Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Implementazione di un algoritmo lineare per l'α-equivalenza di λ-termini con sharing

Relatore: Chiar.mo Prof. Claudio Sacerdoti Coen

Presentata da: Emanuele Sinagra

Sessione terza
Anno Accademico 2016/2017

Alla mia famiglia, che ha creduto in me,
A Rodolfo, compagno di studi e amico.

Indice

Section 1 – Introduzione 4

Section 2 – Prima Implementazione 7

2.1 Definizione di TermGraph 8
2.2 Rappresentazione di un TermGraph 13
2.3 Problema di cancellazione di un arco indiretto 20
2.4 Codice sorgente 22

Section 3 – Test e Risultati 25

3.1 Generazione di tutti i TermGraph 26
3.2 Verifica funzionale 31
3.3 Efficienza dell’algoritmo 32
3.4 Problemi aperti 34

Section 4 – Conclusioni 36

Indice delle Appendici

A – Bibliografia 39

B – Ringraziamenti 41

Section 1 – Introduzione

L'attività di ricerca, oggetto della tesi, nasce dallo studio del Prof. Claudio Sacerdoti Coen e del Dott. Ricercatore Beniamino Accattoli sul costo computazionale delle lambda-riduzioni.

L'argomento di studio interessa i lambda-termini del lambda-calcolo, il quale rappresenta il prototipo di ogni linguaggio di programmazione funzionale. Poiché i lambda-termini comprendono le lambda-astrazioni, che legano il nome del parametro delle funzioni nel loro corpo, i lambda-termini sono normalmente pensati come il quoziente del tipo di dato sintattico rispetto alla congruenza chiamata alpha-conversione, la quale identifica termini a meno dei nomi delle variabili legate.

Studi recenti sul lambda-calcolo mostrano la possibilità di implementare in tempo lineare la riduzione dei lambda-termini sul numero di passi di beta-riduzione, che, nella loro usuale formulazione su lambda-termini, sono operazioni non costanti in tempo. Tali implementazioni rappresentano i lambda-termini per mezzo di un particolare tipo di grafo diretto riducibile, che prende il nome di TermGraph.

L'algoritmo oggetto di studio, risolve in tempo lineare sul numero di passi di beta-riduzione il problema della lambda riduzione dei lambda-termini, detta anche alpha-conversione.
Dato in input un TermGraph, composto da due Termgraph aventi le radici tra loro collegate da un arco indiretto, l'algoritmo verifica che i due TermGraph siano bisimili.
Il problema della bisimulazione tra TermGraph pone il seguente quesito "Uno dei due TermGraph può ricalcare il comportamento dell'altro?"
Segue quindi che i TermGraph sono alpha-convertibili sse i due grafi sono Bisimili.
L'algoritmo è di tipo decisionale e risponde si/no, per farlo modifica il dato in input nel TermGraph di Bisimulazione minimo, se esiste.

Il problema della Bisimulazione su grafi è dimostrato essere NP-Completo, quindi risolubile in tempo polinomiale da una macchina di Turing non deterministica, mentre l'istanza del problema su TermGraph non è NP-Completo, perché risolubile in tempo lineare.

Il lavoro oggetto di tesi consiste nella prima implementazione dell'algoritmo e nell’esecuzione dei relativi test funzionali e di efficienza dell'algoritmo.

Per questioni di comodità il linguaggio implementativo scelto è C#, un linguaggio compilato di medio livello che non inserisce particolari overhead.
Tuttavia, sviluppi futuri potrebbero prevedere l’implementazione in C senza complicazioni.

Section 2 – Prima Implementazione

Nel presente capitolo è descritta l'attività svolta per l'implementazione dell'algoritmo.

L'attività di stesura della tesi inizia con la definizione di TermGraph su cui opera l'algoritmo (2.1), dopo aver dato tale definizione, sono state implementate le strutture dati per la rappresentazione di un TermGraph in memoria (2.2).
Infine è iniziata l'attività di implementazione dell'algoritmo (2.4), e nel paragrafo 2.3 viene descritto il problema della cancellazione di archi indiretti e la soluzione scelta.

2.1 Definizione di TermGraph

Nella presente sezione è fornita la definizione di un particolare tipo grafo diretto riducibile detto TermGraph, utilizzato per la rappresentazione dei lambda-termini.

Un TermGraph è un grafo diretto riducibile, così definito:

- Ogni nodo ha un'etichetta tra Var, App, Lam; ognuna delle quali rispecchia rispettivamente variabili, applicazioni e lambda-astrazioni.
- Un nodo del tipo Var (Figura 1), che indicheremo senza carattere, non ha figli e se è figlio sinistro di un nodo Lam ha un arco diretto al nodo padre (detto legame).
- Un nodo del tipo App (Figura 2), che indicheremo con il carattere '@', ha due nodi figli, sinistro e destro, indicati con App(sx, dx).
- Un nodo del tipo Lam (Figura 3), che indicheremo con il carattere 'l', ha due nodi figli, e il nodo sinistro deve essere del tipo Var: Lam(sx, dx)
- Il grafo è riducibile, ovvero, una volta rimossi gli archi diretti dai nodi Var ai nodi Lam, il grafo ottenuto è aciclico.
- Ogni percorso, che parte da un nodo del tipo Var, che ha un arco legame e che raggiunge una radice del grafo, passa attraverso il nodo Lam a cui il nodo Var è legato, questa proprietà prende il nome di dominanza.

La relazione tra TermGraph e lambda-termini è la seguente:

- Un nodo di tipo App rappresenta un operazione binaria, e i figli rappresentano gli operandi
- Un nodo di tipo Lam rappresenta una lambda-astrazione, in cui:
    - La variabile ad esso legata rappresenta una variabile locale visibile solo all'interno del lambda scope.
    - Il nodo destro rappresenta il corpo della lambda-astrazione e può utilizzare tutte le variabili locali presenti nello scope, più le variabili globali.
- Un nodo di tipo Var senza legami rappresenta una variabile globale della lambda astrazione.

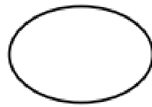

Figura 1 - Var

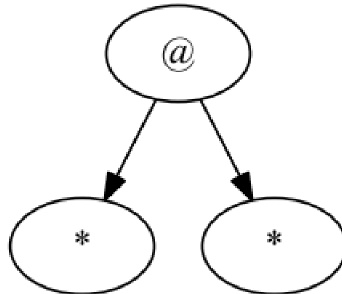

Figura 2 - App

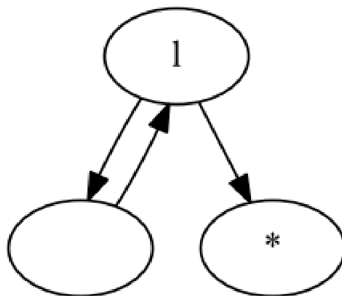

Figura 3 - Lam

Prima di illustrare un esempio di relazione tra un lambda-termine e i suoi TermGraph associati è necessario introdurre qualche definizione:

- Un nodo è **deSharabile** se è del tipo Lam o App ed ha almeno due archi entranti.
- Su un nodo deSharabile è possibile effettuare il suo deSharing, per uno o più dei suoi padri, duplicando il nodo e la sua struttura sottostante e facendo puntare uno dei due archi alla nuova struttura duplicata.
- Un TermGraph è **massimale** se non presenta nodi deSharabili.
- Un TermGraph è **minimale** se presenta la quantità massima di nodi deSharabili.

