PROGETTO DI RAGIONAMENTO AUTOMATICO

A.A. 2010/2011

CNF-SAT PROBLEM
Sat Solver with Backtracking and Graph-Based Backjumping

Roffo Giorgio Mat. VR094159

CONTENUTI

INTRODUZIONE

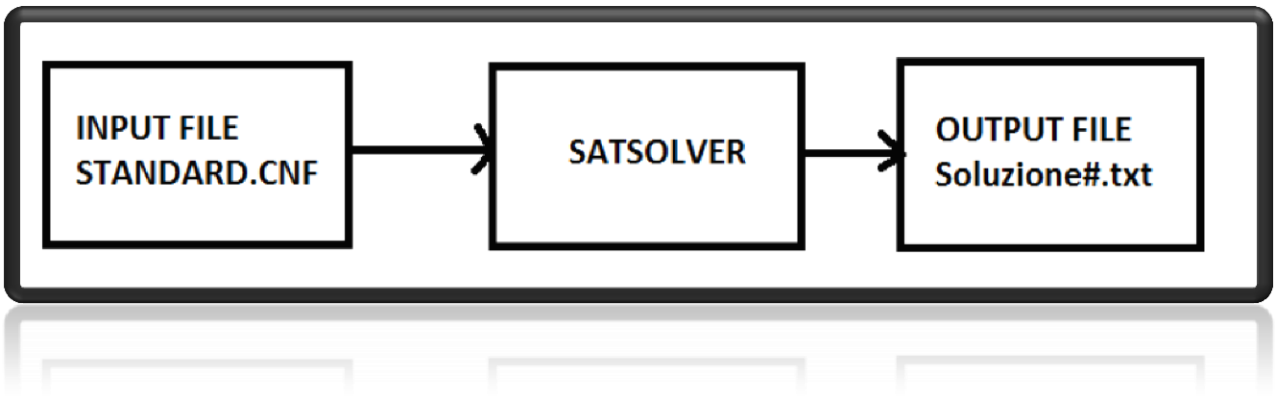
In questo elaborato è stato trattato il problema denominato CNF-SAT ossia quello di decidere la soddisfacibilità booleana di formule di logica proposizionale in forma normale congiuntiva (CNF).

Di seguito è prevista la presentazione di due differenti implementazioni (Backtracking e Graph-Based Backjumping) del famoso algoritmo (*)**DPLL** e dei risultati ottenuti testando l'applicazione su istanze di problemi SAT.

In generale il Sat Solver implementato acquisirà come input una istanza del problema SAT descritta all'interno di un file di testo in formato standard CNF. Le istanze sono disponibili sul sito web SATLIB http://www.satlib.org/ dove è possibile trovare moltissime varianti tutte compatibili con il Solver di questo elaborato.

In una seconda fase, una volta acquisiti i dati, sarà possibile scegliere l'implementazione del DPLL da utilizzare (Backtracking o Graph-Based Backjumping) al termine della procedura il risultato, che in caso positivo saranno tutti gli assegnamenti dei valori alle variabili e le statistiche, sarà salvato nel file di output denominato "Soluzione#.txt".

Per avere un confronto sulle prestazioni è stato utilizzato un secondo Sat Solver "zChaff" scaricabile al seguente indirizzo: http://www.princeton.edu/~chaff/zchaff.html, durante la presentazione dei risultati sarà inserita anche la sua prestazione per avere un confronto sulla difficoltà del problema.



*(*) Il **DPLL** (**Davis-Putnam-Logemann-Loveland**) è un algoritmo completo, basato sul backtracking, è stato introdotto nel 1962 da Martin Davis, Hilary Putnam, George Logemann e Donald W. Loveland, e rappresenta una specializzazione del precedente algoritmo di Davis-Putnam, una procedura basata sulla risoluzione sviluppata da Davis e Putnam nel 1960. Per questo, soprattutto nelle pubblicazioni più vecchie l'algoritmo Davis-Logemann-Loveland è spesso indicato come il "metodo Davis-Putnam" o "algoritmo DP". Altre nomenclature comuni che mantengono la distinzione fra i due sono DLL e DPLL.*

Formato CNF

Il problema della soddisfacibilità nella forma normale congiuntiva consiste nella congiunzione di un insieme di clausole, dove ogni clausole è una disgiunzione di un numero di variabili o della loro negazione. Se Xi è una variabile che può assumere valore true o false allora un esempio di formula in CNF è il seguente:

$$(X1 \vee X3 \vee \neg X4) \wedge (X4) \wedge (X2 \vee \neg X3)$$

Da cui il formato standard CNF per i file di input:

```
c Esempio:
c File:  standard.cnf
p cnf 4 3
1 3 -4 0
4 0
2 -3 0
```

Dove la “c” indica un commento, la “p” ci dice che la formula è in formato cnf con 4 variabili e 3 clausole, ogni clausola termina con uno 0.

