiamata a una di queste due procedure produce risultati noti a priori quando il primo argomento è uguale al secondo e coincide con un grafo di un solo nodo. In questo caso, l'insieme restituito è composto da un singoletto contenente quell'unico nodo.*

```
if Γ = C ∩ Γ.cardinality= 1 then
 |  e ← Γ
 |  I ← ∅
end
else
 |  Grounded(Γ, C, e, I)
end
```

Algorithm 8: Miglioramento nella gestione della chiamata a Grounded.

$A \leftarrow \Gamma \downarrow S_i \backslash O$
$B \leftarrow I \cap C$
if $A = B \ \cap \ A$.cardinality$= 1$ **then**
| $E^* \leftarrow \{A\}$
end
else
| $E^* \leftarrow$Pref(A, B)
end

Algorithm 9: Miglioramento nella gestione della chiamata a Pref.

3.1.2 Sulla chiamata a PrefSAT

La regola espressa informalmente è: *la chiamata alla procedura produce risultati noti a priori quando il primo argomento è uguale al secondo e coincide con un componente fortemente connesso (*SCC*) di cardinalità 1 o 2. Nel primo caso, l'insieme restituito è composto da un singoletto contenente proprio l'*SCC*, nel secondo caso è composto da due singoletti, contenenti ciascuno uno dei due nodi dell'*SCC.

Nel testing però la prima parte di questa regola non si è rivelata coerente con l'output di *PrefSAT* e quindi generalmente ha prodotto risultati non corretti. Si è notato in particolare che talvolta *PrefSAT* veramente si comporta come qui specificato, talvolta restituisce un insieme vuoto. Prudenzialmente quindi si è ristretta la miglioria al solo caso ove la cardinalità è 2. Per giustificare quanto detto, si riportano qui gli output ottenuti a questo proposito, in due dei tre esempi forniti nel documento delle specifiche. Si spera che questo aiuti chi è più esperto a identificare casi degeneri della regola espressa.

- esempio **1**: *PrefSAT* viene chiamato due volte in totale.
  $\Gamma \downarrow S_i = I \cap C = \{n9\}$
  $S_i = \{n9\}$
  $e = \{n1\ n3\ n6\ n13\}$
  $E^*_{suggested} = \{\{n9\}\}$
  $E^*_{returned} = \{\{\}\}$
  il medesimo risultato si ottiene nella seconda chiamata, ove cambia solo $e = \{n1\ n3\ n6\ n14\}$.
  In definitiva *Pref* restituisce $\{\{n1\ n3\ n6\ n9\ n13\}\ \ \{n1\ n3\ n6\ n9\ n14\}\}$ invece che $\{\{n1\ n3\ n6\ n13\}\ \ \{n1\ n3\ n6\ n14\}\}$, come previsto dall'oracolo.

- esempio **3**: *PrefSAT* viene chiamato una volta in totale.
  $\Gamma \downarrow S_i = I \cap C = \{n3\}$
  $S_i = \{n3\}$
  $e = \{n1\ n4\}$
  $E^*_{suggested} = \{\{n3\}\}$
  $E^*_{returned} = \{\{n3\}\}$
  Pref restituisce $\{\{n1\ n2\ n3\ n4\}\}$ come previsto.

Nel codice si troveranno istruzioni leggermente diverse dalle seguenti, ma chiaramente equivalenti, per sfruttare meglio caching e lazy evaluation.

```
A ← Γ ↓ S_i
B ← I ∩ C
if A = B ∩ S_i.cardinality= 2 then
 | {S'_i, S''_i} ← S_i.reduceSingletons()
 | E* ← {S'_i, S''_i}
end
else
 | E* ←PrefSAT(A, B)
end
```

Algorithm 10: Miglioramento nella gestione della chiamata a PrefSAT.

