CSP alldifferent backtracking e consistenza locale

Progetto di Sistemi con vincoli

a.a. 2011/2012

Docente: prof.ssa Francesca Rossi

Autori: Simone Tiso, Andrea Imparato

1 Il vincolo alldifferent

In questo documento verrà analizzato il problema con vincoli alldifferent che riveste una particolare importanza nella programmazione con vincoli per il suo frequente utilizzo e la sua non così ovvia implementazione. Il problema si presenta naturalmente in una larga varietà di problemi come ad esempio i puzzle, il problema delle n regine o i problemi di schedulazione e assegnamento. Affinchè il vincolo alldifferent sia soddisfatto deve essere possibile assegnare un valore diverso ad ogni variabile scegliendolo nel dominio della stessa. Un possibile esempio di problema alldifferent può essere:

$x_1 = \{1, 2\} x_2 = \{2\} x_3 = \{2, 3\}$ ammette soluzione con il seguente assegnamento per le variabili: $x_1 = 1 x_2 = 2 x_3 = 3$.

Più formalmente si ha che:

Siano x_1, x_2, . . . , x_n *variabili con i rispettivi domini* D_{x1},D_{x2}, . . . , D_{xn} *allora*

alldifferent(x_1, x_2, . . . , x_n) *è consistente se e solo se*

$$(x_1, x_2, \ldots, x_n) \in \{ (d_1, d_2, \ldots, d_n) \mid d_i \in D_{xi},\ d_i \neq d_j,\ i \neq j \}$$

Con la definizione precedente il problema è definito da un solo vincolo n-ario, questo non ci permette di ottenere buone prestazioni a livello computazionale. E' necessario dunque una scomposizione del problema che permetta ovviamente di preservare l'insieme delle soluzioni del problema originario. Per questo vincolo possiamo attuare la decomposizione binaria, così definita:

Sia C un vincolo su $x_1, \cdots, x_n$. *Una decomposizione binaria di C è un insieme minimale di vincoli binari* $C_d ec = \{C_1, \cdots, C_k\}$ *(per k>0) sulle coppie di variabili* $x_1, \cdots, x_n$ *tali che l'insieme delle soluzioni di C sia equivalente all'insieme delle soluzioni di* $\cap_{i=1}^{k} C_i$.

La decomposizione binaria di alldifferent$(x_1, \cdots, x_n)$ dunque è:

$$\bigcup_{1 \leq i < j \leq n} x_i \neq x_j$$

In questo modo otteniamo una definizione del problema equivalente a quella data ma lo sforzo computazionale per risolverla è di gran lunga ridotto e possiamo applicare i nostri algoritmi di consistenza.

2 Generazione casuale di problemi

Al fine di testare le implementazioni dei vari algoritmi di consistenza da noi trattati in questo documento, è stato necessario ideare un metodo per la generazione di problemi casuali, l'algoritmo per la generazione dei CSP funziona nel seguente modo:

Riceve in input il numero n di variabili del problema da generare, per ciascuna variabile x_i il suo dominio viene inizializzato con valori nell'intervallo $[a, b]$, $a \leq b$, con a e b scelti casualmente in modo uniforme nell'intervallo 1. . .n.

3 Backtracking

Per ottenere una soluzione di un problema alldifferent è necessario implementare un algoritmo di ricerca nello spazio delle soluzioni del problema stesso, lo pseudocodice dell'algoritmo in questione si può trovare all'indirizzo `http://www.math.unipd.it/~frossi/2007-ch8.pdf`. A grandi linee l'algoritmo funziona nel seguente modo:

Per ogni v_i variabile del problema, per ogni $d_j$valore del dominio della variabile v_i , l'algoritmo istanzia $D_{v_i} = \{d_j\}$. Nella versione del backtracking senza "domain filtering" dopo aver scelto per ogni variabile un valore del suo dominio, dovrebbe essere necessario controllare se l'assegnamento ottenuto è una soluzione o meno del problema e questo comporta un aumento esponenziale dell'albero di ricerca. Per effettuare un pruning dell'albero di ricerca sono stati implementati vari algoritmi che ottengono la consistenza locale dei problemi. In questo modo infatti durante la ricerca è possibile vedere a priori se un problema è consistente o meno prima di aver assegnato ad ogni variabile un valore.