Installazione ed Esecuzione

Il progetto è stato sviluppato in C++ è composto dai file Grafo.cpp Grafo.h MYDPLL.cpp il file eseguibile è SATSolver.x. Per utilizzare l'applicazione estrarre il contenuto del file compresso in una cartella, accedere alla cartella dal terminale ed eseduire il comando **make**.

Per avviare l'applicazione usare il file eseguibile: **./SATSolver.x**

All'interno della cartella CNF sono presenti tutte le istanze in formato standard utilizzate per i test e la produzione dei risultati, per poterle usare sarà sufficiente copiare la formula di interesse nella cartella in cui è presente l'eseguibile SATSolver.x e rinominando sempre il file di input in “standard.cnf”. Il programma caricherà sempre il file con nome “standard.cnf” dalla cartella locale di esecuzione.

Risultati

Il risultato della procedura è binario, se il problema è soddisfacibile quindi esiste un assegnamento completo e consistente per le variabili allora sarà visualizzato a video il modello e anche salvato nel file di output denominato “Soluzione#.txt”, nel caso in cui il problema non fosse soddisfacibile sarà generato comunque il file di output in cui saranno disponibili le statistiche.

ALGORITMO DPLL

L'algoritmo di backtracking di base viene eseguito scegliendo un letterale, assegnandogli un valore di verità (true o false), semplificando la formula e poi, ricorsivamente, verificando se la formula semplificata sia soddisfacibile; se è questo il caso, la formula originale è anch'essa soddisfacibile; altrimenti, si opera la stessa procedura ricorsiva assumendo l'altro valore di verità (false o true). Tale procedimento è noto come *splitting rule*, poiché divide il problema in due sotto-problemi più semplici. Il passo di semplificazione rimuove, essenzialmente, tutte le clausole che sono diventate vere in quell'assegnamento parziale della formula, ed elimina dalle clausole rimanenti tutti i letterali che sono divenuti falsi.

L'algoritmo DPLL potenzia il backtracking con l'utilizzo coatto di queste regole, ad ogni passo:

Propagazione unitaria

> Se una clausola è *unitaria*, i.e. contiene solo un singolo letterale non assegnato, questa clausola sarà soddisfatta solo assegnando il necessario valore di verità che rende tale letterale vero. Quindi non è necessaria alcuna scelta, e nella pratica ciò porta spesso ad una cascata di clausole unitarie che ridurrà la dimensione dello spazio di ricerca.

Eliminazione dei letterali puri

> Se una variabile proposizionale appare nella formula solamente in una polarità, è detta *pura*. I letterali puri possono essere sempre assegnati in modo da rendere vere tutte le clausole che li contengono. Dunque tali clausole non interessano più la ricerca e possono essere eliminate.

L'insoddisfacibilità di un dato assegnamento parziale è verificato se una clausola diventa vuota, i.e. se tutte le sue variabili sono stati assegnate in modo tale da rendere falsi i letterali corrispondenti. La soddisfacibilità della formula è verificata quando tutte le variabili sono assegnate senza generare alcuna clausola vuota, o, nelle implementazioni moderne, se tutte le clausole sono soddisfatte. L'insoddisfacibilità della formula completa può essere verificata solamente dopo una ricerca esaustiva del problema.