3.1.3 Sulla chiamata a Boundcond

La regola espressa informalmente è: *sia Pref*(Γ, C)*, ove* $\Gamma = C$*. La chiamata a Boundcond per ogni SCC senza alcun SCC padre, ovvero non attaccato da altri SCC, restituirà necessariamente come* O *un insieme vuoto e come* I *l'SCC stesso, per qualunque* e*.*

```
if Γ = C ∩ S_i.fathers= ∅ then
 | O ← ∅
 | I ← S_i
end
else
 | (O, I) ←Boundcond(Γ, S_i, e)
end
```

Algorithm 11: Miglioramento nella gestione della chiamata a Boundcond.

3.2 Miglioramenti personali

Per migliorare ulteriormente le prestazioni, è stato eseguito un miglioramento durante la verifica di esistenza di un SetArguments all'interno di un SetArgumentsVector nella funzione *exist*. Normalmente, infatti, per capire se un insieme di nodi è presente in un'istanza di SetArgumentsVector bisognerebbe, per ciascun SetArguments contenuto, evocare la funzione *equal*, la quale verifica se gli insiemi contengono gli stessi argomenti. Essendo tale procedura un processo lungo, tale algoritmo può penalizzare inutilmente le prestazioni.
Per evitare ciò, durante la fase di ricerca si controllano dapprima i singoli puntatori dei SetArguments contenuti nel vettore per verificare il caso in cui il puntatore del SetArguments cercato coincida con uno di quelli immagazzinati: se la ricerca ha successo, non ci sarà bisogno di controllare i singoli nodi contenuti negli insiemi in quanto i puntatori coincidono. Se invece non si riesce a trovare il puntatore cercato non è detto che il SetArguments da trovare non si trovi nell'istanza di SetArgumentsVector: potrebbe darsi che sia immagazzinata solo una sua copia fisica; per questo si analizzano i SetArguments contenuti tramite la funzione *equals*.
Un secondo piccolo miglioramento riguarda l'overloading dell'operatore = per la classe SetArgumentsVector. Nell'algoritmo principale contenuto nella classe PrefAlgorithm, infatti, è stata codificata la seguente riga di codice (avente sia *Ep_ new* che *E_ star* di tipo *SetArgumentsVector*):

```
Ep_new = Ep_new + E_star;
```

All'interno del codice, l'aggiornamento di *Ep_ new* avviene effettuando uno swap fisico di memoria tra la struttura dati *Ep_ new* e quella derivante dalla somma: in questo modo viene evitato l'aggiornamento *SetArguments* per *Setarguments* tra la vecchia e la nuova struttura dati *Ep_ new*; al suo posto viene eseguita una sostituzione del blocco di memoria contenente la nuova struttura dati.

3.2.1 Gestione dei padri di S_i

Si definisce $\alpha_i \triangleq \bigcup \beta_i$, dove β_i è un SCC S_j che attacca S_i. Formalmente dunque α_i è un unione di *SetArguments* e quindi un *SetArguments* a sua volta, ma se si osserva che $S_i \cap S_j \ \forall i \neq j$ (ovvero gli SCC sono insiemi disgiunti di *Arguments*), è possibile

considerare α_i come un semplice insieme generico. Per semplicità è stato quindi considerato un **std::vector**.

Capitolo 4

Analisi computazionale

4.1 Individuazione dei blocchi procedurali d'interesse

La complessità temporale di **Pref** dipende fortemente, oltre che dalle dimensioni del problema, dal numero di chiamate ai seguenti algoritmi:

- *Grounded*
- *SCCSEQ*
- *Boundcond*
- *PrefSAT*
- *Pref* stesso.

L'analisi quindi è stata strutturata per fornire due indici:

- η - una misura del tempo "*tic-toc*" per ciascuna chiamata a una delle suddette procedure, più il tempo totale di esecuzione del programma;
- ξ - il numero di chiamate a ciascuna procedura, nell'arco dell'intera esecuzione.