Partendo dal TermGraph minimale è possibile ricavare il TermGraph massimale deSharando tutti i nodi deSharabili su tutti i loro padri.

Un lambda termine può essere rappresentato da più TermGraph, come nell'esempio proposto in seguito.

Dato il lambda termine

$$[\lambda x. (x \text{ op } y) \text{ op } (x \text{ op } y)] \text{ op } [\lambda x. (x \text{ op } y) \text{ op } (x \text{ op } y)]$$

Dove op rappresenta un'operazione binaria definibile.

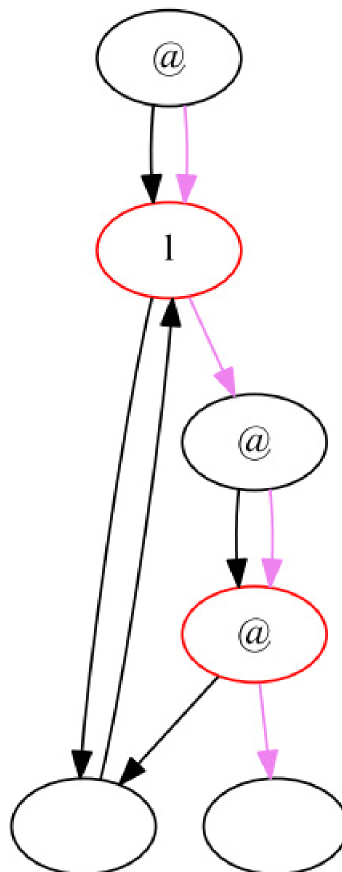

Figura 4 - TermGraph minimale

E' facile vedere che in Figura 4 gli unici nodi deSharabili sono quelli con bordo rosso.
L'arco sinistro ha colore nero, mentre l'arco destro ha colore rosa.

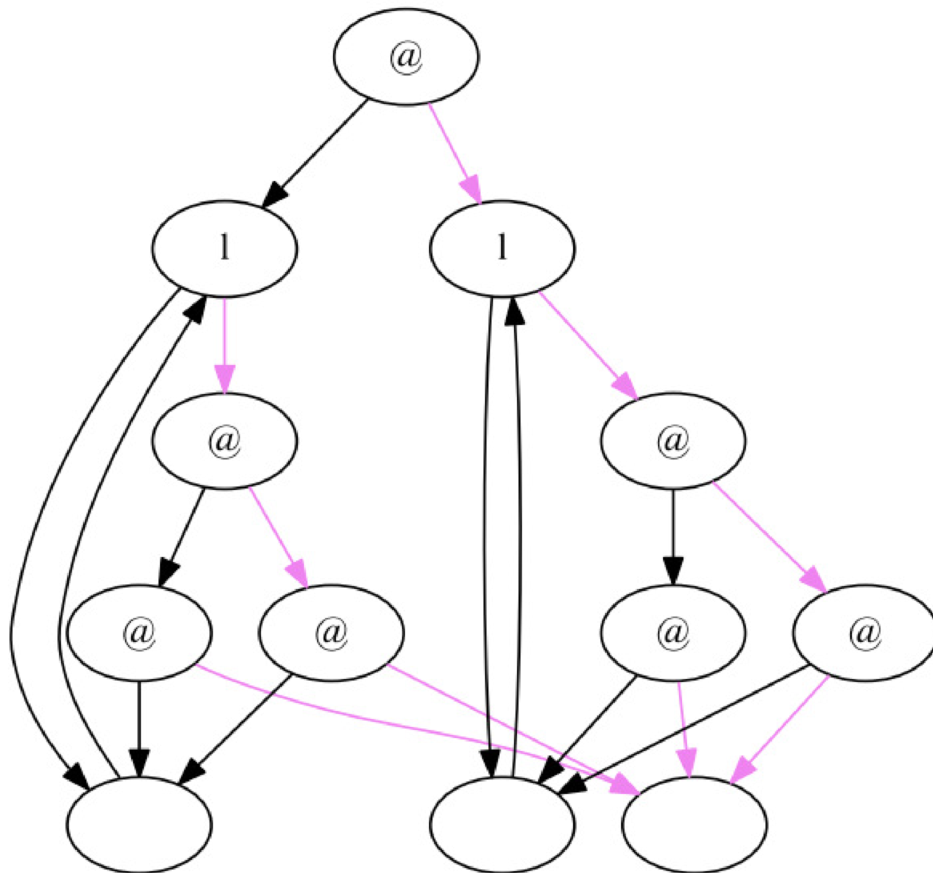

Figura 5 - TermGraph massimale

E' facile vedere che in Figura 5 non sono presenti nodi deSharabili.

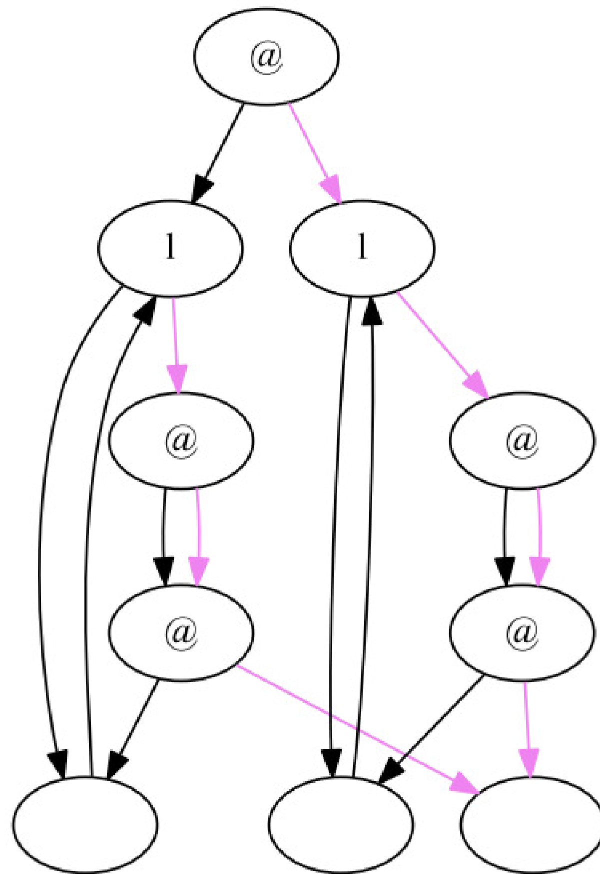

Figura 6 - Desharing parziale

E' possibile inoltre effettuare un deSharing parziale dei nodi del TermGraph, deSharando come nell'esempio, solo il nodo del tipo lambda.

I TermGraph in Figura 4, Figura 5 e Figura 6 sono quindi tutti equivalenti al lambda termine in esame.

2.2 Rappresentazione di un TermGraph

Nella presente sezione sono descritte le strutture necessarie per la rappresentazione del TermGraph su cui lavorerà l'algoritmo.

Prima di definire un TermGraph (Figura 10), è necessario definire la classe TermGraphNode (Figura 7), rappresentante di un nodo del TermGraph, la classe EnrichedTermGraphNode (Figura 8) che estende TermGraphNode con proprietà e metodi propri dell'algoritmo, infine la classe UndirectedEdge (Figura 9), rappresentante di arco indiretto.

Nell'implementazione di tutti i metodi delle classi elencate è stata fatta la scelta progettuale di lasciare al chiamante il controllo che i dati in input siano ben formati.