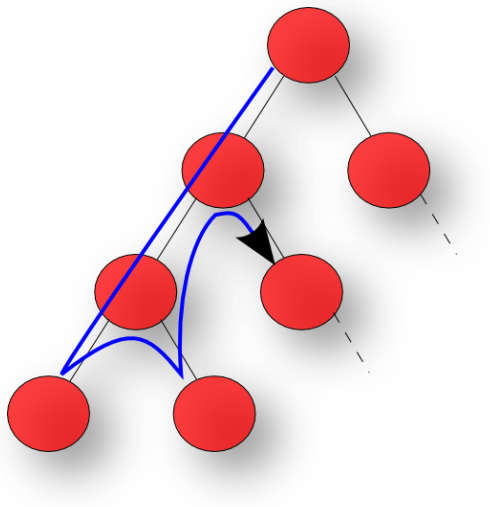
L'algoritmo DPLL può essere sintetizzato da questo pseudocodice, dove Φ è la formula CNF e μ è un assegnamento parziale, inizialmente vuoto:

```
funzione DPLL(Φ, μ)
  if Φ=T
    then return true;
  if Φ=F
    then return false;
  if clausola unitaria (l) si trova in Φ
    then return DPLL(assign(l,Φ), μ∧l);
  if letterale l si trova puro in Φ
    then return DPLL(assign(l,Φ), μ∧l);
  l := scegli-letterale(Φ); //Con una Euristica
  return DPLL(assign(l,Φ), μ∧l) OR DPLL(assign(NOT(l),Φ), μ∧NOT(l));
```

DPLL BACKTRACKING CRONOLOGICO

Con questa tecnica si considerano successivamente tutte le possibili soluzioni, scartando man mano le condizioni che non soddisfano i vincoli. Lo splitting viene eseguito solo dopo che sono state utilizzate le regole di unit propagate e di pure literal questa ottimizzazione riduce il numero delle ramificazioni.

Una tecnica classica consiste nell'esplorazione di strutture ad albero (es: Binario) in cui ogni nodo rappresenta un letterale ed ogni ramo un assegnamento (es: true,false).



Depth-First Search

L’algoritmo di Backtracking, qui proposto, si basa su un grafo di letterali, infatti ogni istanza SAT in formato CNF può essere rappresentata in un grafo in cui ogni nodo rappresenta una variabile (letterale) ed ogni arco un vincolo di tipo OR. Se due variabili sono collegate da un arco significa che compaiono insieme in almeno una clausola in modo positivo o negato.

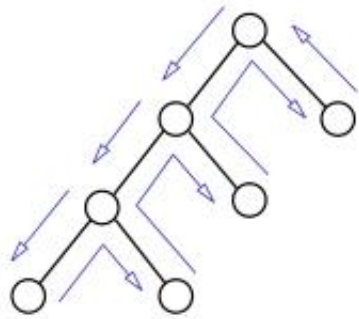
L'algoritmo procede impostando il valore di ogni variabile, finché la formula non è soddisfatta (supponiamo che sia soddisfacibile). Dunque per ogni nodo del grafo l'algoritmo assegnerà il valore “true” a P e false a ¬P se questo porterà ad un fallimento sarà tentata la seconda strada con il valore “true” a ¬P e “false” a P.

```
If ( esegui_BACKTRACKING(G->getNext(n),1) ) return true; //Prova con P
return esegui_BACKTRACKING(G->getNext(n),0);              //Prova con ¬P
```

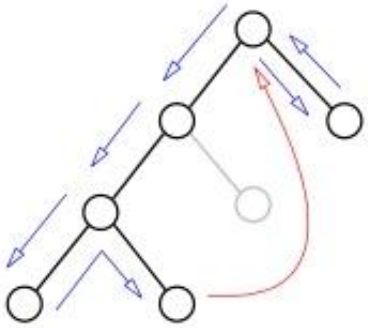
Se si giunge ad un nodo fallimento foglia salterà al nodo precedente provando, se esiste, un assegnazione differente. Il nodo di fallimento non è necessariamente una foglia potrebbe accadere che prima di assegnare un valore a tutte le variabili ci si trovi in uno stato inconsistente, dunque inutile procedere con gli altri assegnamenti. Grazie ad una funzione isConsistent(int i-clausola), chiamata dopo ogni assegnamento, che controlla se dopo un certo assegnamento la clausola i-esima è ancora consistente ossia se è soddisfatta oppure se contiene almeno un letterale non assegnato che potrà soddisfarla in futuro. In caso la clausola sia completamente falsa lo stato attuale è inconsistente e l’ultima scelta fatta sarà cambiata.

DPLL BACKJUMPING GRAPH-BASED

Il **DPLL Backjumping** (o backtracking non cronologico) è un miglioramento della procedura di backtracking. Questo metodo prevede una strategia differente di salto all'indietro, se nel backtracking il salto è fatto sempre al nodo precedente nel backjumping si cerca, tra tutti i nodi già visitati, il possibile responsabile del fallimento.