4 Consistenza locale

Abbiamo implementato diversi algoritmi per ottenere la consistenza locale. Di seguito una descrizione approfondita per ognuno di essi.

4.1 Arc Consistency

Un vincolo binario C(x1,x2) è consistente sugli archi se per ogni valore di $d1 \in x1$ esiste un valore $d2 \in x2$ tale che $(d1, d2) \in C$, e per ogni valore $d2 \in x2$ esiste un valore $d1 \in x1$ tale che $(d1, d2) \in C$. Per ottenere la consistenza sugli archi

dopo aver effettuato la decomposizione binaria del problema alldifferent basta procedere nel seguente modo:

Ogni qualvolta il dominio di una variabile contiene un solo valore, questo viene rimosso dai domini delle altre variabili. Questo è ripetuto fino a quando non si ottiene un dominio vuoto o nessun dominio viene cambiato.

4.2 Intervallo di Hall

Prima di introdurre la bounds consistency è necessario definire cos'è un intervallo di Hall:

Siano x_1, x_2, . . . , x_n *variabili con i rispettivi domini finiti* D_{x1},D_{x2}, . . . , D_{xn}. *Dato un intervallo i, definiamo* $K_i = \{x_i | d_i \subseteq i\}$. *i è un intervallo di Hall se* $|i| = |K_i|$. *Le variabli esterne all'intervallo* K_i *dovreanno assumere valori al di fuori dell'intevallo i .*

4.3 Bounds Consistency

Il vincolo alldifferent(x_1, x_2, . . . , x_n) è bounds consistent se e solo se $|D_i| \geqslant 1$ $i = (1...n)$ e

1. Per ogni intervallo i: $|K_i| \leqslant i$
2. Per ogni intervallo di Hall $i : \{minD_i, maxD_i\} \bigcap I = \emptyset$ per ogni $x_i \notin K_I$.

4.4 Insieme di Hall

Siano x_1, x_2, . . . , x_n *variabili con i rispettivi domini finiti* D_{x1},D_{x2}, . . . , D_{xn}. *Dato* $K \subseteq \{x_1, x_2, ..., x_n\}$,*definiamo l'intervallo* $I_K = [minD_K, maxD_K]$. *K è un insieme di Hall se* $|K| = |I_K|$.

4.5 Range Consistency

Il vincolo alldifferent(x_1, x_2, . . . , x_n) è range consistent se e solo se $|D_i| \geqslant 1$ $i = (1...n)$ e $D_i \bigcap I_K = \emptyset$ per ogni insieme di Hall $K \subseteq x_1, x_2, ..., x_n$ e $x_i \notin K$.

5 Teoria dei Grafi

Per comprendere meglio l'algoritmo in questione e capire fino in fondo il suo funzionamento occorre introdurre alcune nozioni sui grafi. In questa sezione inoltre vengono presentati alcuni teoremi principali utilizzati dall'algoritmo per calcolare la consistenza sugli iperarchi.

Un grafo è definito informalmente come una coppia $G = (V, E)$ dove V è un insieme finito di vertici ed E è un insieme finito di coppie (V, V) chiamate archi. Un arco da $u \epsilon V$ a $v \epsilon V$ è indicato con uv. Un grafo orientato contiene al suo interno "archi orientati " ossia archi caratterizzati da una direzione (es: arco da u a v). Un grafo $G = (V, E)$ è *bipartito* se esiste una partizione S, T tale che $E \subseteq \{s, t \mid s \epsilon S, t \epsilon T\}$, denotato anche $G = (S, T, E)$.