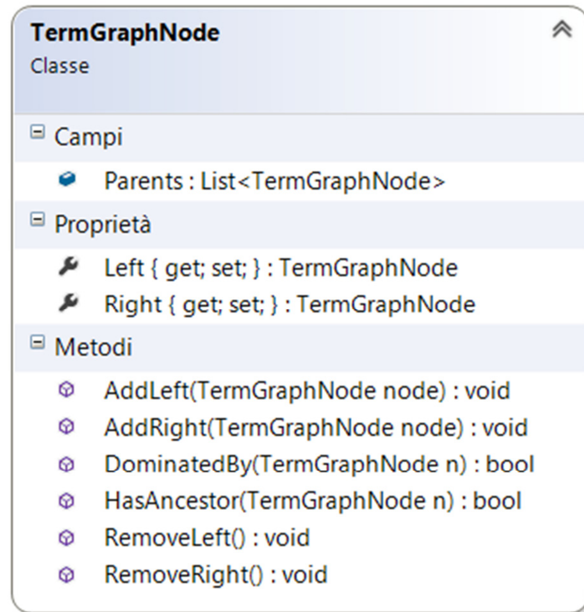

Figura 7 - TermGraphNode

La classe TermGraphNode è così definita

```
public class TermGraphNode
{
   public TermGraphNode Left { get; set; } = null;
   public TermGraphNode Right { get; set; } = null;
   public List<TermGraphNode> Parents = new List<TermGraphNode>();

   public void AddLeft(TermGraphNode node)
   {
      Left = node;
      node.Parents.Add(this);
   }

   public void AddRight(TermGraphNode node)
   {
      Right = node;
      node.Parents.Add(this);
   }

   public bool DominatedBy(TermGraphNode n)
   {
      if (this == n) return true;
      else if (Parents.Count() == 0) return false;
      else return Parents.TrueForAll(m => m.DominatedBy(n));
   }

   public bool HasAncestor(TermGraphNode n) =>
    (this == n) ? true : Parents.Exists(m => m.HasAncestor(n));

   public void RemoveLeft()
   {
      Left.Parents.Remove(this);
      Left = null;
   }

   public void RemoveRight()
   {
      Right.Parents.Remove(this);
      Right = null;
   }
}
```

Per definizione un nodo a è dominato da un nodo n se per ogni percorso che risale la gerarchia di a si passa da n. La funzione bool DominatedBy(TermGraphNode n) indica se l'istanza del nodo è dominata o meno da n.

Di seguito è indicato lo pseudocodice dell'algoritmo proposto dal professore Claudio Sacerdoti Coen, sul quale è stata effettuata una analisi per creare la classe EnrichedTermGraphNode (Figura 8) che estende i TermGraphNode con proprietà e metodi propri dell'algoritmo.

```
Procedure Main()
    while there is any alive node t do Finish(t);

Procedure Finish(r)
    stack ← 0;
    if canonic(r) is undefined then canonic(r) ← r else fail;
    while r has some alive parent t do Finish(t);
    while there is an undirected edge(r, t) do Push(stack, r, r, t);
    while not stack.empty() do
        s ← stack.pop();
        if r, s have different types then fail;
        while s has some alive parent t do Finish(t);
        while there is an undirected edge(s, t) do Push(stack, r, s, t);
        Propagate(r, s);
    end
    delete r and directed arcs out of it;
    return

Procedure Push(stack, r, s, t)
    if canonic(t) is undefined then
        canonic(t) ← r;
        stack.push(t);
    else if canonic(t) ≠ r then fail;
    delete undirected edge(s, t);

Procedure Propagate(r, s)
    if r = Var(i) and s = Var(j) then
        if binder(r) is undefined and binder(s) is undefined then
            if (i ≠ j) then fail;
        else if canonic(binder(r)) ≠ canonic(binder(s)) then fail;
    else if r = Lam(v1, r') and s = Lam(v2, s') then
        create undirected edges(v1, v2) and (r', s')
    else if r = App(r1, r2) and s = App(s1, s2) then
        create undirected edges(r1, s1) and (r2, s2)
    end
    delete s and directed arcs out of it
```

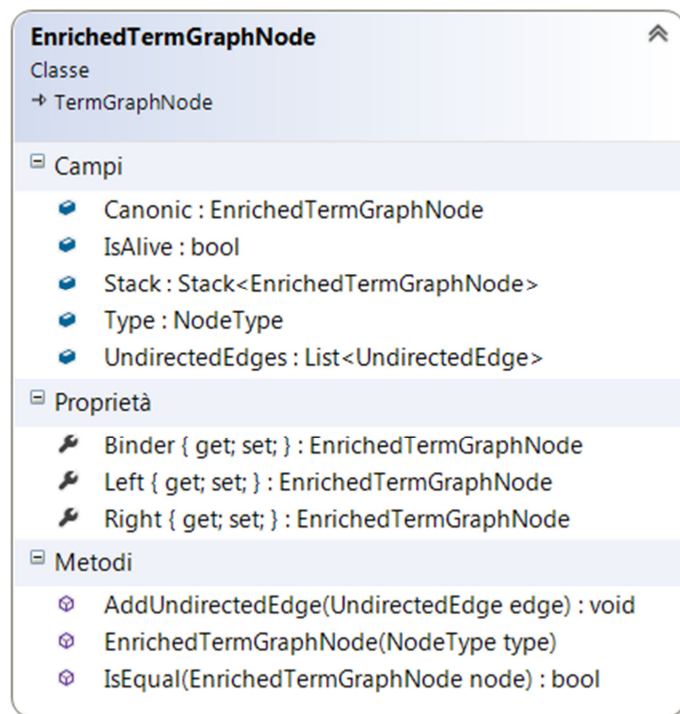

Figura 8 - EnrichedTermGraphNode

La classe EnrichedTermGraphNode è così definita

```
public class EnrichedTermGraphNode : TermGraphNode
{
   public EnrichedTermGraphNode Binder { get { return Left; } set { Left = value; } }
   public EnrichedTermGraphNode Canonic = null;
   public bool IsAlive = true;
   public Stack<EnrichedTermGraphNode> Stack = new Stack<EnrichedTermGraphNode>();
   public readonly NodeType Type;
   public readonly List<UndirectedEdge> UndirectedEdges = new List<UndirectedEdge>();

   public EnrichedTermGraphNode(NodeType type) { Type = type; }

   public void AddUndirectedEdge(UndirectedEdge edge) => UndirectedEdges.Add(edge);

   public bool IsEqual(EnrichedTermGraphNode node) => node.Type == Type &&
(Type == NodeType.Var ? node == this || Binder != null : (Left == Right) ==
(node.Left == node.Right) && node.Left.IsEqual(Left) && node.Right.IsEqual(Right));

   public new EnrichedTermGraphNode Left
   {
      get { return (EnrichedTermGraphNode)(base.Left); }
      set { base.Left = value; }
   }

   public new EnrichedTermGraphNode Right
   {
      get { return (EnrichedTermGraphNode)(base.Right); }
      set { base.Right = value; }
   }
}
```

Nella classe EnrichedTermGraphNode è stata assegnata la responsabilità di gestione degli archi indiretti, per maggiore leggibilità del codice. Tale responsabilità potrebbe in futuro essere spostata nella superclasse TermGraphNode.

