Progetto del corso di Ragionamento Automatico:

Algoritmo DPLL con sviluppo di diverse euristiche di splitting

Anno accademico: 2009/2010
Docente: Alessandro Farinelli

Realizzata da:
Peroni Nicola vr077942

Indice

1. Introduzione

1.1. Introduzione all'algoritmo DPLL

L'algoritmo DPLL utilizzato è quello “ricorsivo” con l'utilizzo della “Unit Propagate”.

L'algoritmo prende in input una formula CNF e restituisce:

1. La soddisfacibilità della formula e (se soddisfacibile) la soluzione trovata (che non è detto essere l'unica).
2. Il numero di operazioni di SPLIT utilizzate
3. Il tempo impiegato a completare l'algoritmo

Il programma utilizzato mette a disposizione una piccola interfaccia utente (vedi 1.2) che forza l'esecuzione con una formula CNF ben scritta, mentre l'output viene rappresentato solo testualmente nella shell.

L’idea che sta alla base dell’implementazione è quella di simulare in maniera automatica l’esecuzione del problema che si esegue solitamente a mano in un esercizio.

Il programma è realizzato in c++ ed utilizza alcune direttive dell'ambiente linux.

1.2. Interfaccia utente

L'interfaccia utente mostra la formula ed il tipo di euristica di splitting attualmente selezionate e permette di scegliere una opzione tra:

1. Cambiare formula corrente tra quelle caricate dal file “formule.dat” (presente nella stessa directory dell'eseguibile).
2. Inserire una nuova formula, tramite un form che realizza una formula ben formata in CNF.
3. Caricare un’instanza da file *.cnf
4. Cambiare l'eursitca di splitting corrente tra le 4 proposte.
5. Eseguire l'algoritmo DPLL.
6. Uscire dal programma.

1.3. Esecuzione e strutture dati usate

L'esecuzione che utilizza il programma “segue” l'algorimto descritto nelle slides adattandolo alla struttura dati utilizzata.

Per comodità, inizialmente la struttura della formula è in formato di stringa (più precisamente è in una struttura “frml” che contiene un campo stringa e due interi con numero di clausole e numero di letterali totale), in maniera tale da poter gestire caricamento da file ed inserimento senza troppi problemi.

La struttura cambia però al momento dell'esecuzione.

Al momento dell'esecuzione viene infatti prima invocato un metodo *init_DPLL* (in Formal_DPLL.cpp) che prende in input la formula e la mette in una matrice di stringhe *matrix_cnf[ ][ ]* dove le righe corrispondolo alle clausole e le colonne ai letterali di ogni clausola. Inoltre salva nel vettore *unitarie[ ]* le clausole unitarie trovate scorrendo la stringa.

Esistono delle variabili globali *max_clausole* e *max_letterali* impostate inizialmente a 100. E' dunque possibile costruire una matrice di dimensione massima max_clausole*max_letterali.

Ad esempio la matrice dopo l’esecuzione di *init_DPLL* sulla formula *(X1 or X2 or !X3),(!X1 or !X2 or !X3),(X1 or X3),(!X1)* avrà la seguente forma:

i/j	0	1	2	...	max_letterali
0	X1	X2	!X3	“”	“”
1	!X1	!X2	!X3	“”	“”
2	X1	X3	“”	“”	“”
3	!X1	“”	“”	“”	“”
...	“”	“”	“”	“”	“”
max_clausole	“”	“”	“”	“”	“”

Mentre il vettore delle unitarie conterrà solamente “!X1” alla posizione 0.

1.3.1. Unit Propagate

Una volta impostata la matrice, l’algoritmo prevede l’esecuzione della *Unit propagate* che propaga tutti i letterali presenti nel vettore *unitarie[ ]*.

UnitPropagate ogni clausola unitaria (finchè il vettore non diventa vuoto) e scorre ogni elemento della matrice (fino a trovare “”) e opera nel seguente modo:

- Se trova un letterale uguale a quello da propagare, imposta tutti gli elementi della riga a “true”.

- Se trova un letterale opposto a quello da propagare, imposta il letterale a “false”.

Quando una riga viene modificata, viene chiamata una funzione *controlla_clausola( )* che controlla la clausola passata come parametro verificando se si è formata una nuova clausola unitaria o se si è trovato “BOX” (ossia tutte le clausole presenti sono false).

Nell'esempio precedente, facendo la unit propagate su !X1 si ottieme:

i/j	0	1	2	...	max_letterali
0	false	X2	!X3	“”	“”
1	**true**	**true**	**true**	**true**	**true**
2	**false**	X3	“”	“”	“”
3	**true**	**true**	**true**	**true**	**true**
...	“”	“”	“”	“”	“”
max_clausole	“”	“”	“”	“”	“”

Ed il vettore delle unitarie sarà: “X3”.

1.3.2. Splitting

Una volta terminata la unit propagate su tutti gli elementi presenti nel vettore *unitarie[ ]*, si esegue un controllo se tutte le variabili sono state assegnate:

- In caso positivo si passa alla visualizzazione della soluzione e delle statistiche.
- In caso negativo si esegue lo splitting su uno dei letterali rimasti in base all’euristica scelta.

Lo splitting viene eseguito inserendo il letterale scelto dall'euristica, nella prima riga vuota della matrice.

Ad esempio su una matrice:

i/j	0	1	2	...	max_letterali
0	X1	X2	!X3	“”	“”
1	!X1	!X2	!X3	“”	“”
2	X1	X3	“”	“”	“”
3	“”	“”	“”	“”	“”
...	“”	“”	“”	“”	“”
max_clausole	“”	“”	“”	“”	“”

Se l'eurisca di splitting ritorna ad esempio “X1” la matrice che verrà mandata alla chiamata ricorsiva (e mi ritornerà un valore booleano true o false) sarà:

i/j	0	1	2	...	max_letterali
0	X1	X2	!X3	“”	“”
1	!X1	!X2	!X3	“”	“”
2	X1	X3	“”	“”	“”
3	**X1**	“”	“”	“”	“”
...	“”	“”	“”	“”	“”
max_clausole	“”	“”	“”	“”	“”

messa in “or con la chiamata ricorsiva con la matrice:

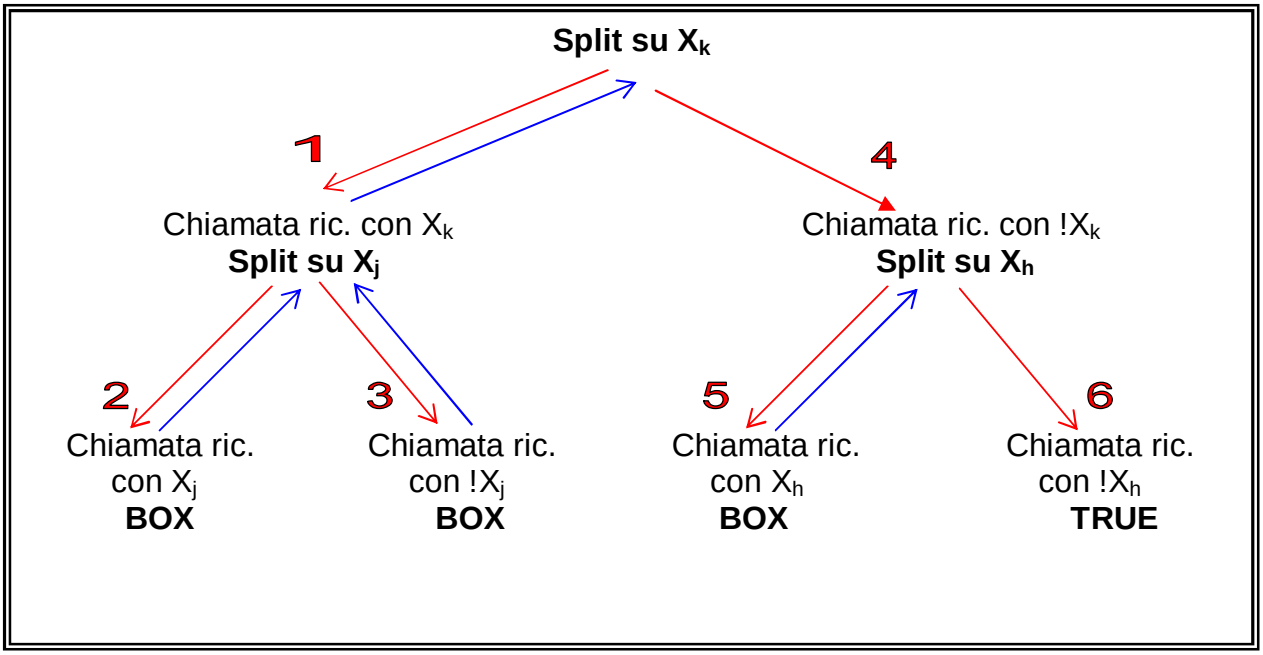
i/j	0	1	2	...	max_letterali
0	X1	X2	!X3	“”	“”
1	!X1	!X2	!X3	“”	“”
2	X1	X3	“”	“”	“”
3	“!X1”	“”	“”	“”	“”
...	“”	“”	“”	“”	“”
max_clausole	“”	“”	“”	“”	“”

1.3.3. BackTracking

L’algoritmo provede un’esecuzione delle istanze in profondità.