5.1 Matching theory

Dato un grafo non orientato $G = (V, E)$, un *matching* in G è un insieme $M \subseteq E$ di archi disgiunti (ex: due archi in M non possono condividere un vertice). Da questo si ricavano alcune nozioni importanti per la costruzione e la gestione del grafo:

- *M-free* vertex: Vertice v che non è coperto da M in G.
- *Massimo accoppiamento:* si tratta di trovare il massimo accoppiamento tra i vertici nelle due partizioni del grafo, in particolare si cerca di trovare un accoppiamento di grado massimo quindi da coprire più vertici possibile.
- *Path M-augmentin:* Sia M un maching in G, P è chiamato M-augmentin se P ha lunghezza dispari, termina in un vertice fuori da M, ed i suoi archi sono alternativamente furi e dentro M.
- *M-alternating:* Un circuito C è chiamato M-alternating se i suoi archi sono alternativamente fuori e dentro M.

5.2 Jgrapht

JGraphT è una libreria grafica open-source in Java che fornisce degli oggetti e algoritmi matematici basati sulla teoria dei grafi..JGraphT supporta vari tipi di grafi tra cui:

- Grafi orientati e non orientati;
- Grafi con archi pesati / non pesati / etichettati o di un qualsiasi tipo definito dall'utente;
- Varie opzioni di molteplicità di arco, tra cui:
  - grafi semplici-multigraphs, pseudographs;
  - grafi non modificabili - moduli che permettono di fornire un accesso "sola lettura" per i grafi interni;
  - grafi e sottografi che sono auto-aggiornamento delle viste sottografo su altri grafici.;

Anche se potente, JGraphT è stato progettato per essere semplice e type-safe (via generici Java). Per esempio, i vertici di un grafo possono essere di qualsiasi oggetto, inoltre è possibile creare grafi in base a: Stringhe, URLs, documenti XML, ecc, si può creare anche grafici di grafici. Oltre a questo sono implementati alcuni algoritmi particolarmente ottimizzati per operazioni su grafi, alcuni di questi sono descritti di seguito:

- *ConnectivityInspector :* Ispeziona il grafo per controllare se è di tipo "connesso";

- *CycleDetector :* Controlla se sono presenti cicli all'interno del grafo;
- *EdmondsKarpMaximumFlow :* Utilizza l'algoritmo di Edmonds-Karp per calcolare il "Flusso massimo", oltre che il valore del flusso questa classe ritorna anche il sottografo corrispondente;
- *KruskalMinimumSpanningTree :* Calcola l'albero di copertura minimo del grafo;
- *StrongConnectivityInspector :* Questa classe controlla se il grafo è fortemente connesso e ritorna una lista di sottografi corrispondenti alle componenti fortemente connesse del grafo di partenza.

6 Implementazione Algoritmo

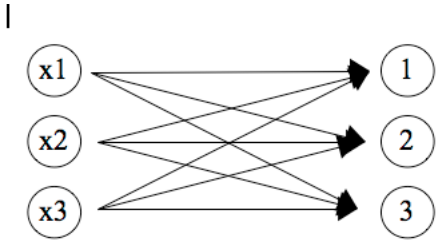
Questo algoritmo per ottenere la consistenza sugli iperarchi per il vincolo all-different è stato proposto da Règin (1994) e si basa fondamentalmente sulla teoria del matching, descritta nella sezione 2.1. Per l'implementazione di questo algoritmo si è utilizzato il linguaggio java che pur essendo più lento in termini computazionali offre una maggior versatilità in merito alle librerie che occorrevano a questo progetto. Di seguito definiamo alcuni teoremi necessari a descrivere successivamente il funzionamento dell'algoritmo.

- *Definizione 1 (Tight set):* Siano x_1, x_2, . . . , x_n variabili con rispettivi domini finiti D_1, D_2, . . . , D_n, $K \subseteq \{x_1, x_2, ..., x_n\}$ è un *tight set* se $\mid K \mid = \mid D_K \mid$.
- *Teorema 1:* Il vincolo all-different(x_1, x_2, . . . , x_n) è consistente sugli iperarchi se e solo se $\mid D_i \mid \geq 1$ $(i = 1, \ldots, n)$ e $D_i \bigcap D_K = \emptyset$ per ogni tight set $K \subseteq \{x_1, x_2, ..., x_n\}$ e ogni $x_i \notin K$.