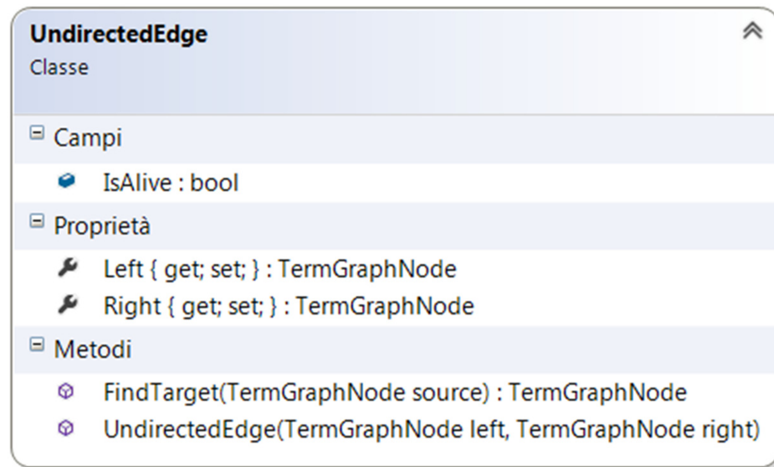

Figura 9 - UndirectedEdge UML

La classe UndirectedEdge è così definita.

```
public class UndirectedEdge
{
    public bool IsAlive = true;
    public EnrichedTermGraphNode Left { get; private set; }
    public EnrichedTermGraphNode Right { get; private set; }

    public UndirectedEdge(EnrichedTermGraphNode left, EnrichedTermGraphNode right)
    {
        Left = left;
        Right = right;
    }

    public EnrichedTermGraphNode FindTarget(EnrichedTermGraphNode source)
    {
        if (source == Left)
            return Right;
        else if (source == Right)
            return Left;

        throw new ExecutionException("can't find target.");
    }
}
```

La funzione EnrichedTermGraphNode FindTarget(EnrichedTermGraphNode source) dato in input uno dei due nodi collegati dall'arco, essa restituisce l'altro nodo.

Nel paragrafo successivo verrà spiegato il perché di questa implementazione.

E’ ora possibile definire la classe TermGraph, rappresentante dei lambda termini per mezzo di EnrichedTermGraphNode.

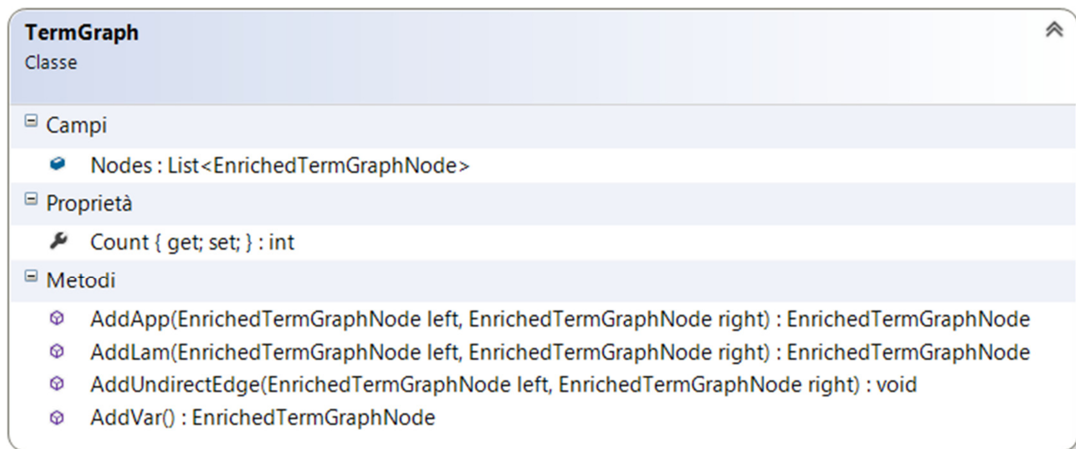

Figura 10 – TermGraph UML

La classe TermGraph è così definita.

```
class TermGraph
{
   public readonly List<EnrichedTermGraphNode> Nodes = new List<EnrichedTermGraphNode>();
   public int Count => Nodes.Count;

   public EnrichedTermGraphNode AddApp(EnrichedTermGraphNode left, EnrichedTermGraphNode right)
   {
      EnrichedTermGraphNode node = new EnrichedTermGraphNode(NodeType.App);
      node.AddLeft(left);
      node.AddRight(right);
      Nodes.Add(node);
      return node;
   }

   public EnrichedTermGraphNode AddLam(EnrichedTermGraphNode left, EnrichedTermGraphNode right)
   {
      if (left.Type != NodeType.Var)
         throw new BuildException("left.Type != Node.NodeType.Var");

      EnrichedTermGraphNode node = new EnrichedTermGraphNode(NodeType.Lam);
      node.AddLeft(left);
      node.AddRight(right);
      left.Binder = node;
      Nodes.Add(node);
      return node;
   }

   public EnrichedTermGraphNode AddVar()
   {
      EnrichedTermGraphNode node = new EnrichedTermGraphNode(NodeType.Var);
      Nodes.Add(node);
      return node;
   }

   public void AddUndirectEdge(EnrichedTermGraphNode left, EnrichedTermGraphNode right)
   {
      UndirectedEdge edge = new UndirectedEdge(left, right);
      left.AddUndirectedEdge(edge);
```

```
        right.AddUndirectedEdge(edge);
    }
}
```

La classe espone i metodi per aggiungere EnrichedTermGraphNode di tipo Add, Var, Lam, rispettando i vincoli di inserimento definiti in Figura 1,Figura 2,Figura 3.

2.3 Problema di cancellazione di un arco indiretto

Nella presente sezione è descritto il problema riscontrato durante la cancellazione di un arco indiretto selezionato. Nello pseudocodice tale problema era considerato banale e pensato con costo O(1), tuttavia nella sua implementazione risulta essere un problema non banale.

Dato un arco indiretto tra un nodo s e un nodo t ci si pone il problema di cancellare l'arco indiretto che collega s con t.
L'operazione di cancellazione deve quindi avere costo O(1), altrimenti l'algoritmo non ha più costo in tempo lineare sul numero di passi di beta-riduzione.

Questo paragrafo illustra i tentativi nel risolvere il problema della cancellazione dell'arco indiretto con costo costante, fino ad arrivare alla soluzione finale.

Il primo approccio implementativo è stato di rappresentare gli archi indiretti come elementi del TermGraph.
Nella classe TermGraph è stata aggiunta la collezione di archi indiretti come proprietà, dove ogni elemento della collezione è una coppia s e t.
Utilizzando questa rappresentazione, la selezione dell'arco da cancellare richiede la ricerca dell'arco con coppia di nodi s e t sulla collezione del TermGraph.
Tale ricerca non ha costo costante, ma dipende dall'implementazione della collezione, introducendo così un costo aggiuntivo nell'operazione di cancellazione.

Il secondo approccio è stato di rappresentare un arco indiretto come un arco diretto tra s e t, e viceversa.
Nella classe Node è stata aggiunta la collezione di nodi come proprietà, dove ogni elemento nella collezione rappresenta la destinazione di un arco diretto ed ha come valore il riferimento del nodo destinazione.
Per effettuare la cancellazione di un arco indiretto tra s e t è quindi necessario effettuare la cancellazione dell'arco diretto da s a t e viceversa.
Nel cancellare l'arco da t ad s, è necessario effettuare anche la ricerca di s sulla collezione di t.
Anche in questo caso la ricerca introduce un costo non costante.