Se ad esempio la formula farà uno primo split su una variabile X_k e la prima chiamata ricorsiva (dove X_k verrà passata diretta) dovrà fare a sua volta un’ulteriore split ad esempio su X_j, allora verrà fatta prima la chiamata ricorsiva con quest’ultima variabile diretta ed indiretta, prima di quella con X_k indiretta.

Vediamo ad esempio cosa accade in una esecuzione che esegue due split che ha soluzione solo nello split eseguito con le variabili scelte non dirette.



Il backtracking è eseguito salvandosi ogni volta la matrice delle formule prima di aggiugere la variabile di splitting e della chiamata ricorsiva. In caso la chiamata ritorni “false”, ad ogni passo viene ripristinata la soluzione precedente ed eseguita la chiamata con la variabile negata.

1.4. Debug

Come descritto nella sezione 1.1 l'algoritmo vuole simulare l'esecuzione dell'algoritmo che normalmente una persona esegue a penna.

Per far questo sono state aggiunte molte righe di commenti nei quali si può notare le azioni chiave eseguite dall'applicazione ad ogni passo.

Per vedere tali righe basterà, una volta eseguito l'algoritmo, scorrere la shell verso l'alto fino a dove si è lanciata l'esecuzione.

1.4.1. Esempio di Debug e simulazione passo passo

Eseguiamo ora l'applicazione vedendo ed illustrando l'esecuzione attraverso le righe di debug. Consideriamo la formula "(X1 or X3),(!X2 or X3),(!X2 or !X3),(X3 or !X1),(!X1 or !X3)" con l'euristica "random distinct" (vedi 2.1):

```
DEBUG: esecuzione DPLL su formula: (X1 or X3),(!X2 or X3),(!X2 or !X3),(X3 or
!X1),(!X1 or !X3)
DEBUG: ____________________________________________________________________
DEBUG: ____________________Valorizzazione matrice_cnf____________________
DEBUG: matrix_cnf[0][0]=X1
DEBUG: matrix_cnf[0][1]=X3
DEBUG: matrix_cnf[1][0]=!X2
DEBUG: matrix_cnf[1][1]=X3
DEBUG: matrix_cnf[2][0]=!X2
DEBUG: matrix_cnf[2][1]=!X3
DEBUG: matrix_cnf[3][0]=X3
DEBUG: matrix_cnf[3][1]=!X1
DEBUG: matrix_cnf[4][0]=!X1
DEBUG: matrix_cnf[4][1]=!X3
DEBUG: _________________fine valorizzazione matrice_cnf_________________
```

Come accennato in 1.3 viene valorizzata ma matrice di stringhe.

```
DEBUG: ____________________________________________________________________
DEBUG: ________________________Inizio Unit Propagate________________________
DEBUG: numero clausole unitarie totale: 0
DEBUG: ________________________fine unit propagate________________________
DEBUG: la clausola 0 e' la prima che non ha tutti i letterali assegnati!
DEBUG: L'esecuzione quindi non è finita e devo utilizzare uno split.
DEBUG: salvo gli array per un eventuale backtrack
```

La Unit Progagate non trova clausole unitarie, allora si passa allo step successivo: viene verificato che tutte le variabili non siano già state assegnate, si passa quindi al salvataggio dell'istanza attuale (come descritto in 1.3.3) e, successivamente, allo split.

```
DEBUG: ____________________SPLITTING con euristica: 1____________________
DEBUG: Selezione letterale di splitting: euristica scelta: "random distinct"
DEBUG: Letterali tra i quali scegliere quello di splitting:
DEBUG: X2 X1 X3
DEBUG: letterale scelto: X2
DEBUG: impostato matrix_cnf[5][0] con il letterale scelto
```

Viene scelto il letterale "X2" in maniera randomica tra le 3 possibili scelte trovate e (come descritto in 1.3.2) nella prima riga libera del vettore, in questo caso la 5.

```
DEBUG: _____________________DPLL con nuova clausola: X2_____________________
DEBUG: ________________________________________________________________________
DEBUG: ________________________Inizio Unit Propagate________________________
DEBUG: numero clausole unitarie totale: 1
DEBUG: VETTORE UNITARIE : [ X2 ]
DEBUG: _________________Selezionato clausola unitaria: "X2"_________________
DEBUG: ________________________________________________________________________
DEBUG: matrix_cnf[1][0] = !X2 è opposto di X2 (let_selezionato)
DEBUG: impostato matrix_cnf[1][0] = false
DEBUG: Clausola 1 modificata! Inizio controllo
DEBUG: Trovato un letterale! sicuramente NON ho box
DEBUG: la clausola matrix_cnf[1] ora ha solo la clausola: X3
DEBUG: Impostato unitarie[1]=X3
DEBUG: ________________________________________________________________________
DEBUG: matrix_cnf[2][0] = !X2 è opposto di X2 (let_selezionato)
DEBUG: impostato matrix_cnf[2][0] = false
DEBUG: Clausola 2 modificata! Inizio controllo
DEBUG: Trovato un letterale! sicuramente NON ho box
DEBUG: la clausola matrix_cnf[2] ora ha solo la clausola: !X3
DEBUG: Impostato unitarie[2]=!X3
DEBUG: ________________________________________________________________________
DEBUG: Trovato matrix_cnf[5][0] == X2 (let_selezionato)
DEBUG: impostato matrix_cnf[5][0] = "true"
DEBUG: Clausola 5 modificata! Inizio controllo
DEBUG: matrix_cnf[5][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 5
DEBUG: SOLUZIONE PARZIALE: X2
DEBUG: VETTORE UNITARIE : [ X2 X3 !X3 ]
```

Adesso viene mandata la matrice alla chiamata ricorsiva con la variabile di split diretta e presente del vettore delle unitarie. Questa volta quindi viene eseguita la Unit Propagate.

Nell'esecuzine viene impostata i letterali di posizione [1][0] ed [2][0] a *false*, mentre la clausola 5 viene messa a *true*:

X1	X3	…	…	…
false	*X3*	…	…	…
false	*!X3*	…	…	…
X3	!X1	…	…	…
!X1	!X3	…	…	…
true	**true**	**true**	**true**	**true**
…	…	…	…	…

Da notare che, durante l'esecuzione, mettendo i letterali a false, vengono individuate due nuove clausole unitarie X3 e !X3 (le quali però ci fanno già intuire che troveremo box).

```
DEBUG: _________________Selezionato clausola unitaria: "X3"_________________
DEBUG: ____________________________________________________________________
DEBUG: Trovato matrix_cnf[0][1] == X3 (let_selezionato)
DEBUG: impostato matrix_cnf[0][0] = "true"
DEBUG: Clausola 0 modificata! Inizio controllo
DEBUG: matrix_cnf[0][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 0
DEBUG: ____________________________________________________________________
DEBUG: Trovato matrix_cnf[1][1] == X3 (let_selezionato)
DEBUG: impostato matrix_cnf[1][0] = "true"
DEBUG: Clausola 1 modificata! Inizio controllo
DEBUG: matrix_cnf[1][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 1
DEBUG: ____________________________________________________________________
DEBUG: matrix_cnf[2][1] = !X3 è opposto di X3 (let_selezionato)
DEBUG: impostato matrix_cnf[2][1] = false
DEBUG: Clausola 2 modificata! Inizio controllo
DEBUG: ***************** Trovato BOX! ****************
DEBUG: backtracking... ripristino stato precedente
```

La Unit Propagate prosegue con la clausola X3 ma, mettendo anche il letterale in [2][1] a *false*, tutta la clausola risulta falsa e si arriva al BOX con conseguente ripristino dello stato precedente prima dello split.

```
DEBUG: ____________________________________________________________________
DEBUG: ____________________Inizio Unit Propagate_____________________
DEBUG: numero clausole unitarie totale: 1
DEBUG: VETTORE UNITARIE : [ !X2 ]
DEBUG: _________________Selezionato clausola unitaria: "!X2"_________________
DEBUG: ____________________________________________________________________
DEBUG: Trovato matrix_cnf[1][0] == !X2 (let_selezionato)
DEBUG: impostato matrix_cnf[1][0] = "true"
DEBUG: Clausola 1 modificata! Inizio controllo
DEBUG: matrix_cnf[1][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 1
DEBUG: ____________________________________________________________________
DEBUG: Trovato matrix_cnf[2][0] == !X2 (let_selezionato)
DEBUG: impostato matrix_cnf[2][0] = "true"
DEBUG: Clausola 2 modificata! Inizio controllo
DEBUG: matrix_cnf[2][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 2
DEBUG: ____________________________________________________________________
DEBUG: Trovato matrix_cnf[5][0] == !X2 (let_selezionato)
DEBUG: impostato matrix_cnf[5][0] = "true"
DEBUG: Clausola 5 modificata! Inizio controllo
DEBUG: matrix_cnf[5][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 5
DEBUG: SOLUZIONE PARZIALE: !X2
DEBUG: ____________________fine unit propagate_____________________
```

Si esegue allora la chiamata ricorsiva nella quale viene inserito nella matrice il letterale di splitting negato. Si esegue quindi la Unit Propagate che termina questa volta senza individuare nessuna nuova clausola unitaria e nessun box.