Seguendo il teorema 1 il problema di consistenza potrebbe essere sviluppato generando tutti i tight set K ed aggiornando i relativi domini per tutte le variabili, similmente a ciò che si farebbe per la consistenza sui limiti e sugli intervalli. Tuttavia nella pratica questo tipo di approccio è irrealizzabile, in quanto il numero di sottoinsiemi K su n variabili è esponenziale in n. Per questo motivo si utilizzano i grafi, in particolare la *matching theory* riferita ad essi.

- *Definizione 2 (value graph):* Sia X una sequenza di variabili. Il grafo bipartito $G = (X, D(X), E)$ con $E = \{xd \mid d \in D(x), x \in X\}$ è chiamato il grafo dei valori di X.
- *Teorema 2:* Sia G un grafo e M un accoppiamento massimo in G. Un arco appartiene all'accoppiamento massimo in G se e solo se: o appartiene ad M, o ad un ancora M-alternating path partendo da un vertice M-free, o ad un ancora in un M-alternating circuito.

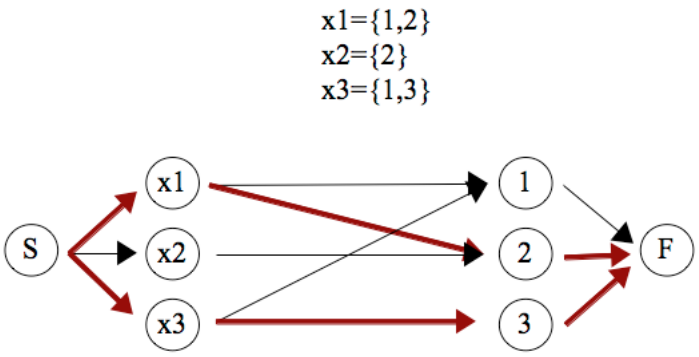
L'algoritmo inizia costruendo il value graph G relativamente alla sequenza di variabili del problema, in particolare ogni variabile è rappresentata da un oggetto che al suo interno contiene il relativo dominio. Il problema è rappresentato da questo grafo G bipartito orientato, il quale contiene le variabili x_1, x_2, . . . , x_n in una partizione e i valori dei domini $1, 2, \ldots, n$, in particolare ogni variabile è connessa ai valori del suo dominio mediante n archi, orientati dalla variabile al valore, con $n = |D_n|$. Una rappresentazione grafica si può vedere nella figura seguente:



Grafo orientato bipartito

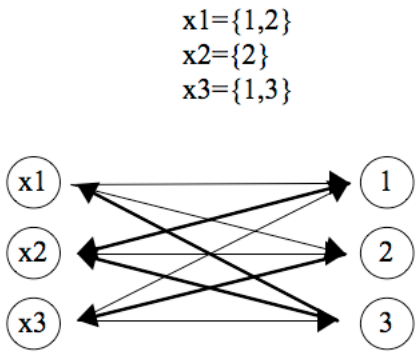
Una volta che il grafo è creato si calcola l'accoppiamento massimo M in G, inserendo due vertici S (sorgente) ed F (destinazione) al grafo G. Il vertice S è connesso a tutte le variabili attraverso archi: orientati da S a variabile, analogamente il vertice F è connesso a tutti i valori dei domini: da valore a vertice F. Ci si è serviti di questa particolare modifica per applicare l'algoritmo di Edmund-karp, il quale permette di calcolare il sottografo relativo al flusso che parte da un sorgente S e raggiunge una destinazione F in $O(VE)$. A seguito di questa piccola modifica è possibile ottenere l'accoppiamento massimo M in G. Inoltre è necessario effettuare un controllo sulla cardinalità di M, in particolare se quest'ultima dovesse risultare minore rispetto alla cardinalità di X (numero di variabili) significherebbe che nell'accoppiamento massimo non sono presenti tutte le variabili del problema, per cui non ci sarebbe speranza di ottenere la consistenza. In questo caso l'algoritmo ritorna un valore di tipo "false".