La soluzione scelta (Figura 11) è stata quella di rappresentare gli archi indiretti tramite la classe UndirectedEdge, e aggiungervi la proprietà IsAlive per marcare l'arco come vivo/morto.

Nella classe Node è stata quindi aggiunta come proprietà la collezione di UndirectedEdge, dove ogni elemento nella collezione rappresenta un arco indiretto ed ha come valore, per entrambi i nodi, il riferimento alla stessa istanza di UndirectedEdge.

Durante la costruzione dell'istanza di un arco indiretto viene aggiunto il riferimento di tale istanza nella collezione di s e t; per effettuare la cancellazione è quindi sufficiente impostare la proprietà IsAlive a false di tale istanza.

Al successivo accesso all'istanza da entrambi i nodi, l'arco sarà trovato morto e non verrà gestito.

Tale soluzione non presenta costi di gestione aggiuntivi, perché se la proprietà IsAlive dell'arco ha valore false viene interrotta la computazione, inoltre ogni nodo e tutti i suoi archi indiretti vengono visitati una e una sola volta.

L'operazione assume così costo O(1), rispettando la semantica dell'algoritmo.

In Figura 11 è visibile la rappresentazione di una istanza della classe UndirectedEdge.

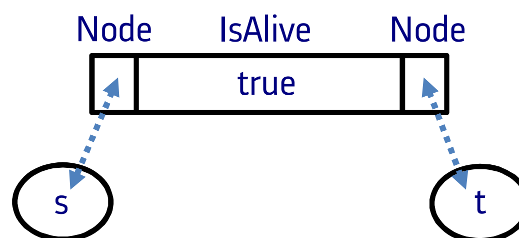

Figura 11- istanza di UndirectedEdge

2.4 Codice sorgente

Nella presente sezione è descritta l'implementazione dell'algoritmo e il relativo codice sorgente.

```
static class Algorithm
{
   private static TermGraph Tg;

   private static void Finish(EnrichedTermGraphNode r)
   {
      if (r.Canonic == null)
         r.Canonic = r;
      else
         throw new ExecutionException("Canonic(t) != null");

      foreach (EnrichedTermGraphNode t in r.Parents)
         if (t.IsAlive)
            Finish(t);

      foreach (UndirectedEdge edge in r.UndirectedEdges)
         if (edge.IsAlive)
         {
            EnrichedTermGraphNode t = edge.FindTarget(r);
            Push(r.Stack, r, t, edge);
         }

      while (r.Stack.Count > 0)
      {
         EnrichedTermGraphNode s = r.Stack.Pop();

         if (r.Type != s.Type)
            throw new ExecutionException("r.Type != s.Type");

         foreach (EnrichedTermGraphNode t in s.Parents)
            if (t.IsAlive)
               Finish(t);

         foreach (UndirectedEdge edge in s.UndirectedEdges)
            if (edge.IsAlive)
            {
               EnrichedTermGraphNode t = edge.FindTarget(s);
               Push(r.Stack, r, t, edge);
            }

            Propagate(r, s);
         }

         r.IsAlive = false;
      }
```

```
    private static void Propagate(EnrichedTermGraphNode r, EnrichedTermGraphNode s)
    {
        if (r.Type == NodeType.Var && s.Type == NodeType.Var)
        {
            if (r.Left == null || s.Left == null)
                throw new ExecutionException("Binder r || s undefined");
            else if (r.Canonic != s.Canonic)
                throw new ExecutionException("r & s have different Canonical Pointers");
        }
        else
        {
            Tg.AddUndirectEdge(r.Left, s.Left);
            Tg.AddUndirectEdge(r.Right, s.Right);
        }

        s.IsAlive = false;
    }

    private static void Push(Stack<EnrichedTermGraphNode> stack,
EnrichedTermGraphNode r, EnrichedTermGraphNode t, UndirectedEdge edge)
    {
        if (t.Canonic == null)
        {
            t.Canonic = r;
            stack.Push(t);
        }
        else if (t.Canonic != r)
            throw new ExecutionException("Canonic(t) != r");

        edge.IsAlive = false;
    }

    public static void Start(TermGraph tg)
    {
        Tg = tg;
        foreach (EnrichedTermGraphNode t in Tg.Nodes)
            if (t.IsAlive)
                Finish(t);
    }
}
```

Section 3 – Test e Risultati

Nella presente sezione sono descritte le modalità di test dell'algoritmo e i risultati ottenuti.

Nella prima fase di test ho deciso di generare programmaticamente tutti i TermGraph (3.1) con n nodi al variare di n, formando così una base dati su cui provare l'algoritmo.
Nella seconda fase di test ho verificato che, dando in input due TermGraph all'algoritmo, esso fornisse risultati corretti (3.2).
Infine ho testato l'efficienza dell'algoritmo (3.3), verificando che sia rispettato il costo in tempo lineare sul numero di passi di beta-riduzione.

I tempi indicati fanno sempre riferimento alla media dei risultati di 3 computazioni con input uguali.

3.1 Generazione di tutti i TermGraph

Nella presente sezione viene descritto il metodo utilizzato per la generazione di tutti i TermGraph ben formati, su cui verranno effettuati test funzionali e di efficienza dell'algoritmo.

Per effettuare la creazione di tutti i TermGraph ho scritto un algoritmo che, dato in input il numero di nodi n del TermGraph, genera progressivamente tutti i pseudo-TermGraph con numero di nodi n, facendo uso di backtracking.
Uno pseudo-TermGraph è un graph che rispetta i vincoli del TermGraph, senza rispettare necessariamente il vincolo di dominanza.

Dopo la generazione di ogni pseudo-TermGraph viene effettuato un test di dominanza, scartando i pseudo-TermGraph che non rispettano il vincolo di dominanza.

Eseguendo così l'algoritmo di generazione da 1 a N, teoricamente è possibile la generazione di tutti i TermGraph, nella pratica è stato notato che la quantità di TermGraph esistenti per numero di nodi cresce esponenzialmente (3.1), rendendo così impossibile generare in tempi ragionevoli tutti i TermGraph oltre i 10 nodi.

In Tabella 1 è possibile vedere i risultati nella generazione di tutti i TermGraph fino a 10 nodi.

n° nodi	n° TermGraph	n° pseudo-TermGraph	% Termgraph	tempo di generazione dei pseudo-TermGraph	tempo / n° pseudo-Termgraph
1	1	1	100	-	
2	2	2	100	-	
3	7	8	87,5	-	
4	40	50	80	-	
5	309	430	71,9	-	
6	3173	4846	65,5	15 ms	0,003095336
7	41462	68358	60,7	253 ms	0,003701103
8	666196	1167186	57,0	5,239 sec.	0,004488573
9	12796561	23514144	54,4	125,011 sec.	0,005316417
10	287208089	547863192	52,4	3395,404 sec.	0,00619754

Tabella 1- Generazione dei TermGraph

L'algoritmo di generazione dei TermGraph ha complessità non lineare, infatti al crescere del numero di nodi aumenta il tempo di generazione di ogni singolo TermGraph, il calcolo in Figura 12 viene eseguito dividendo il tempo di generazione dei pseudo-TermGraph per la loro quantità.