Backtracking



Backjumping

L'algoritmo implemetato è il Graph-Based Backjumping che, a differenza della versione classica Gaschnig's Backjumping, permette di tagliare parti significanti dello spazio di ricerca anche grazie alla capacità di effettuare dei jump non solo su una leaf dead end ma anche su nodi internal dead end risparmiando così molte operazioni.

L'algoritmo ha come fasi iniziali le stesse del DPLL con Backtracking:

Propagazione unitaria (One-Clause)

Eliminazione dei letterali puri (Pure-Literal)

Splitting Rule

```
If ( esegui_BACKTRACKING(G->getNext(n),1) ) return true;  //Prova con   P
if ( esegui_BACKTRACKING(G->getNext(n),0) ) return true;  //Prova con ¬P
else { BACKJUMPING }
```

Quando si giunge ad uno stato inconsistente si è arrivati ad una Leaf-Dead-End oppure un Internal-Dead-End, non avendo più modo di riportare la clausola al valore Vero si procedere al jump.

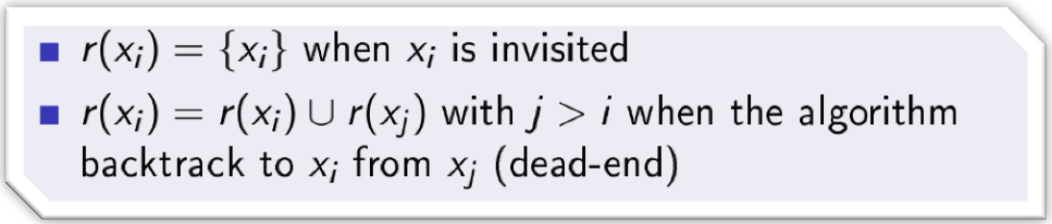
Il Jumping prevede una prima fase in cui si calcola il nodo a cui saltare in modo intelligente, lo scopo è quello di saltare al nodo che ha causato il fallimento che non necessariamente è l'ultimo nodo visitato, la seconda fase consiste nel tornare (nel grafo) al nodo scelto.

Definiamo con **Anc(n)**, dove **n** è un nodo del grafo, l'insieme di tutti i parenti del nodo e con **Father(n)** il parente più vicino, se il nodo **n** è un leaf dead end calcoliamo l'insieme:

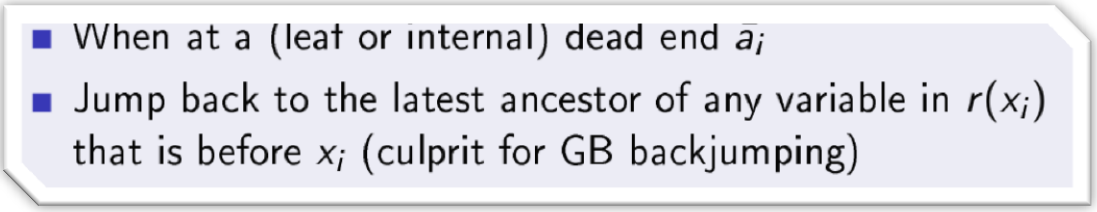
Anc(n) U Anc(Father(n))

Dal quale sarà scelto il nodo più vicino secondo l'ordine fissato. L'ordine dipenderà dall'euristica scelta per selezionare il nodo in fase di splitting.

Se il nodo è un internal dead-end calcolo i parenti conservando I nodi già visitati:



Per ogni elemento di r si calcolano tutti gli ancestor facendo l'unione insiemistica, il salto sarà fatto verso il nodo più vicino secondo l'ordine scelto:



(Slide Prof. Farinelli)

Per la scelta della variabile di splitting, quindi l'ordinamento dei letterali, sono state usate tre euristiche che in certi casi migliorano le prestazioni dei metodi.

EURISTICHE

I metodi descritti sono stati implementati in C++ e si basano sulla classe Grafo, implementata ad hoc per il problema, che rappresenta la struttura dati di base sulla quale lavorano gli algoritmi.

I metodi possono essere eseguiti con tre diverse euristiche, le euristiche entrano in gioco nel momento in cui, in fase di splitting, si deve decidere quale sarà la successiva variabile a cui assegnare un valore.