Il grafo risultante è mostrato nella figura seguente:



Grafo massimo accoppiamento

Successivamente è definito in nuovo grafo G_M costruito nel modo seguente: gli archi che appartengono ad M sono orientati dalla variabile dal relativo valore nel dominio, mentre gli archi che non appartengono ad M sono orientati nel verso opposto come in figura seguente:



Grafo orientato bipartito G_M

Inizialmente in questo grafo tutti gli archi sono marcati come "inconsistenti". Fatto questo si calcolano le componenti fortemente connesse del grafo G_M sfruttando una classe disponibile nel pacchetto "jgrapht", la quale ritorna una lista di sottografi orientati definiti come "componenti fortemente connesse". Gli archi tra vertici di una stessa componente fortemente connessa appartengono ad un "M-alternating circuit" in G_M e vengono marcati come consistenti a prescindere dal verso in cui sono nel grafo. Successivamente si cercano i vertici M-free (vedi definizione sezione 2.1) e si esegue una ricerca breadth-first partendo proprio

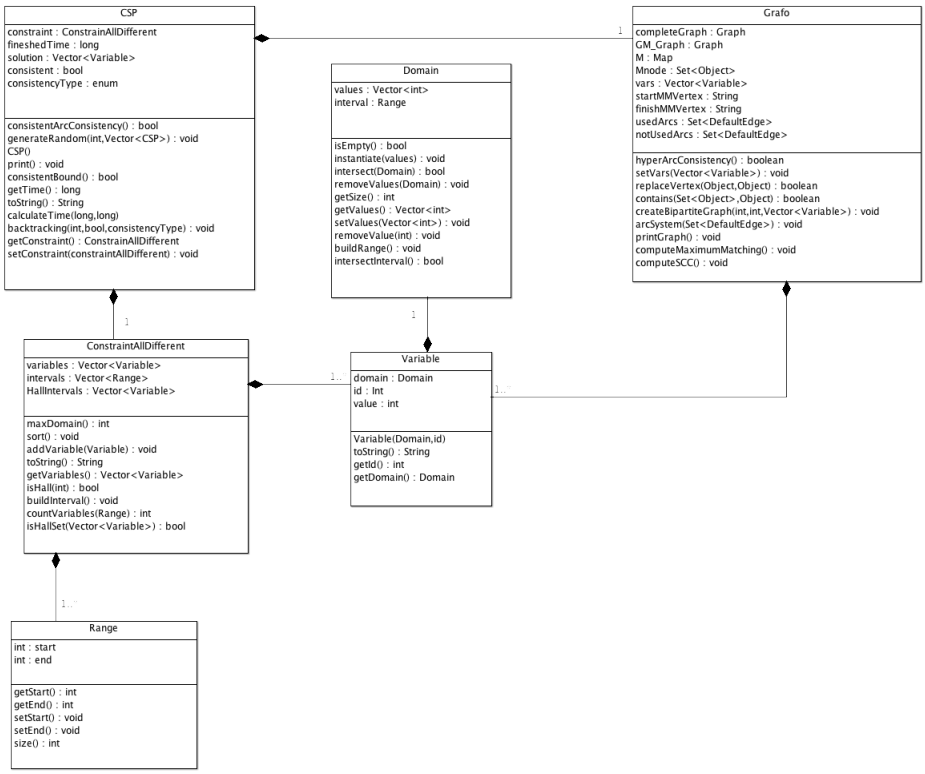
da essi, in questo modo si ispezionano tutti gli archi connessi e quest'ultimi si marcano come "consistenti" perchè appartengono ad un cammino diretto al grafo G_M con sorgente un vertice M-free. Questo lo si fa in tempo $O(m)$ con m = numero degli archi.