Attualmente la matrice è così composta:

X1	X3	…	…	…
true	true	true	true	true
true	true	true	true	true
X3	!X1	…	…	…
!X1	!X3	…	…	…
true	true	true	true	true
…	…	…	…	…

```
DEBUG: la clausola 0 e' la prima che non ha tutti i letterali assegnati!
DEBUG: L'esecuzione quindi non è finita e devo utilizzare uno split.
DEBUG: salvo gli array per un eventuale backtrack
```

Non tutte le variabili risultano assegnate, quindi è necessario un altro split.

```
DEBUG: _____________________SPLITTING con euristica: 1_____________________
DEBUG: Selezione letterale di splitting: euristica scelta: "random distinct"
DEBUG: Letterali tra i quali scegliere quello di splitting:
DEBUG: X1 X3
DEBUG: letterale scelto: X1
DEBUG: impostato matrix_cnf[6][0] con il letterale scelto
DEBUG: _____________________DPLL con nuova clausola: X1_____________________
DEBUG: ________________________________________________________________________
DEBUG: ___________________________Inizio Unit Propagate___________________________
DEBUG: numero clausole unitarie totale: 1
DEBUG: VETTORE UNITARIE : [ X1 ]
DEBUG: _____________Selezionato clausola unitaria: "X1"_____________
DEBUG: ________________________________________________________________________
DEBUG: Trovato matrix_cnf[0][0] == X1 (let_selezionato)
DEBUG: impostato matrix_cnf[0][0] = "true"
DEBUG: Clausola 0 modificata! Inizio controllo
DEBUG: matrix_cnf[0][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 0
DEBUG: ________________________________________________________________________
DEBUG: matrix_cnf[3][1] = !X1 è opposto di X1 (let_selezionato)
DEBUG: impostato matrix_cnf[3][1] = false
DEBUG: Clausola 3 modificata! Inizio controllo
DEBUG: Trovato un letterale! sicuramente NON ho box
DEBUG: la clausola matrix_cnf[3] ora ha solo la clausola: X3
DEBUG: Impostato unitarie[1]=X3
DEBUG: ________________________________________________________________________
DEBUG: matrix_cnf[4][0] = !X1 è opposto di X1 (let_selezionato)
DEBUG: impostato matrix_cnf[4][0] = false
DEBUG: Clausola 4 modificata! Inizio controllo
DEBUG: Trovato un letterale! sicuramente NON ho box
DEBUG: la clausola matrix_cnf[4] ora ha solo la clausola: !X3
DEBUG: Impostato unitarie[2]=!X3
DEBUG: ________________________________________________________________________
DEBUG: Trovato matrix_cnf[6][0] == X1 (let_selezionato)
DEBUG: impostato matrix_cnf[6][0] = "true"
DEBUG: Clausola 6 modificata! Inizio controllo
DEBUG: matrix_cnf[6][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 6
```

```
DEBUG: SOLUZIONE PARZIALE: !X2 X1
DEBUG: VETTORE UNITARIE : [ X1 X3 !X3 ]
DEBUG: _____________Selezionato clausola unitaria: "X3"_____________
DEBUG: ____________________________________________________________
DEBUG: Trovato matrix_cnf[3][0] == X3 (let_selezionato)
DEBUG: impostato matrix_cnf[3][0] = "true"
DEBUG: Clausola 3 modificata! Inizio controllo
DEBUG: matrix_cnf[3][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 3
DEBUG: ____________________________________________________________
DEBUG: matrix_cnf[4][1] = !X3 è opposto di X3 (let_selezionato)
DEBUG: impostato matrix_cnf[4][1] = false
DEBUG: Clausola 4 modificata! Inizio controllo
DEBUG: ********** Trovato BOX! **********
DEBUG: backtracking... ripristino stato precedente
DEBUG: ____________________________________________________________
```

Viene scelta in maniera randomica la variabile X1 che però, similmente a quando visto prima ci porta ad un BOX e quindi al backtrack con la chiamata ricorsiva questa volta con “!X1”.

```
DEBUG: _____________________Inizio Unit Propagate_____________________
DEBUG: numero clausole unitarie totale: 1
DEBUG: VETTORE UNITARIE : [ !X1 ]
DEBUG: _____________Selezionato clausola unitaria: "!X1"_____________
DEBUG: ____________________________________________________________
DEBUG: matrix_cnf[0][0] = X1 è opposto di !X1 (let_selezionato)
DEBUG: impostato matrix_cnf[0][0] = false
DEBUG: Clausola 0 modificata! Inizio controllo
DEBUG: Trovato un letterale! sicuramente NON ho box
DEBUG: la clausola matrix_cnf[0] ora ha solo la clausola: X3
DEBUG: Impostato unitarie[1]=X3
DEBUG: ____________________________________________________________
DEBUG: Trovato matrix_cnf[3][1] == !X1 (let_selezionato)
DEBUG: impostato matrix_cnf[3][0] = "true"
DEBUG: Clausola 3 modificata! Inizio controllo
DEBUG: matrix_cnf[3][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 3
DEBUG: ____________________________________________________________
DEBUG: Trovato matrix_cnf[4][0] == !X1 (let_selezionato)
DEBUG: impostato matrix_cnf[4][0] = "true"
DEBUG: Clausola 4 modificata! Inizio controllo
DEBUG: matrix_cnf[4][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 4
DEBUG: ____________________________________________________________
DEBUG: Trovato matrix_cnf[6][0] == !X1 (let_selezionato)
DEBUG: impostato matrix_cnf[6][0] = "true"
DEBUG: Clausola 6 modificata! Inizio controllo
DEBUG: matrix_cnf[6][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 6
DEBUG: SOLUZIONE PARZIALE: !X2 !X1
DEBUG: VETTORE UNITARIE : [ !X1 X3 ]
DEBUG: _____________Selezionato clausola unitaria: "X3"_____________
DEBUG: ____________________________________________________________
DEBUG: Trovato matrix_cnf[0][1] == X3 (let_selezionato)
DEBUG: impostato matrix_cnf[0][0] = "true"
DEBUG: Clausola 0 modificata! Inizio controllo
DEBUG: matrix_cnf[0][0] = true. Imposto tutta la clausola a true
DEBUG: NON ho trovato box nella clausola 0
DEBUG: SOLUZIONE PARZIALE: !X2 !X1 X3
DEBUG: _____________________fine unit propagate_____________________
```

```
DEBUG: tutte le variabili sono state assegnate!
DEBUG: Soluzione formula: !X2 !X1 X3
DEBUG: ____________________________________________________________________________
DEBUG: Fine esecuzone DPLL
```

Questa volta l'esecuzione va a buon fine, si riesce ad assegnare *true* a tutte le variabili ottenendo così la soluzione "*!X2 !X1 X3*" che è facile verificare essere valida nella formula iniziale.

```
____________________________________________________________________________
La formula: (X1 or X3),(!X2 or X3),(!X2 or !X3),(X3 or !X1),(!X1 or !X3)
-e' soddisfacibile!
-una sua soluzione e': !X2 !X1 X3
____________________________________________________________________________
Algoritmo DPLL:
-euristica scelta letterale di split:    "random distinct"
-numero operazioni di split eseguite:     2
-tempo di esecuzione impiegato (sec):     0.009411
____________________________________________________________________________
Premi invio per tornare al menu'...
```

Viene messa a video una schermata di riepilogo che mostra che la formula è soddisfacibile con la soluzione trovata, e le statistiche tempo/split ottenute nell'esecuzione appena terminata.

2. Euristiche

Di seguito saranno illustrate le varie euristiche implementate nel progetto.

2.1. Random Distinct

Questa euristica scorre tutti i letterali salvandoseli in maniera distinta in un vettore. In pratica ogni letterale trovato, viene prima eventualmente convertito in diretto (se questo era negato), e poi inserito nel vettore solo se non era già presente.

Una volta salvato il vettore viene gestita casualmente la scelta tra i suoi elementi.

2.1.1. Esempio

Sia la matrice la seguente:

X1	X3	…
!X2	X3	…
!X2	!X3	…
X3	!X1	…
!X1	!X3	…
…	…	…

Allora il vettore risultate su cui eseguire la scelta sarà: “[X1, X2, X3]”.

2.2. Random Pesato

Similmente al precedente, scorre tutti gli elementi, questa volta però salvandoli (in maniera diretta) tante volte quante compaiono.

Una volta salvato il vettore viene gestita casualmente la scelta tra i suoi elementi. In questo caso sarà quindi più probabile che la variabile scelta sia una che compare molte volte nella matrice.

2.2.1. Esempio

Sia la matrice quella vista in precedenza. Questa volta il vettore sul quale verrà scelto l’elemento casualmente sarà: “[X1, X3, X2, X3, X2, X3, X3, X1, X1, X3]”.

2.3. Ricerca Pure literal

Questa euristica inizialmente scorre ogni elemento salvandoselo in un primo vettore in maniera distinta ma contanto anche le riccorrenze di ogni letterale. Una volta finita questa operazione scorre tutti gli elementi di quest'ultimo vettore verificando, per ogni elemento, se questo compare solo con il suo segno (diretto o negato che sia).

Se vengono trovati più elementi puri, viene ritornato casualmente un elemento fra i puri.

Se non vengono trovati elementi puri l'euristica sviluppata prevede di ritornare l'elemento che ha più ricorrenze (trasformato in diretto, nel caso questo sia negato)..

2.3.1. Esempio

Sia la matrice quella vista in precedenza. Il "primo" vettore questa volta salverà "[X1, X3, !X2, !X3, !X1]" con le relative ricorrenze "[1, 3, 2, 2, 2]".