Mentre il primo indice è atto a relazionare la dimensione del problema al tempo in sè ed è fortemente mediato dalle prestazioni della macchina di esecuzione, il secondo è una misura delle differenze tra la versione *base* di **Pref** e la versione *improved*, implementazione dello studio presentato nel capitolo 3. Chiamando il programma sullo stesso problema nelle

due modalità, infatti, è rapidamente visibile il risparmio in termini di chiamate introdotto dalle ottimizzazioni. Si noti che *SCCSEQ* non è stato introdotto nell'indice ξ siccome non è sottoposto ad ottimizzazioni, pertanto il numero delle sue chiamate è sempre pari a quello relativo a *Pref*.

Entrambi gli indici sono sempre restituiti dal programma sullo standard output.

4.2 Verifica di correttezza: gli oracoli

Per testare innanzitutto i requisiti funzionali ci si è avvalsi dell'oracolo *ASPARTIX* disponibile come web service dall'*URL* `http://rull.dbai.tuwien.ac.at:8080/ASPARTIX`. L'applicazione oracolo, oltre a confermare gli output dei tre esempi forniti insieme alle specifiche, ha validato anche altri test, generati secondo il metodo descritto in 4.5.

4.3 Metodo di misura del tempo

La scelta è ricaduta sulla libreria **chrono** del C_{++}, che costituisce anche un *subnamespace std::chrono*. I concetti principali implementati sono tre:

- le *durate temporali* misurano lassi di tempo, fino all'ordine dei millisecondi. In questa libreria sono rappresentate come oggetti della classe template *duration*, che accoppiano una rappresentazione di conto e una precisione di periodo (*e.g.*, 10 millisecondi ha 10 come rappresentazione di conto e 'millisecondi' come precisione di periodo.)
- gli *istanti temporali* sono riferimenti a dei punti precisi sull'asse temporale, espressi come una *duration* da un punto fissato per convenzione.
- un *clock* è un framework collegante un *istante temporale* a un'istante del mondo reale.

La libreria offre almeno tre clocks diversi per esprimere il tempo corrente come un valore numerico: *system_ clock*, *steady_ clock* e *high_ resolution_ clock*. Data l'alta precisione di high_resolution_clock, tale clock sarà utilizzato nelle benchmark [1].

[1] Si noti che per utilizzare **chrono** è obbligatoria una compilazione conforme allo standard $C_{++}11$.

4.4 Specifiche tecniche del calcolatore utilizzato

Le benchmarks sono state eseguite su un notebook così caratterizzato:

- **OS** Linux Ubuntu 10.04 con istruzioni a 32 bit
- **CPU** Intel Pentium(R) Dual-Core CPU T4200 , clock 2.00GHz e tecnologia *Hyper Threading*
- **RAM** 4GB DDR3, di cui 3 disponibili all'utente.
- **Compiler** Gnu C Compiler (gcc) in versione $C^{++}11$ e ottimizzato.@@@ specificare la versione

4.5 Un generatore di problemi

Le tre istanze di prova allegate alle specifiche del problema vengono risolte in un tempo dell'ordine dei 10 millisecondi, con una varianza elevata, e quindi non sono utilizzabili per una profilazione consistente della complessità temporale. Per generare grafi più onerosi è stato implementato un piccolo applicativo indipendente, che libera l'analista dallo sforzo di pensare e codificare argumentation framework anche di migliaia di nodi o archi. I parametri richiesti sono:

- numero N di *Arguments*, ovvero nodi del grafo. Verranno automaticamente denominati n_i, $i \in [1, N]$.
- il numero massimo $\overline{A}$ di *Arguments* che attaccano o sono attaccati da ogni *Argument*. Ricordando che un grafo orientato di N nodi completamente magliato è dotato di $N(N-1)$ archi, il valore $\frac{N(N-1)}{N} = N - 1$ costituisce un upperbound per $\overline{A}$. Un grafo denso "a metà" è chiaramente caratterizzato da un $\overline{A} = \frac{N}{2}$.
- nome del file in formato *dl* dove il *framework* verrà salvato.
- $\{graph, dot, nograph\}$ per indicare se si desidera anche un output grafico in formato jpg, rappresentante il grafo. Se l'opzione è settata su "graph" verrà generata un

immagine tramite il programma esterno "dot". Nel caso "dot" il programma genererà soltanto un file compilabile con il programma "dot" mentre se nello scenario in "nograph" sia scelto il programma non genererà alcuna immagine.