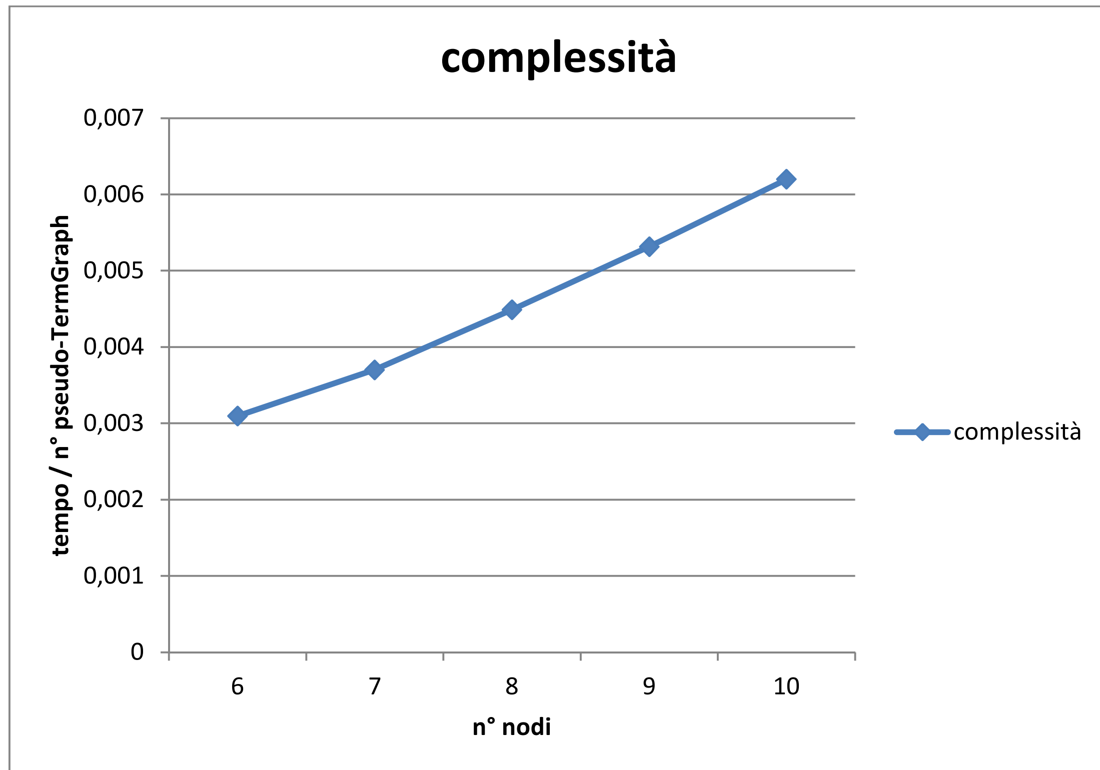

Figura 12 – Complessità dell'algoritmo di generazione

In Figura 13 e Figura 14 è fornita la rappresentazione grafica della crescita dei TermGraph e pseudo-TermGraph, al variare del numero di nodi.

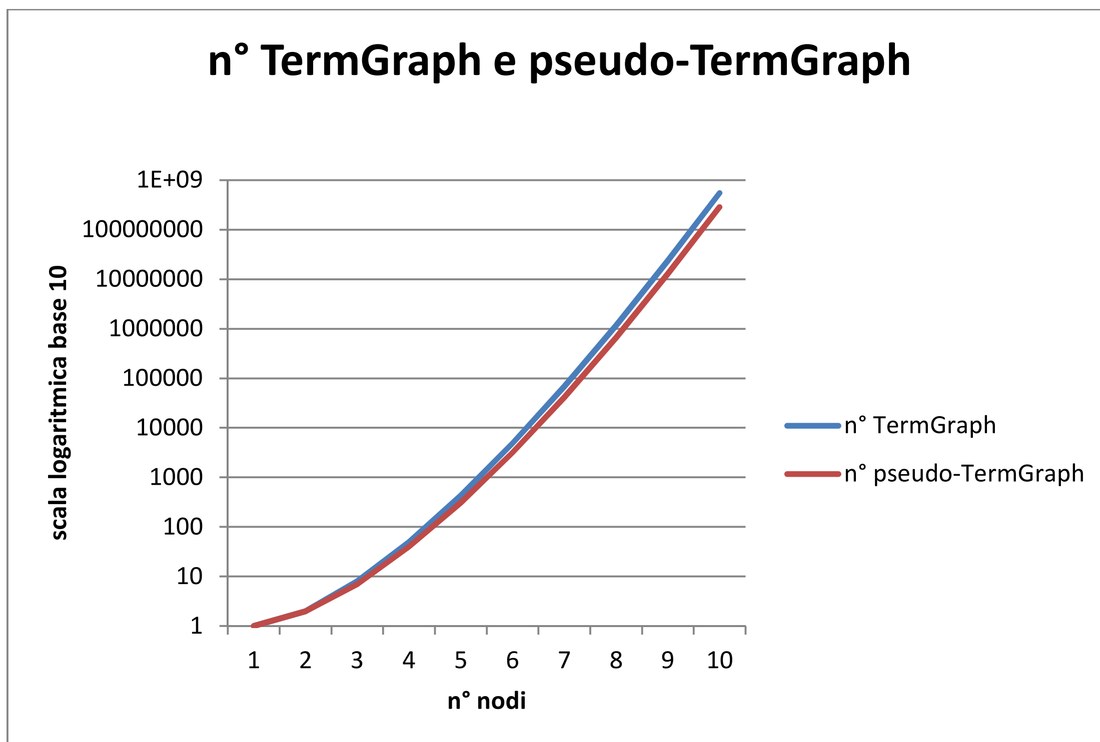

Figura 13 - n° TermGraph per n nodi

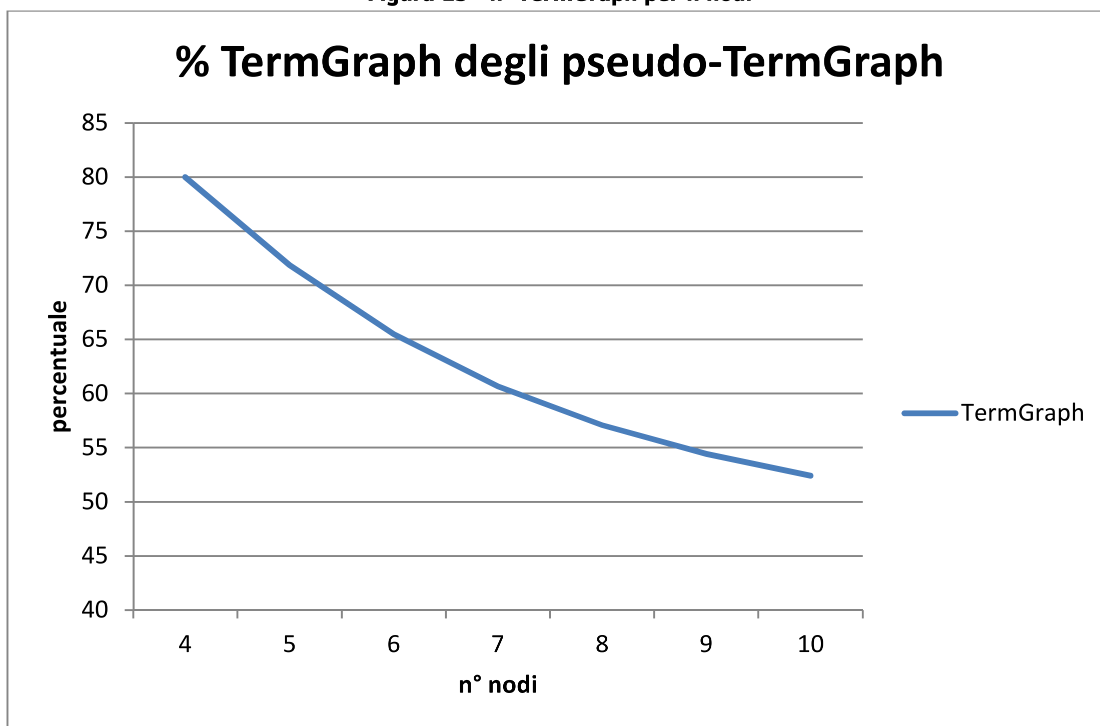

Figura 14 - % TermGraph

Per la generazione di tutti i TermGraph è stata creata la classe TermGraphBuilder (Figura 15).