Scorrendo a sua volta questo vettore si trova che X1 ed X3 (coi relativi opposti) non sono puri, mentre !X2 compare solamente negato. Questa euristica tornerà quindi come letterale scelto "!X2".

Se matrice precedente avesse avuto "X2" al primo posto della seconda riga invece che "!X2". L'algoritmo calcolerebbe il primo vettore "[X1, X3, X2, !X2, !X3, !X1]" con le relative ricorrenze "[1, 3, 1, 1, 2, 2]". In questo caso l'elemento ritornato sarà X3 che è quello con più ricorrenze (ed a parità di numero verrà tornato il primo selezionato).

2.4. Letterale più presente in clausole più piccole

Quest'euristica scorre una prima volta la matrice contando i letterali ed individuando la dimensione minima delle clausole.

Dopo ciò, per ogni clausola che ha la dimensione minima si comporta similmente all'euristica precedente, salvandosi in un primo vettore le variabili distinte (ma solamente dirette) e le loro ricorrenze.

Infine ritorna il letterale con maggiori ricorrenze e, a parità di ricorrenze, il primo selezionato.

2.4.1. Esempio

Sia la matrice quella vista in precedenza. Il primo ciclo individua in 2 la dimensione minima della clausole. Il primo vettore questa volta salverà "[X1, X2, X3]" con le relative ricorrenze "[3, 2, 5]".

Verrà quindi ritornato il letterale X3.

3. Benchmark

Lo scopo dei benchmark è quello di provare a mettere a confronto le quattro euristiche su istanze diverse e vedere quali “in generale” si comportano meglio delle altre.

3.1. Sistema e variabili di benchmark

I prossimi benchmark sono basati sul seguente sistema (presente nel mio portatile):

- *Processore: intel® Core™2 Duo avente due CPU T5250 @ 1,50 GHz.*
- *Memoria: 2061MB*
- *Sistema Operativo: Ubuntu 9.10 (karmic), kernel Linux 2.6.31-19-generic, GNOME 2.28.1*

Come variabili per la valutazione utilizzeremo il tempo ed il numero di split eseguiti. Verranno eseguite in totale 10 prove per ogni instanza e si terrà il valore medio.

3.2. Istanza n°1

La prima istanza sulla quale eseguiremo il benchmark è la formula precedentemente descritta come esempio, ossia:

“(X1 or X3),(!X2 or X3),(!X2 or !X3),(X3 or !X1),(!X1 or !X3)”

L’instanza è abbastanza piccola, è soddisfacibile e il numero di split può variare tra un minimo di 1 ed un massimo di 2.

3.2.1. Random Distinct

Ci si aspetta che gli split varino essendoci la scelta casuale, e che i casi quando sono 2 siano circa 1/3 = 33% delle prove, mentre un solo split dovrebbe evvenire in circa nel 66% delle prove.

#	TEMPO (ms)	SPLIT
1	6,443	1
2	9,462	2
3	9,059	2
4	6,592	1
5	4,378	1
6	4,354	1
7	4,343	1
8	6,635	1
9	6,557	1
10	4,361	1

Tempo Medio:	6,2184
# Split Medio:	1,2

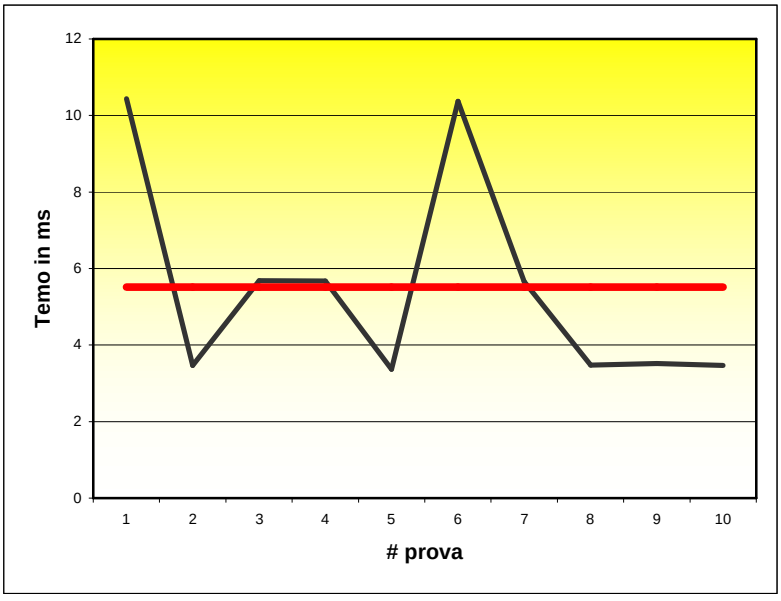
Come si può vedere il tempo si alza quando si devono usare gli 2 split. Il numero di prove con 2 split è stato leggermente minore rispetto alle aspettative.

3.2.2. Random Pesato

Ci si aspetta che il numero di volte che lo split viene eseguito su X2 (variabile che porta al secondo split) sia ancora minore e pari circa al 20% delle prove.

#	TEMPO (ms)	SPLIT
1	10,435	2
2	3,469	1
3	5,685	1
4	5,672	1
5	3,364	1
6	10,370	2
7	5,655	1
8	3,473	1
9	3,522	1
10	3,469	1

Tempo Medio:	5,5114
# Split Medio:	1,2



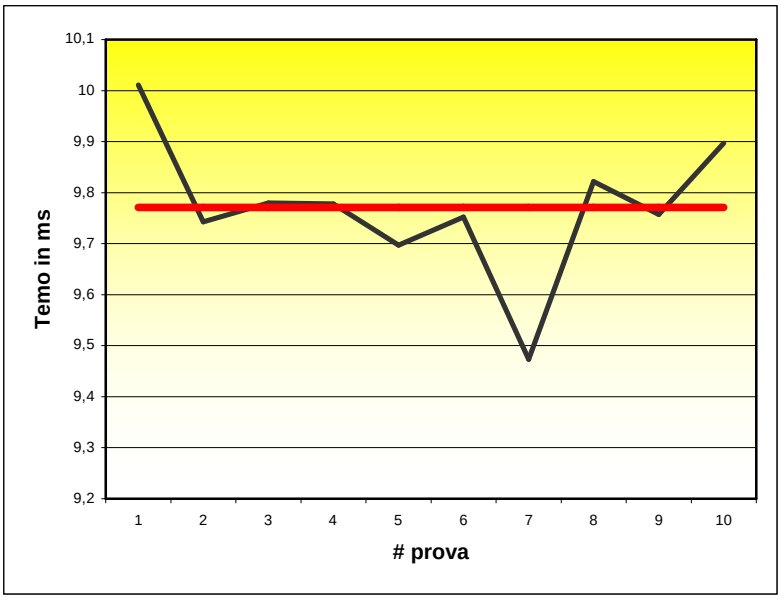
In questo esperimento si nota che il tempo d'esecuzione quando si utilizzano due split, risulta essere ancora maggiore del precedente, ma che mediamente il tempo è minore.

3.2.3. Ricerca Letterale Puro

Questa istanza non è molto buona per questa euristica in quanto *il letterale puro* ci seleziona sempre !X2 come variabile di splitting; tale variabile ci costringe sempre ad eseguire il secondo split. Anche questo però viene scelto deterministicamente (in base all'istanza) quindi ci si attende che i dati siano molto simili dal punto di vista anche del tempo.

#	TEMPO (ms)	SPLIT
1	10,011	2
2	9,743	2
3	9,780	2
4	9,778	2
5	9,697	2
6	9,752	2
7	9,473	2
8	9,822	2
9	9,757	2
10	9,897	2

Tempo Medio:	9,771
# Split Medio:	2



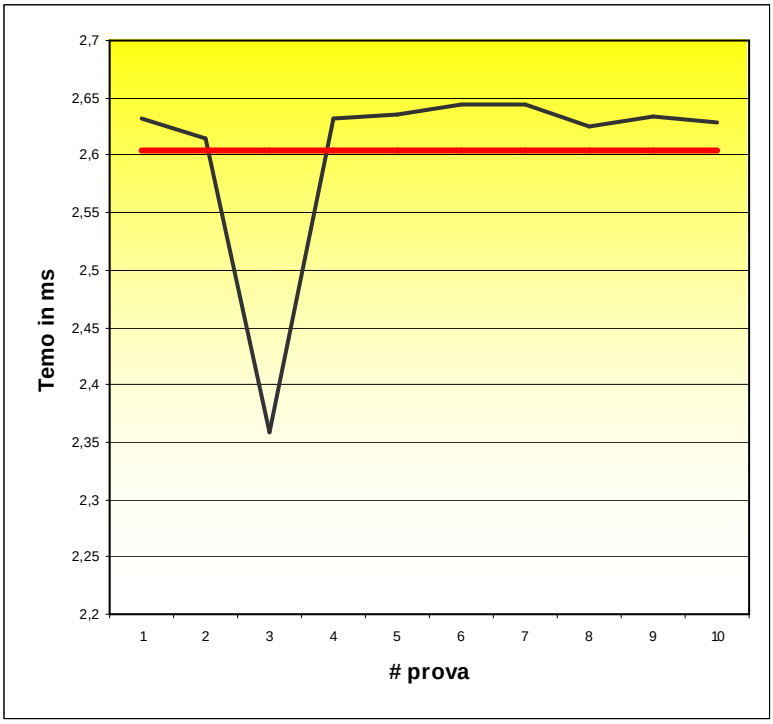
Come si vede questa formula non costituisce una buona istanza per questo tipo di risoluzione ed il tempo di esecuzione è molto simile, come ci si aspettava.