Applicativo **graphGenerator** e relativo sorgente vengono allegati al progetto finale. Vengono inoltre forniti degli esempi casuali generati con esso, utilizzati nei test che saranno presentati a breve. I loro nomi rispettano la sintassi $N\_\overline{A}trial.dl$, dove trial è una lettera $a, b, \ldots n$ che distingua i problemi di uguale dimensione generati. È stato deciso inizialmente di esplorare gli ordini di grandezza di N da 10 a 10^4, ma, come si vedrà, già il passaggio da 10^2 a 10^3 si mostrerà proibitivo.

Se si suppone una complessità $O(1)$ per elaborare ciascun nodo, esclusa dalla chiamata la gravosa elaborazione dell'immagine, la complessità di graphGenerator è chiaramente $O(N\overline{A})$. Lo spazio richiesto per la memorizzazione dei grafi si comporta anch'esso linearmente rispetto alle singole dimensioni $N, \overline{A}$, come suggerito in figura 4.1.

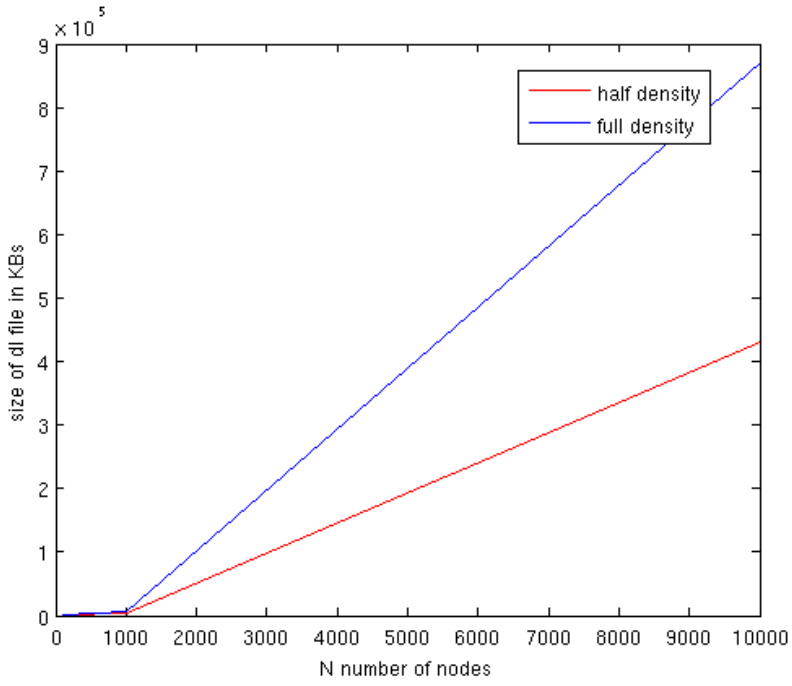

Figura 4.1: Dimensione su disco delle istanze del problema, al variare di N (ascisse) e con due $\overline{A}$ fissate relativamente a N.

4.6 Benchmarks sui test generati

Note: un trial contrassegnato dal "+" rappresenta l'esecuzione migliorata su quel grafo. Nella parte destra della tabella, per non appesantire la lettura sono stati omessi i valori invariati tra ciascuna coppia di esecuzioni. I tempi sono espressi in millisecondi.

4.7 Considerazioni finali

In tabella sono stati riportati solo i tempi totali e non alle singole chiamate, ma si registra che il tempo utilizzato da *PrefSAT* domina fino al 90% della durata totale di ogni esecuzione. È altresì la più onerosa in termini di memoria principale allocata: tale consumo di memoria è anchhe causato da alcun memory leak presenti nel codice fornito. Si può dunque affermare che l'ottimizzazione più rilevante di ogni altra sia quella che eviti la chiamata al problema SAT, peraltro notorialmente *NP-hard*.