Dopo aver preparato la struttura dati (il grafo) ed agito su di essa, è possibile considerare fisicamente i domini delle variabili effettuando un "ciclo for" su tutti gli archi del grafo G_M marcati come "inconsistenti". Per ogni arco xd si esegue la seguente istruzione: $D(x) = D(x) - d$, in pratica si rimuove il valore d dal dominio della variabile x . Se qualche dominio dovesse diventare vuoto nel corso del processo di rimozione dei valori, l'algoritmo ritorna un valore di tipo "false". Se invece il ciclo termina con almeno un valore dentro il dominio di ogni variabile l'algoritmo ritorna un valore "true" per indicare che è stata ottenuta la consistenza.

In conclusione questo algoritmo permette di controllare se un problema è consistente sugli iperarchi, in caso contrario renderlo tale, relativamente al vincolo all-Different con un sforzo computazionale di $O(m\sqrt{n})$.

7 Implementazione

Di seguito lo schema UML relativo alle classi del progetto



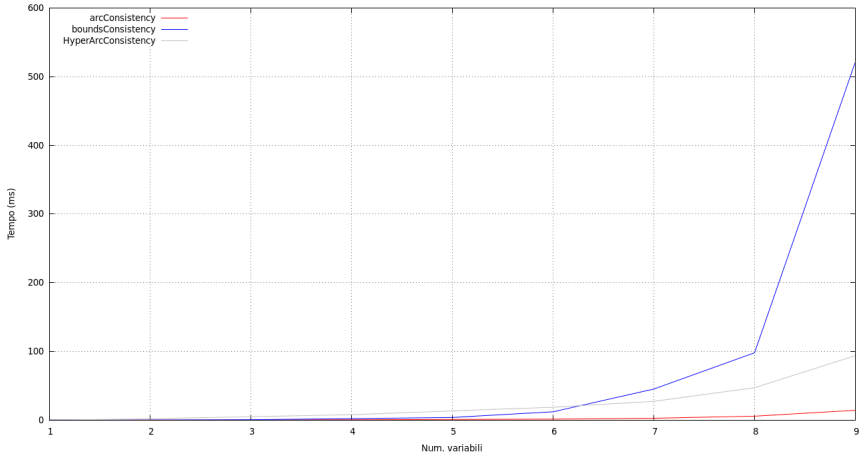
- *CSP:* Rappresenta il problema alldifferent
- *CostraintAllDifferent*: il vincolo del problema contiene al suo interno le variabili
- *Range:* intervallo globale delle variabili
- *Variable:* variabili del problema
- *Domain:* dominio intero delle variabili
- *Grafo:* rappresenta il grafo utilizzato dall'algoritmo hyperArcConsistency.