3.2.4. Ricerca Letterale Più Presente

Anche in questo caso l'esecuzione dovrebbe seguire una linea deterministica in base all'istanza. Il letterale più presente nella clausole più piccole (grandezza = 2) è X3. Quindi ci si aspetta di eseguire un solo splitting.

#	TEMPO (ms)	SPLIT
1	2,632	1
2	2,615	1
3	1,759	1
4	2,632	1
5	2,635	1
6	2,644	1
7	2,645	1
8	2,625	1
9	2,633	1
10	2,629	1

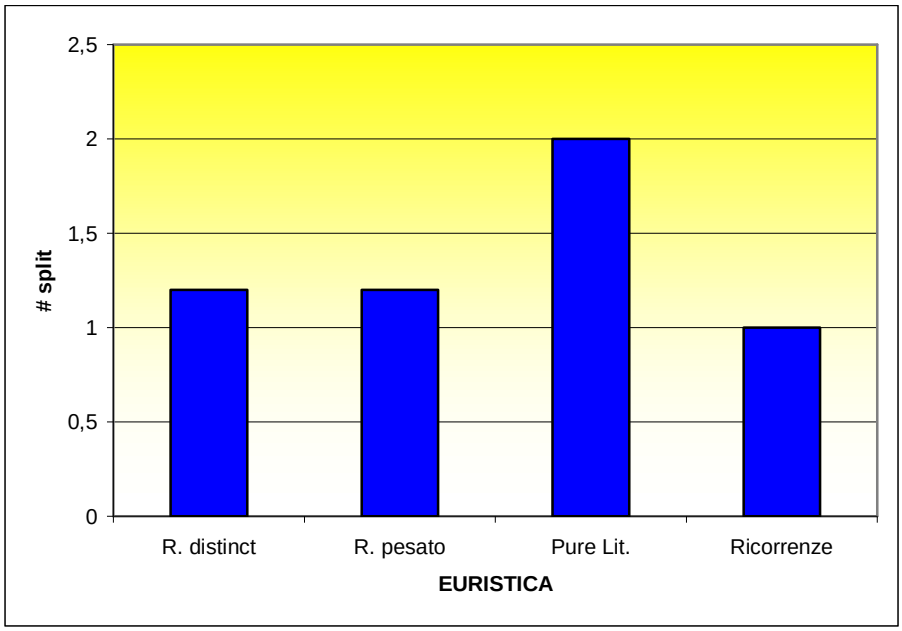
Tempo Medio:	2,5449
# Split Medio:	1



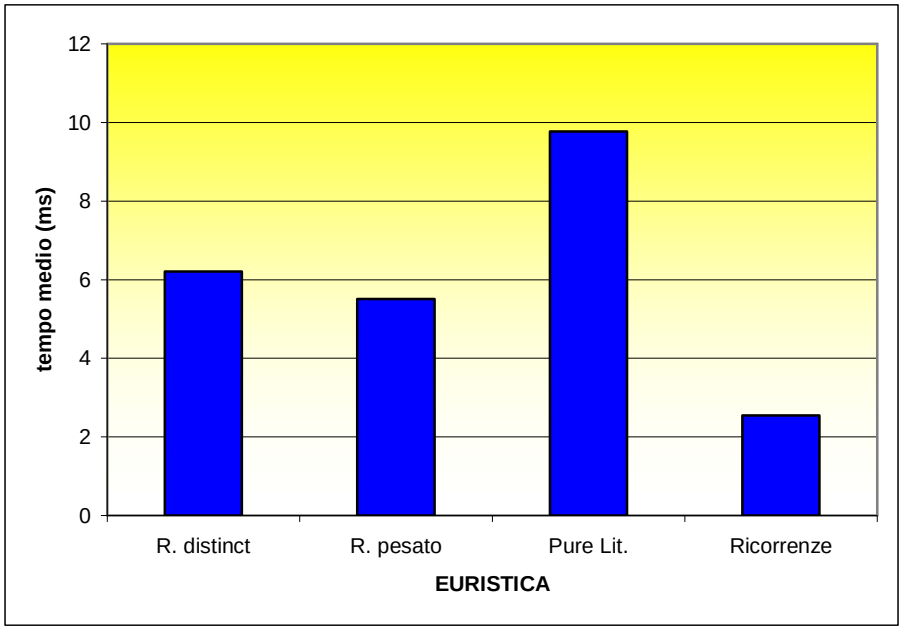
Anche qui i risultati sono quelli aspettati. Tra l'altro si vede che questa euristica sembra essera la migliore per questa instanza.

3.2.5. Confronto Euristiche

Se guardiamo seplicemente il numero di Split medio, vediamo già come l'ultima euristica (denominata "Ricorrenze") in questa istanza funzioni molto bene:



Il risultato è ancora più evidente se si guardano i tempi medi d'esecuzione, benchè questi siano solitamente legati al numero di split:



3.3. Istanza n°2

La seconda istanza sulla quale eseguiremo il benchmark è presente il “formule.dat” e caricabile quindi direttamente dal programma:

(X1 or X2 or X3 or !X4),(X1 or X2 or !X3 or X4),(X1 or !X2 or X3 or X4),

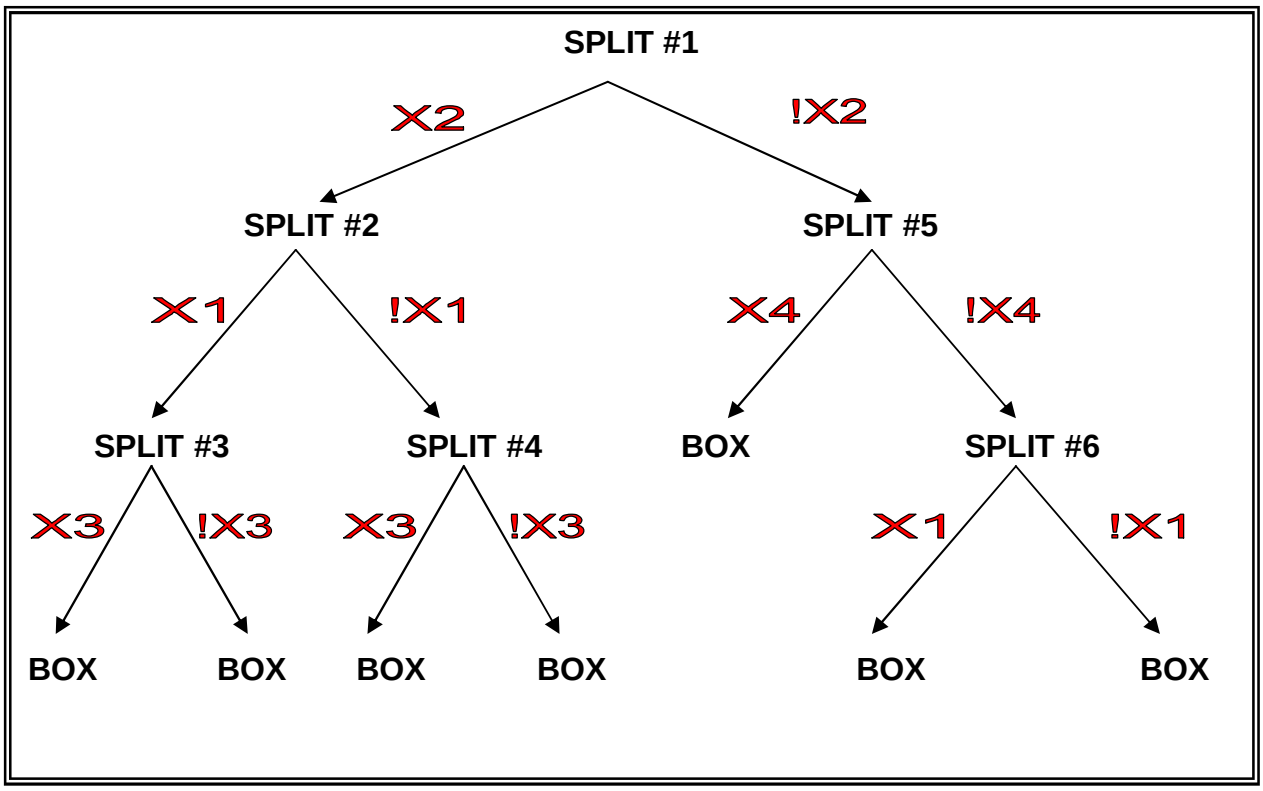
(!X1 or X2 or X3 or X4),(!X3 or !X4),(!X2 or !X3 or X4),

(!X2 or X3 or !X4),(!X1 or X2 or !X3 or X4),

(!X1 or !X2 or X3 or X4),(!X1 or X2 or X3 or !X4),(X1 or X2 or X3)

L’instanza è già più grande della precedente ed è non soddisfacibile. Si possono avere da un minimo di 2 split ad un massimo di 7 split.