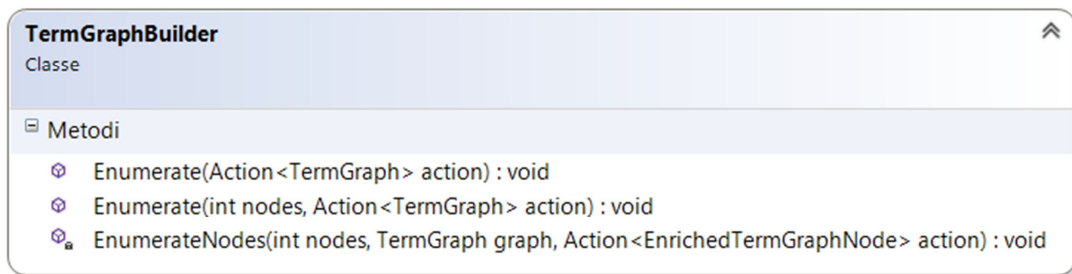

Figura 15 - TermGraphBuilder

La classe TermGraphBuilder è così definita

```
class TermGraphBuilder
{
   public static void Enumerate(Action<TermGraph> action)
   {
      for (int nodes = 1; ; nodes++)
      Enumerate(nodes, action);
   }

   public static void Enumerate(int nodes, Action<TermGraph> action)
   {
      TermGraph graph = new TermGraph();
      EnumerateNodes(nodes, graph, root =>
      {
         if (graph.Count == nodes && !graph.Nodes.Any(v => v.Type ==
NodeType.Var && v.Binder != null && !v.DominatedBy(v.Binder)))
            action(graph);
      });
   }

   private static void EnumerateNodes(int nodes, TermGraph graph,
Action<EnrichedTermGraphNode> action)
   {
      if (graph.Count < nodes)
      {
         EnrichedTermGraphNode var = graph.AddVar();
         action(var);
         graph.Nodes.Remove(var);

         EnrichedTermGraphNode app = new EnrichedTermGraphNode(NodeType.App);
         graph.Nodes.Add(app);
         EnumerateNodes(nodes, graph, left =>
         {
            if (!app.HasAncestor(left))
               EnumerateNodes(nodes, graph, right =>
               {
                  if (!app.HasAncestor(right))
                  {
                     app.AddLeft(left);
                     app.AddRight(right);
                     action(app);
                     app.RemoveRight();
                     app.RemoveLeft();
                  }
```

```
                });
            });
            graph.Nodes.Remove(app);

            if (graph.Count < nodes - 1)
            {
                EnrichedTermGraphNode lam = new EnrichedTermGraphNode(NodeType.Lam);
                graph.Nodes.Add(lam);
                lam.AddLeft(graph.AddVar());
                lam.Left.Binder = lam;
                EnumerateNodes(nodes, graph, right =>
                {
                    if (!lam.HasAncestor(right))
                    {
                        lam.AddRight(right);
                        action(lam);
                        lam.RemoveRight();
                    }
                });
                graph.Nodes.Remove(lam.Left);
                lam.RemoveLeft();
                graph.Nodes.Remove(lam);
            }
        }

        foreach (EnrichedTermGraphNode node in graph.Nodes.ToArray())
            action(node);
    }
}
```

3.2 Verifica funzionale

Nella presente sezione viene descritta la metodologia di test per la verifica della correttezza nell'implementazione dell'algoritmo.

In una prima fase sono state create piccole unità di test, con dati in input definiti manualmente, per verificare che l'algoritmo rispondesse correttamente agli input forniti.
Queste unità di test sono servite anche per verificare che ad ogni modifica del codice ne fosse ancora rispettata la correttezza.

Ultimata la scrittura dell'algoritmo, sono stati effettuati test su larga scala nella seguente modalità: per ogni TermGraph t con radice r generato si è effettuata una copia r_2 di r all'interno di t, effettuando un desharing parziale dei suoi figli (un nodo si uno no), quindi si sono collegate le radici r_1 e r_2 tramite arco indiretto, formando così il TermGraph di input i_1 desiderato dall'algoritmo.
Infine è stato lanciato l'algoritmo con input i_1.
Per verificare il corretto risultato dell'algoritmo è stato creato un secondo algoritmo, che preso in input i_1, ne effettua il deSharing massimo, formando così il TermGraph di input i_2.
Il secondo algoritmo risponde vero se le due radici di i_2 formano due TermGraph identici.

L'esempio in Figura 6 mostra il desharing massimo di radici aventi due TermGraph identici.

Nei test effettuati i risultati dei due algoritmi sono stati sempre concordi, aumentando così la confidenza nell'algoritmo in esame.

3.3 Efficienza dell’algoritmo

Nella presente sezione viene descritta la metodologia di test sull’efficienza dell’algoritmo.

In prima fase è stato generato un TermGraph per numero di nodi indicati in **Tabella 2**, successivamente è stato eseguito l’algoritmo 500 volte su ogni TermGraph.
Infine è stata fatta la media tra 3 computazioni, ottenendo così i tempi di esecuzione.

Nel calcolo del coefficiente di complessità è stato fatto il rapporto tra il tempo e la formula presente nell’articolo:

Funzione di ranking = tempo / (2 * #Nodi + #ArchiDiretti + 2)

N° Nodi	Tempo esecuzione (ms)	Funzione di ranking
150	59,5	0,396667
300	113,5	0,378333
450	203	0,451111
600	245	0,408333
750	342,5	0,456667
900	344,5	0,382778
1050	413	0,393333
1200	418	0,348333
1350	526	0,39
1500	620	0,413333

Tabella 2 – risultati

Come è possibile vedere dai risultati ottenuti e dalla sua rappresentazione in Figura 16, il coefficiente di complessità rimane circa costante, redendo confidenti del fatto che questa implementazione ha complessità lineare.