La prima euristica è la first-Order Variable la quale sceglie le variabili secondo l'ordine naturale di inserimento.

La seconda euristica è la MRV Heuristics (Most Constrained Variable) che seleziona il letterale più vincolato in questa prospettiva se le scelte sono fallimentari si arriverà prima al dead-end per procedere con una strada diversa. Questa euristica è chiamata Minimum Remaining Values in quanto selzionando la variabile più vincolata ci saranno molte più variabili con un dominio ridotto quindi con dominio ridotto e scelta più limitata.

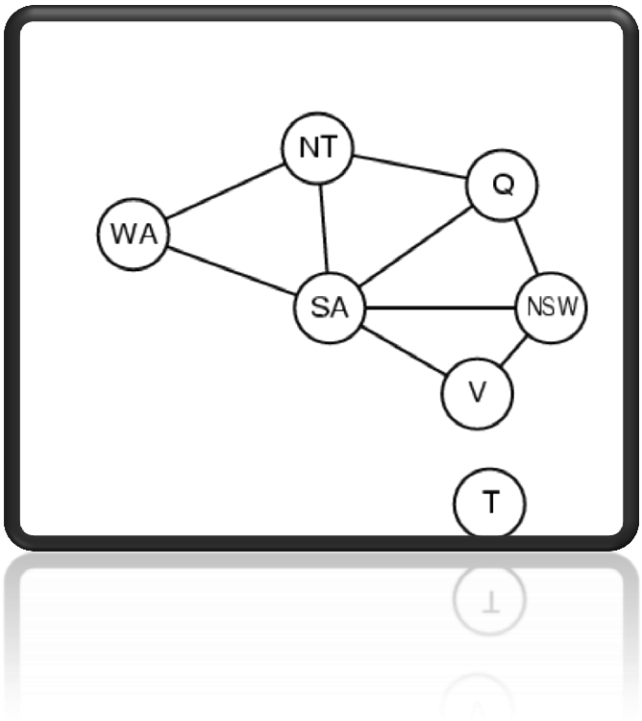


Figura 1

Dall'esempio in figura la prima variabile scelta sarebbe SA con cinque vincoli, assegnato il valore si continuerà con NT,Q,NSW e così via.

La terza euristica è la Least Constrained Variable o LCV Heuristics che seleziona il letterale meno vincolato lasciando così una maggiore flessibilità al sottoproblema rimanente.

Basandoci sulla figura 1 sarà scelta per prima la variabile T senza vincoli, successivamente WA con due vincoli

poi V,NSW,Q,NT in fine SA.

Per realizzare queste euristiche, a livello implementativo, è stato necessario effettuare un ordinamento dei nodi del grafo, nel caso della MRV, dopo aver contato in un opportuno buffer la cardinalità dei vincoli per ogni letterale, si procede inserendo nel grafo per prima I nodi con più vincoli. Viceversa nella LCV un ordinamento decrescende sul numero dei voncoli.

L'ordinamento dei nodi del grafo per le euristiche è fatto in fase di pre-processing e resta costante durante l'esecuzione dell'algoritmo.

PRESTAZIONI E STATISTICHE

Una volta avviata l'applicazione saranno visualizzati a schermo i risultati con le statistiche e contemporaneamente salvati nel file di output Soluzione#.txt, dove # indica la numerazione della soluzione. Nelle statistiche sono disponibili:

- **Numero di split effettuati**
- **Numero di jump (◄—►backjumping)**
- **Numero di letterali implicati (◄—► SAT)**
- **Tempo effettivo**
- **Numero clausole**
- **Numero Letterali**
- **Euristica utilizzata**

Il file Soluzione#.txt è generato sia in caso di successo con in aggiunta la soluzione trovata con tutti gli assegnamenti, sia in caso di insuccesso con le statistiche.

I letterali implicati sono tutti quelli che prendono un valore dalle regole di One-Clause o Pure-Literal.