Ad esempio una istanza con 6 split (presa da una esecuzione) fa tutti i seguenti split-backtrack prima di trovare box:

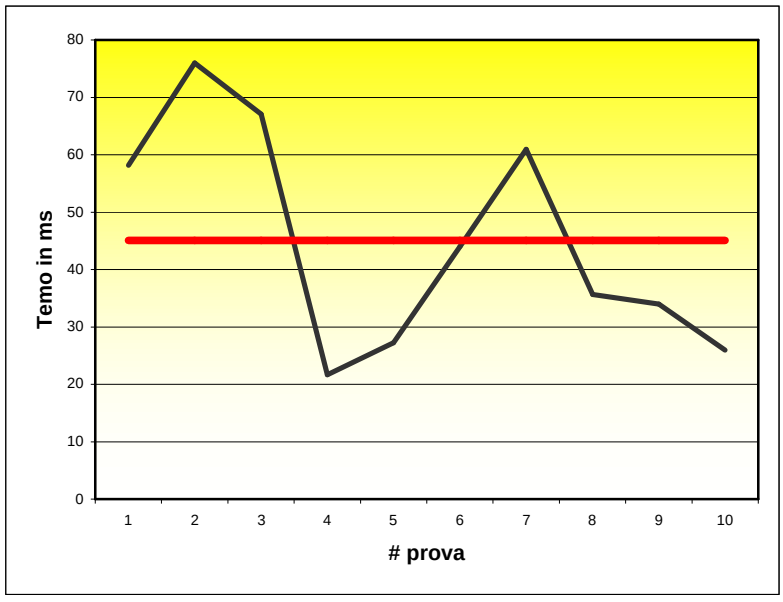


3.3.1. Random Distinct

Il fattore dello split random può far ottente risultati molto diversi in quanto, prima di determinare l'insoddisfacibilità, si possono eseguire split su variabili "sfortunate" che ci portano più tardi al box.

#	TEMPO (ms)	SPLIT
1	58,164	4
2	76,006	4
3	67,091	6
4	21,654	2
5	27,258	3
6	44,008	4
7	60,929	3
8	35,636	3
9	33,986	3
10	25,974	3

Tempo Medio:	45,0706
# Split Medio:	3,5



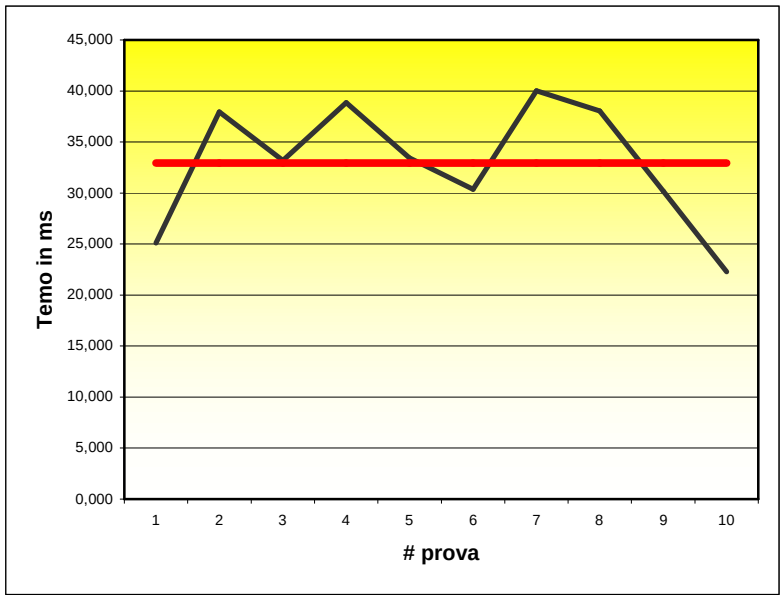
Come aspettato il numero di split, come il tempo di esecuzione, varia molto. I casi molto "fortunati" (2 split) e "molto sfortunati" (6 split), come ci si aspettava sono più improbabili.

3.3.2. Random Pesato

Il random pesato, in questa istanza, dovrebbe avere un leggero vantaggio rispetto al normale in quanto è più probabile che venga scelta una variabile tra X3 ed X4 (che ci fanno trovare prima il box) in quanto più presenti.

#	TEMPO (ms)	SPLIT
1	25,090	3
2	37,949	4
3	33,188	4
4	38,854	3
5	33,450	5
6	30,375	4
7	40,023	6
8	38,057	4
9	30,181	4
10	22,295	2

Tempo Medio:	32,9462
# Split Medio:	3,9



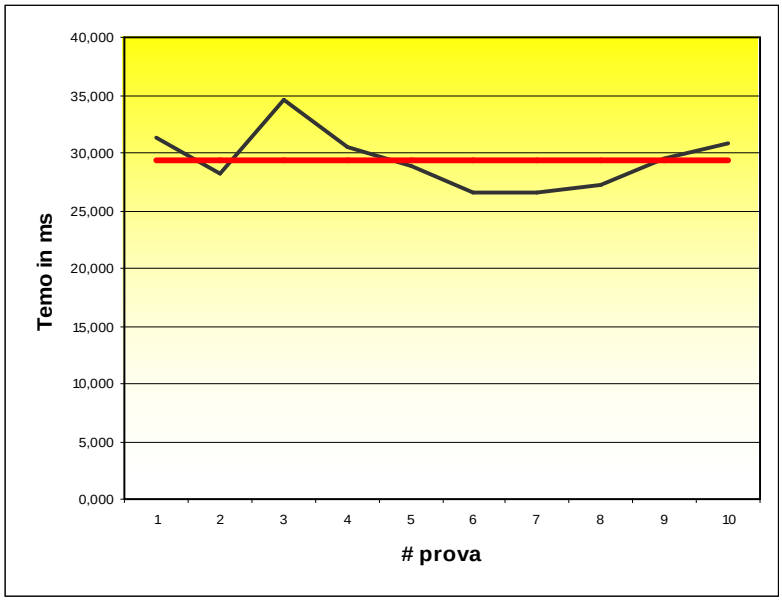
I risultati sembrano anche migliori di quando previsto; il tempo medio d'esecuzione è notevolmente ridotto rispetto al random distinto. Il numero di split medio però, al contrario di quello che si pensava, non è risultato minore (ma può essere dovuto al piccolo numero di esperimenti).

3.3.3. Ricerca Letterale Puro

Al contrario della precedente istanza, qui il letterale puro funziona discretamente in quanto l'euristica prevede, in caso non si trovi un puro, di ritornare come lettere di split, quello col maggior numero di ricorrenze ossia X3. che porta a non avere molti split rispetto al minimo.

#	TEMPO (ms)	SPLIT
1	31,392	3
2	28,155	3
3	34,632	3
4	30,528	3
5	28,877	3
6	26,583	3
7	26,487	3
8	27,236	3
9	29,486	3
10	30,869	3

Tempo Medio:	29,4245
# Split Medio:	3



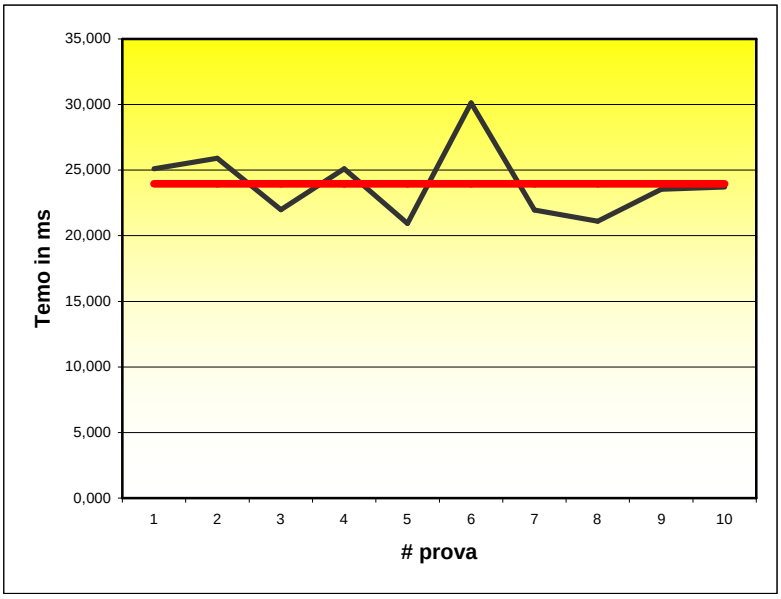
Come si vede questa euristica funziona bene con questa istanza; il tempo medio è ancora diminuito rispetto al random pesato, ed il numero di split medio è ulteriormente abbassato e fisso a 3.

3.3.4. Ricerca Letterale Più Presente

Anche in questo caso l'esecuzione dovrebbe seguire una linea deterministica in base all'istanza. Anche in questo caso, l'euristica risulta molto buona ed eseguirà il minor numero di split possibile in quando splitterà prima su X3 e successivamente su X1.