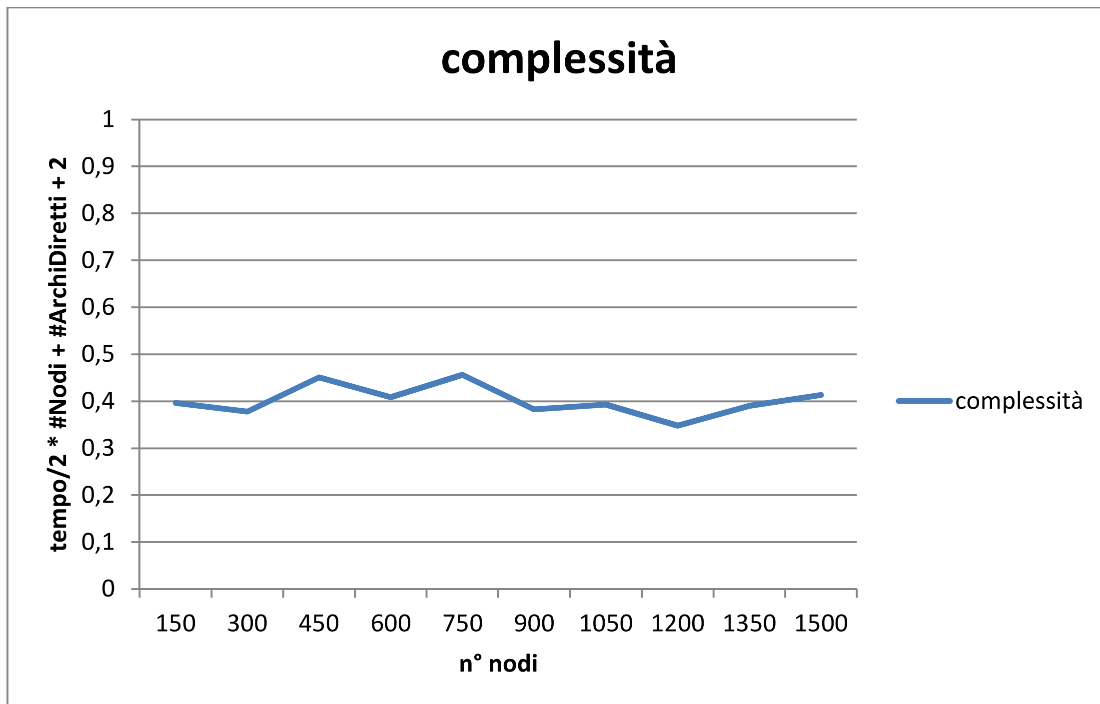

Figura 16 - complessità

3.4 Problemi aperti

Nella presente sezione sono descritti i problemi rimasti aperti per possibili sviluppi futuri.

Utilizzando l'algoritmo di generazione di tutti i TermGraph è possibile generare tutti quelli con massimo 10 nodi; è possibile anche generare grafi molto grandi con più di mille nodi, ma non in maniera esaustiva.

La generazione di TermGraph con numero di nodi deSharabili definita a priori rimane un problema aperto.

La generazione di tutti i TermGraph è un problema molto complesso. L'algoritmo naif utilizzato non ha complessità lineare, e pertanto sono state provate altre tecniche di generazione, ed alla fine è stato possibile trarre le seguenti conclusioni:

- Risulta facile generare tutti i TermGraph e verificare a posteriori che sia rispettata la dominanza, ma tale test ha un costo che rende la complessità non lineare.
- Utilizzando tecniche di backtracking è possibile creare direttamente TermGraph che rispettano la dominanza, ma è complesso riuscire a generarli tutti.
- Generando a priori espressioni let-in è possibile creare direttamente TermGraph che rispettano la dominanza, ma si generano TermGraph duplicati. Eliminare i duplicati rende la complessità non lineare.

La generazione di tutti i TermGraph con complessità lineare e di singoli TermGraph con quantità di nodi deSharabili definita a priori rimangono problemi aperti.

Section 4 – Conclusioni

Grazie all'implementazione descritta nel primo capitolo e ai relativi test descritti nel secondo capitolo è stato possibile rispettivamente implementare l'algoritmo, verificarne i risultati e valutarne i tempi di esecuzione.

E' possibile suddividere l'esperienza di tesi in varie fasi.

La prima fase di apprendimento del problema e delle terminologie adatte a descrivere il problema; fase in cui ho acquisito maggiori competenze nello studio dei grafi e nel lambda calcolo.

La seconda fase progettuale di dichiarazione delle strutture necessarie volte a risolve il problema per mezzo dell'algoritmo, nella quale non ho avuto particolari difficoltà, grazie all'esperienza maturata con il linguaggio di programmazione C#.

La terza fase di implementazione dell'algoritmo, che mi ha posto davanti al un problema descritto nel cap 2.3 e che ha richiesto un ragionamento più approfondito e una modifica nella dichiarazione delle strutture.

La quarta fase di testing dell'algoritmo, probabilmente la più complessa, che mi ha posto davanti numerose difficoltà nella creazione di tutti i TermGraph.
Difficoltà che, in buona parte, sono state superate anche grazie all'aiuto del professore.

Nella fase di verifica dei risultati è stato necessario capire intuitivamente come funziona l'algoritmo e creare piccoli test per verificare se l'algoritmo rispondeva in maniera conforme allo pseudocodice. In questa fase è stato anche necessario inventare un piccolo algoritmo di controprova, descritto nel cap. 3.2.

Nell'ultima fase di test sull'efficienza dell'algoritmo sono state incontrate numerose difficoltà a creare dei test significativi, anche queste superate grazie all'aiuto del professore.

Complessivamente il processo di sviluppo della tesi mi ha impegnato per circa cento ore, facendomi acquisire maggiori conoscenze su problematiche legate ai grafi, sull'implementazione di algoritmi complessi e sui test di efficienza. Argomenti, gli ultimi due, che non avevo mai avuto l'occasione di approfondire.

Grazie a questa esperienza ho acquisito maggiore coscienza di come la scrittura di un algoritmo in pseudocodice possa indurre ad usare strutture dati sbagliate; e dell'importanza, ma anche della difficoltà, del creare una buona base dati su cui effettuare i test necessari.

A – Bibliografia

[1] Beniamino Accattoli e Claudio Sacerdoti Coen, *"On the Relative Usefulness of Fireballs",* IEEE Computer Society, LICS '15 Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pp. 141-155, 2015

[2] Beniamino Accattoli e Ugo Dal Lago, *"(Leftmost-Outermost) Beta Reduction is Invariant, Indeed.",* ACM, Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), vol. 8, pp. 1-10, 2014

[3] Ugo Dal Lago e Simone Martini, "*The weak lambda calculus as a reasonable machine.",* Theorical Computer Science, vol.398, Issues 1-3, pp. 32-50, 2008

[4] Beniamino Accattoli e Ugo Dal Lago, "*On the Invariance of the Unitary Cost Model for Head Reduction (Long Version)",* CoRR, vol. abs/1202.1641, 2012

[5] Andrea Asperti, *"On the complexity of beta-reduction.",* ACM, Proc. 23rd ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 110–118, 1996

[6] Agostino Dovier e Carla Piazza, "The Subgraph Bisimulation Problem and its Complexity", ACM/IEEE Transactions on Knowledge and Data Engineering, vol 15, Issue 4, pp. 1055-1056, 2003

B – Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno aiutato nella stesura della tesi. A loro va la mia gratitudine, anche se spetta soltanto a me la responsabilità per ogni errore contenuto in questa tesi.

Ringrazio anzitutto il professor Claudio Sacerdoti Coen, senza il cui supporto e la guida sapiente questa tesi non esisterebbe.

Un ringraziamento particolare va anche ai colleghi ed agli amici che mi hanno incoraggiato o che hanno speso parte del proprio tempo per leggere e discutere con me le bozze del lavoro.

Vorrei infine ringraziare le persone a me più care: la mia famiglia, e il mio compagno di studi e amico Rodolfo, a cui questo lavoro è dedicato.