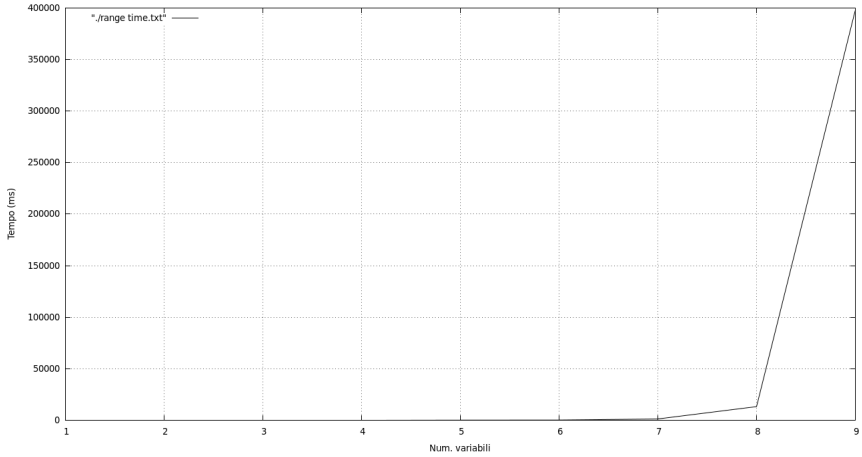
8 Manuale d'uso

Per avviare l'applicazione dare il seguente comando: *java -jar project.jar* [$numVar$] [$algoConsistenza$] dove numVar indica il limite massimo di variabili dei problemi generati casualmente e algoConsistenza è rispettivamente: *"arc, bounds, range, bipartite"* ed indica il rispettivo algoritmo di consistenza da applicare. Per ogni istanza il programma genera due file di log che hanno nome "algoConsistenza time.txt" e "algoConsistenza log.txt" che contengono rispettivamente: il primo la media dei tempi di esecuzione per ogni problema generato con variabili da 1 a numVar, il secondo contiene gli stati finali di tutti i problemi generati dopo aver applicato il backtracking (il quale applica l'algoritmo di consistenza indicato da *algoConsistenza*).

9 Risultati

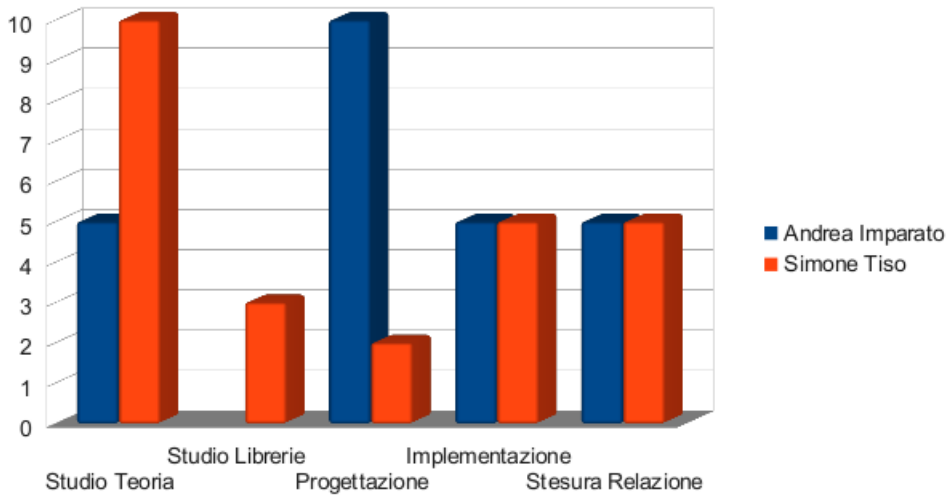
Di seguito i grafici dei tempi d'esecuzione in funzione del numero di variabili dei problemi. I tempi sono stati ottenuti generando e risolvendo per ogni istanza 100 problemi casuali e facendo una media dei loro relativi tempi d'esecuzione.




"./range time.txt"
400000
350000
300000
250000
200000
150000
100000
50000
0
Tempo (ms)
1
2
3
4
5
6
7
8
9
Num. variabili

10 Consuntivo

Di seguito sono inserite le ore di consuntivo per lo sviluppo di questo progetto per un totale di 25 ore per ciascun membro del gruppo.



11 Conclusioni e sviluppi futuri

Come si può vedere dai grafici dei tempi d'esecuzione solo applicando l'algoritmo di arcConsistency si riescono ad ottenere delle soluzioni in un tempo accettabile. Questo è dovuto al fatto che abbiamo implementato gli algoritmi di consistenza così com'erano nella loro definizione "ufficiale". Il risultato più sorprendente che non ci aspettavamo, è relativo al confronto tra l'algoritmo arcConsistency ed hyperArcConsistency, il primo infatti ha una complessità computazionale maggiore del secondo e minore anche degli altri algoritmi considerati, ma nonostante questo hyperArcConsistency ha dei tempi di esecuzione maggiori. Questo

è probabilmente dovuto ad un overhead nell'uso delle librerie jgrapht. Se avessimo avuto più tempo, potrebbe essere stato interessante implementare altri algoritmi come quello di Bartàk del 2003.