#	TEMPO (ms)	SPLIT
1	25,104	2
2	25,901	2
3	21,995	2
4	25,104	2
5	20,945	2
6	30,127	2
7	21,970	2
8	21,116	2
9	23,556	2
10	23,724	2

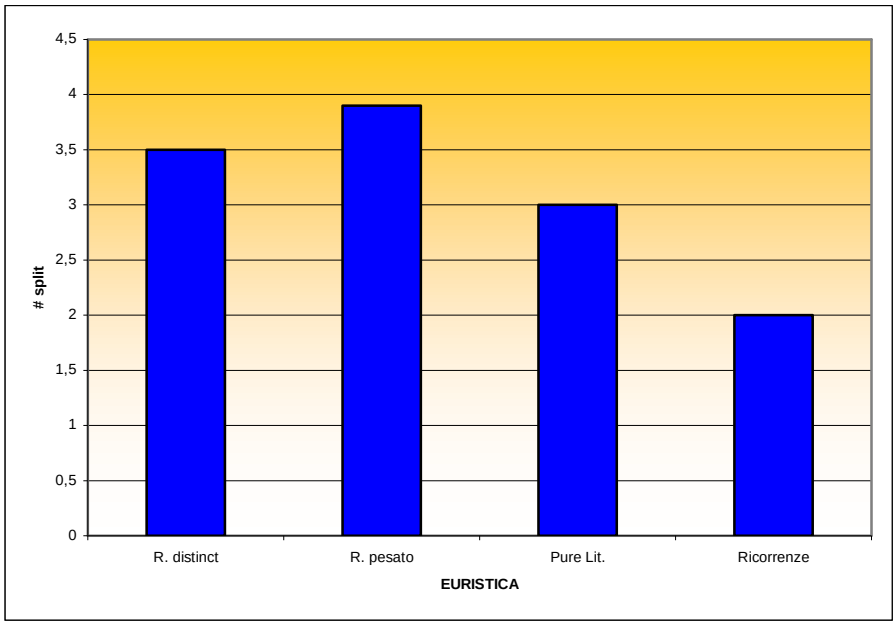
Tempo Medio:	23,9542
# Split Medio:	2



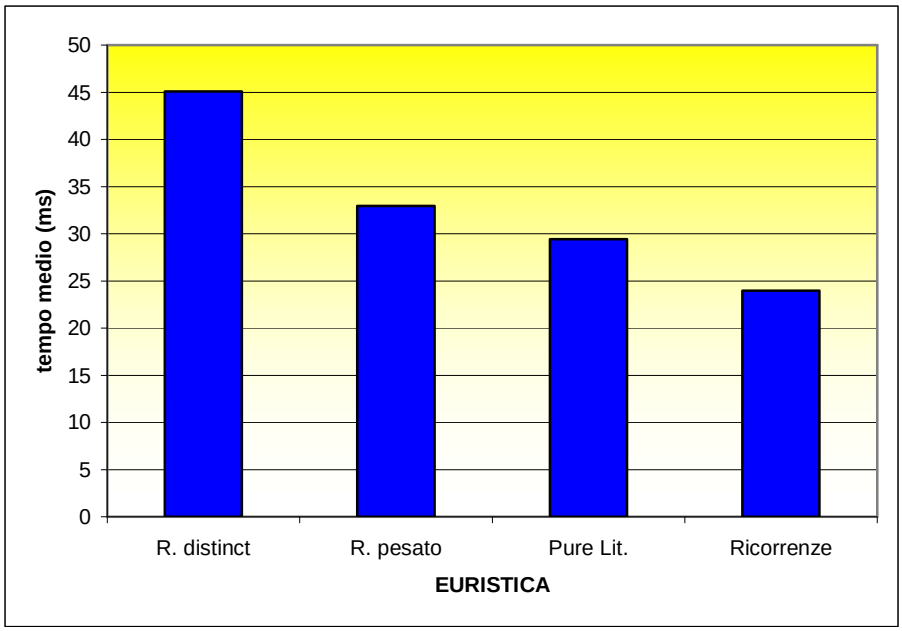
Anche qui i risultati sono quelli aspettati. Il numero di split fissato al minimo fa si che anche il tempo di esecuzione sia il più basso di tutti. Anche con questa istanza questa euristica sembra essere la migliore delle quattro proposte.

3.3.5. Confronto Euristiche

Anche in questa istanza se si guardano il numero di split, l'euristica che trova il letterale più ricorrente nelle clausole più piccole, vince nettamente il confronto con le altre.



Ed i tempi di esecuzione danno comunque conferma del vantaggio che, anche in questa istanza, porta questa euristica.



3.4. Istanza n°2

La terza ed ultima istanza è una del "*pigeon hole problem*" trovabile su sito http://people.cs.ubc.ca/~hoos/SATLIB/benchm.html .

L'istanza è stata leggermente ridotta per ovviare ad un problema del compilatore (problema ancora da fixare e che non mi permette di inserire più di 100 clausole o 100 letterali) ma rispetta il modello precedente.

Per eseguirla bisogna caricarla tramile l'opzione "3 = carica formula da file *.cnf" ed inseire "hole.cnf" che è presente nella directory del programma. L'istanza è decisamente più grande delle altre e possiede 28 clausole con ben 20 letterali distinti.

La formula è insoddisfacibile ed è rappresentata dal seguente testo:

```
p cnf 28 45
-1     -7    0
-1     -13   0
-1     -19   0
-1     -25   0
-7     -13   0
-7     -19   0
-7     -25   0
-13    -19   0
-13    -25   0
-19    -25   0
-2     -8    0
-2     -14   0
-2     -20   0
-2     -26   0
-8     -14   0
-8     -20   0
-8     -26   0
-14    -20   0
-14    -26   0
-20    -26   0
-3     -9    0
-3     -15   0
-3     -21   0
-3     -27   0
-9     -15   0
-9     -21   0
-9     -27   0
-15    -21   0
-15    -27   0
-21    -27   0
-4     -10   0
-4     -16   0
-4     -22   0
-4     -28   0
-10    -16   0
-10    -22   0
-10    -28   0
-16    -22   0
-16    -28   0
-22    -28   0
 4      3      2      1    0
 10     9      8      7    0
 16     15     14     13   0
 22     21     20     19   0
 28     27     26     25   0
```

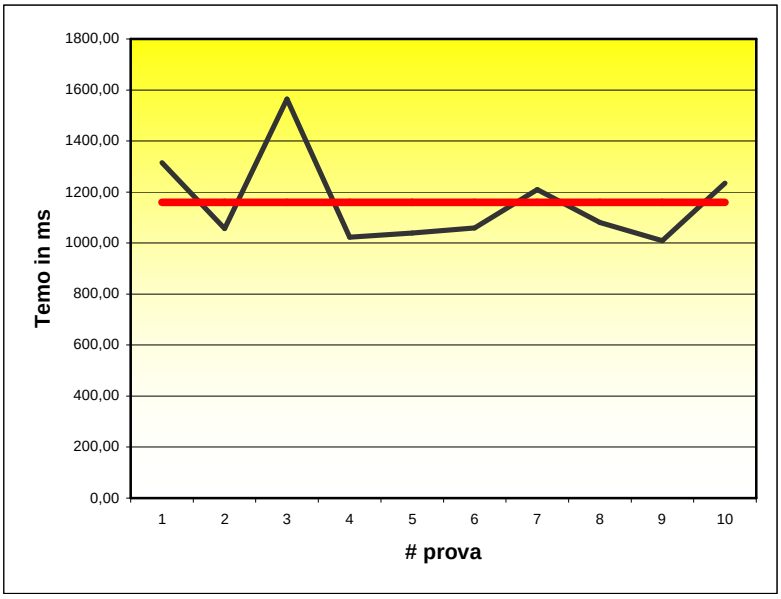
Dove ogni riga rappresenta una clausola e ad esempio "-1 -7 0" viene tradotto nel programma in "!X1 or !X7".

3.4.1. Random Distinct

I tempi d'esecuzione sono decisamente aumentati rispetto alle istanze precedenti e, come sempre con le euristiche che introducono una scelta randomica, ci si aspetta varino di esecuzione in esecuzione.

#	TEMPO (ms)	SPLIT
1	1314,83	40
2	1056,23	29
3	1564,96	46
4	1023,39	27
5	1039,35	29
6	1059,06	35
7	1209,21	38
8	1080,59	31
9	1009,19	29
10	1234,62	34

Tempo Medio:	1159,143
# Split Medio:	33,8

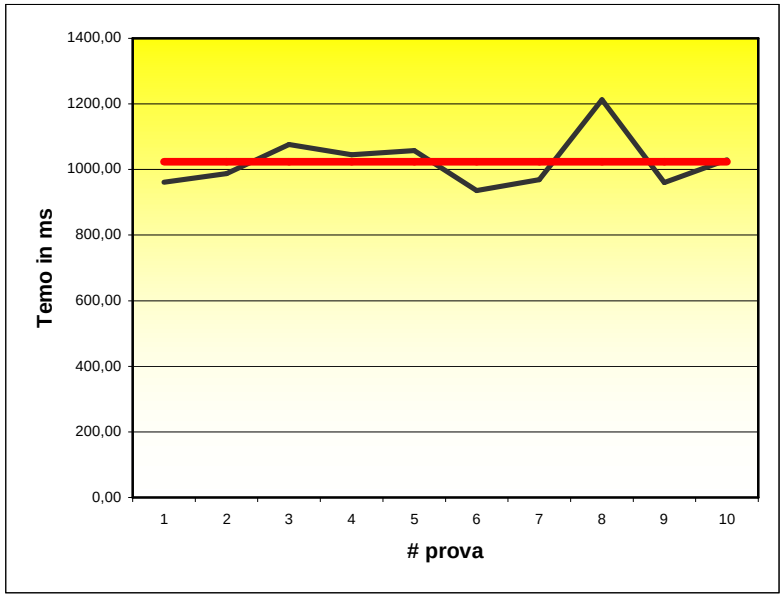


In questa istanza si vede ancora di più come questa scelta possa portare a risultati nettamente diversi (esperimento #4 con soli 27 split, mentre esperimento #3 con ben 46 split).