Moltissime volte, mentre *Pref* gira in versione migliorata, viene risparmiata la chiamata a *Boundcond* poichè il relativo *SCC* non ha nessun padre. Questo è possibile che sia legato alla generazione puramente pseudocasuale dei grafi, che non cerca di costruire nessun percorso arbitrariamente lungo sui nodi, ma si limita a computare un certo numero di probabilità di collegamento. Versioni migliorate della struttura di test potrebbero incorporare un'euristica della "connessione" del grafo generato; il trascurabile vantaggio del miglioramento su *Boundcond* non ne giustifica per ora lo sforzo.

Si noti inoltre la grande differenza nel numero di chiamate, tra tre istanze note come *esempio1*, *esempio2* e *esempio3*, e invece i grafi generati casualmente. È evidente che argumentation framework strutturati senza una logica apparente rimangono alquanto sconnessi e, oltre a non avere interesse reale nella loro risoluzione, ne hanno ben poco dal punto di vista dell'indice ξ. Qualora fossero disponibili istanze di grafi di dimensioni considerevoli e ben strutturati, saremo lieti di incorporarli in una seconda e più informativa versione del testing.

Parimenti si può dire che le istanze generate casualmente, dato il numero sempre minimo di chiamate alle varie procedure, costituiscono una sorta di *lower bound* sulle prestazioni di *Pref* in generale.

N	$\overline{A}$	*trial*	*total time*	*#Grounded*	*#Boundcond*	*#PrefSAT*	*#Pref*
esempio		1	4	1	5	5	1
		1^+			2	4	
		2	1	1	1	1	1
		2^+			0	0	
		3	3	1	3	3	1
		3^+				2	
50	25	a	100	1	1	1	1
		a^+	95				
		b					
		b^+					
		c					
		c^+					
50	49	a	63000	1	1	1	1
		a^+					
		b					
		b^+					
		c					
		c^+					
100	50	a	900	1	1	1	1
		a^+	850		0		
		b	1150	1	1	1	1
		b^+	1160		0		
		c	550	1	1	1	1
		c^+	560		0		
100	99	a	6000	1	1	1	1
		a^+	6000		0		
		b	4150	1	1	1	1
		b^+	4150		0		
		c	4700	1	1	1	1
		c^+	4700		0		
200	100	a	63000	1	1	1	1
		a^+	60000		0		
		b	34000	1	1	1	1
		b^+	33000		0		
		c	17900	1	1	1	1
		c^+	17800		0		

Capitolo 5

Varie ed eventuali

5.1 Documentazione del codice

La documentazione del codice *SemOPT* era pressochè assente ed è stata largamente migliorata, basandosi sullo studio del codice. Sia per quella che per quella relativa a *Pref* è stato utilizzato **Doxygen** per generare del codice HTML navigabile, e il linguaggio **dot** per aggiungervi dei grafi. Si rimanda all'html allegato alla Documentazione, ed eventualmente al codice sorgente.

5.2 Testing del codice

Per individuare il maggior numero di bug presenti nel codice, è stata realizzata una suite di testing per verificare la correttezza delle funzioni principali del programma. La scelta circa il programma di testing si è basata su 4 fattori:

1. compatibilità con C++;
2. facilità d'uso;
3. facilità di installazione;
4. presenza di un plugin in Eclipse per l'utilizzo.

Da questi criteri si è infine deciso di utilizzare il set di funzioni CUTE [1]. Per testare nel migliore dei modi il codice del progetto, sono stati realizzati 37 test totali, divisi in 2 categorie:

1. test sulla correttezza delle funzioni: sono state verificate le funzioni da noi sviluppate, come per esempio la funzione *addSetArgumentsToAll* oppure la funzione *getAllAttackers*;

2. test sulla correttezza del valore delle preferred extension: per verificare la correttezza dei risultati del progetto, è stato confrontato il valore di ritorno del programma con il valore computato dall'oracolo.

La suite di test è, insieme alla documentazione, allegata al progetto.

[1] Le istruzioni di utilizzo, di downlaod ed il plugin di Eclipse sono disponibili all'URL http://cute-test.com/