3.4.2. Random Pesato

#	TEMPO (ms)	SPLIT
1	961,15	29
2	987,18	26
3	1076,04	29
4	1044,87	33
5	1058,02	29
6	935,45	27
7	968,90	28
8	1212,94	37
9	960,31	33
10	1029,90	30

Tempo Medio:	1023,476
# Split Medio:	30,1



A parte l'esecuzione numero 8, i tempi sono decisamente abbassati rispetto al random distinct. Anche il numero di split medio risulta leggermente minore.

3.4.3. Ricerca Letterale Puro

#	TEMPO (ms)	SPLIT
1	757,68	36
2	806,97	36
3	787,33	36
4	794,31	36
5	799,96	36
6	816,98	36
7	796,83	36
8	762,97	36
9	809,01	36
10	791,80	36

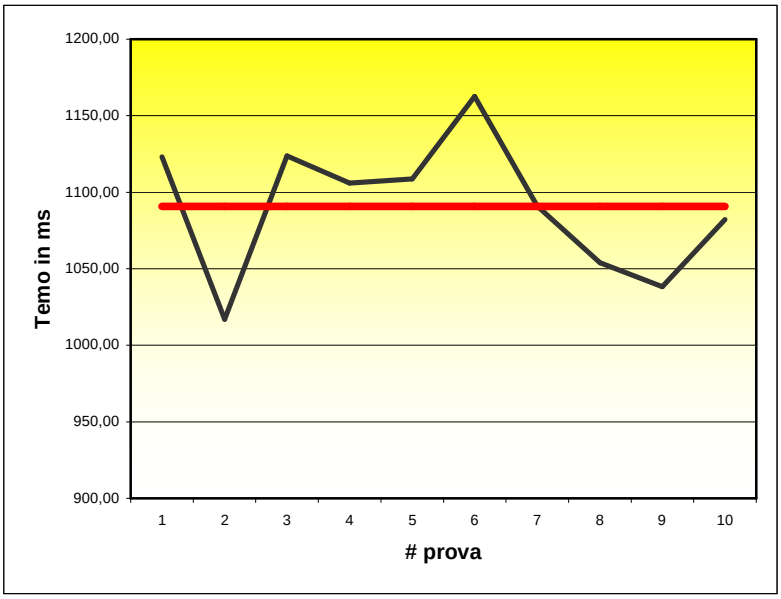
Tempo Medio:	792,3835
# Split Medio:	36

Il numero di split è ovviamente fermo (in quanto l'euristica è deterministica in base all'istanza), ma quello che sorprende è che il tempo medio d'esecuzione è comunque molto basso pur eseguendo ogni volta 36 split, prima di poter dire che l'istanza è insoddisfacibile. Il fatto sembra comunque dovuto al caso in quando non vi sono letterali puri nell'istanza ed il numero di ricorrenze di ogni letterale è sempre 2.

3.4.4. Ricerca Letterale Più Presente

#	TEMPO (ms)	SPLIT
1	1123,20	35
2	1016,96	35
3	1123,84	35
4	1105,94	35
5	1108,80	35
6	1162,54	35
7	1091,04	35
8	1054,04	35
9	1038,33	35
10	1082,14	35

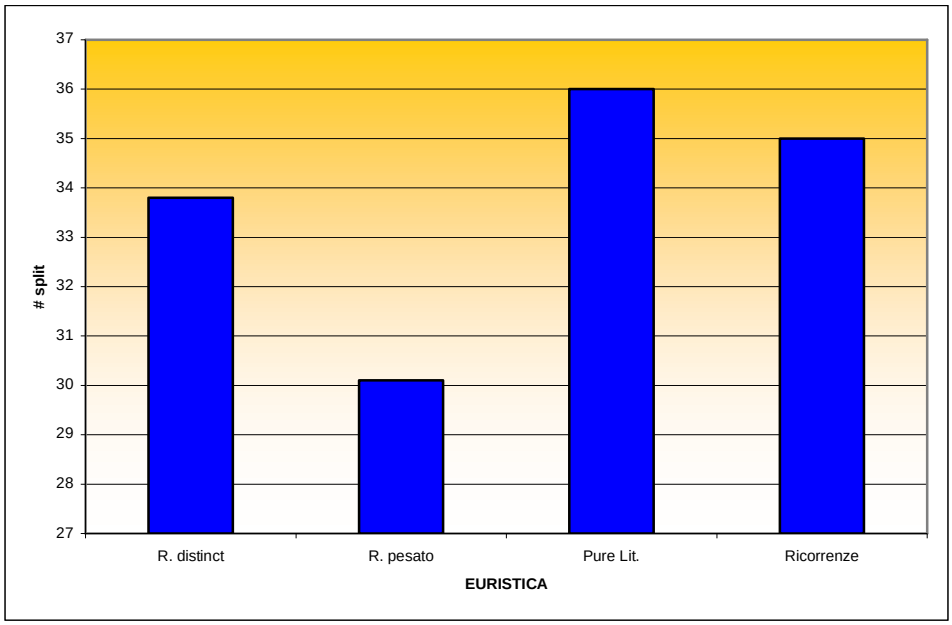
Tempo Medio:	1090,683
# Split Medio:	35



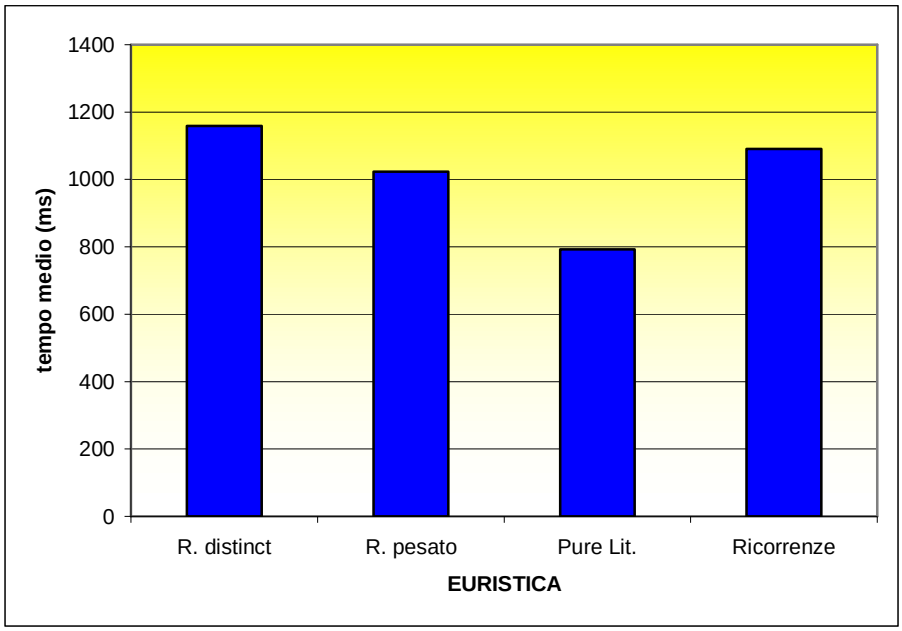
In questa istanza, malgrado il numero di split medio sia comunque abbastanza basso, questa euristica ha quasi il peggior tempo d'esecuzione.

3.4.5. Confronto Euristiche

Questa istanza funziona molto bene con l'euristica del random pesato, che esegue un numero medio di split molto minore degli altri.



Dal punto di vista dell'esecuzione il random pesato si comporta comunque bene, ma il pure literal (che però come accennato è più dovuto al caso) ha un tempo d'esecuzione medio molto minore degli altri. L'ultima euristica, che nelle altre due istanze si era dimostrata la migliore, ha un tempo d'esecuzione addirittura più elevato del random pesato.



3.5. Conclusioni basate sulle tre istanze sperimentali

Come intuibile, nei problemi NP-Completi come SAT non esiste un'euristica che si dimostra essere sempre la migliore per ogni istanza. Esistono però classi di problemi dove un'euristica si comporta mediamente meglio delle altre.

Nei primi due esperimenti si era riscontrato come la quarta euristica era stabilmente più vantaggiosa delle altre; nel terzo esperimento però questa si è verificata invece come una delle peggiori dal punto di vista di tempo d'esecuzione e numero di split.

L'euristica che ricerca il letterale puro invece nella prima istanza era la peggiore delle quattro in quanto la scelta deterministica le faceva eseguire sempre il maggior numero di split; nella terza istanza invece si è dimostrata nettamente la migliore dal punto di vista del tempo d'esecuzione. Vi è però da far presente che, a parte la prima istanza, nelle altre due non si trovano letterali puri (dove questa euristica può effettivamente sfruttare la sua forza, eseguendo solo uno dei due rami dello split).

Piccola nota può essere fatta nel confronto tra il random distinct ed il random pesato. In questo ne esce sicuramente vincitore (almeno nelle 3 istanze di prova) il random pesato che, a parte il primo esercizio dove è risutato peggiore dal punto di vista degli split, è sempre stato migliore del "rivale".

Questi benchmark restano comunque insufficienti per provare a determinare quale euristica riesca a comportarsi meglio nella maggior parte dei problemi. Bisognerebbe avere a disposizione e testare molti problemi diversi per avere già una prima stima.