Le statistiche sono state raccolte per istanze di dimensione crescente sia con backtracking che con backjumping su due macchine diverse:

- **Macchina M1**: Processore Intel® Pentium® D – (Lab.Alfa)
  - Freq. 2.80 GHz (default)
  - Sistema Linux Ubuntu 10,04
  - RAM 2GB
- **Macchina M2**: Processore Intel® Core™2 Quad Processor Q6600
  - Freq. 2.40 GHz
  - Sistema Linux Ubuntu 10,04
  - RAM 4GB

PRESENTAZIONE DEI RISULTATI

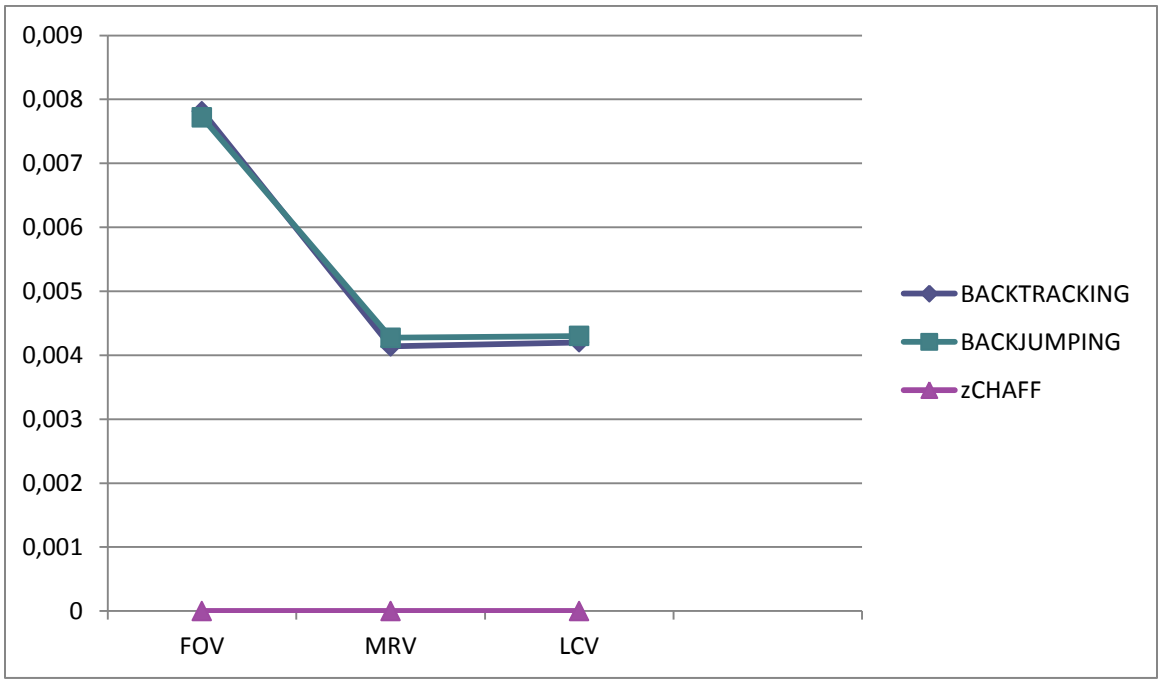
	TEST 1 (SATISFIABLE)	PROBLEM: 45 Clauses and 28 Literals INPUT FILE: standard1.cnf EURISTICA: First order variables		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	*61*	*61*	*61*	*61*
TIME	*0.007815 (sec)*	*0.007719 (sec)*	0.008301 *(sec)*	0.00831 *(sec)*
IMPLY	*12*	*12*	*12*	*12*
JUMP	*N0*	*0*	*N0*	*0*

	TEST 1 (SATISFIABLE)	PROBLEM: 45 Clauses and 28 Literals INPUT FILE: standard1.cnf EURISTICA: Minimum Remaining Values (MRV)		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	*37*	*36*	*37*	*36*
TIME	0.004138 (sec)	0.004272 (sec)	0.003792 (sec)	0.004608 (sec)
IMPLY	*12*	*14*	*12*	*14*
JUMP	*N0*	*3*	*N0*	*3*

	TEST 1 (SATISFIABLE)	PROBLEM: 45 Clauses and 28 Literals INPUT FILE: standard1.cnf EURISTICA: Least Constrained Variable or LCV Heuristics		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	*37*	*36*	*37*	*36*
TIME	0.004206 (sec)	0.004314 (sec)	0.004613 (sec)	0.004786 (sec)
IMPLY	*12*	*14*	*12*	*14*
JUMP	*N0*	*3*	*N0*	*3*

ZCHAFF	TIME	IMPLY	SAT/UNSAT
	0	28	SAT

Rappresentazione delle curve di prestazione temporale in base all'euristica scelta:



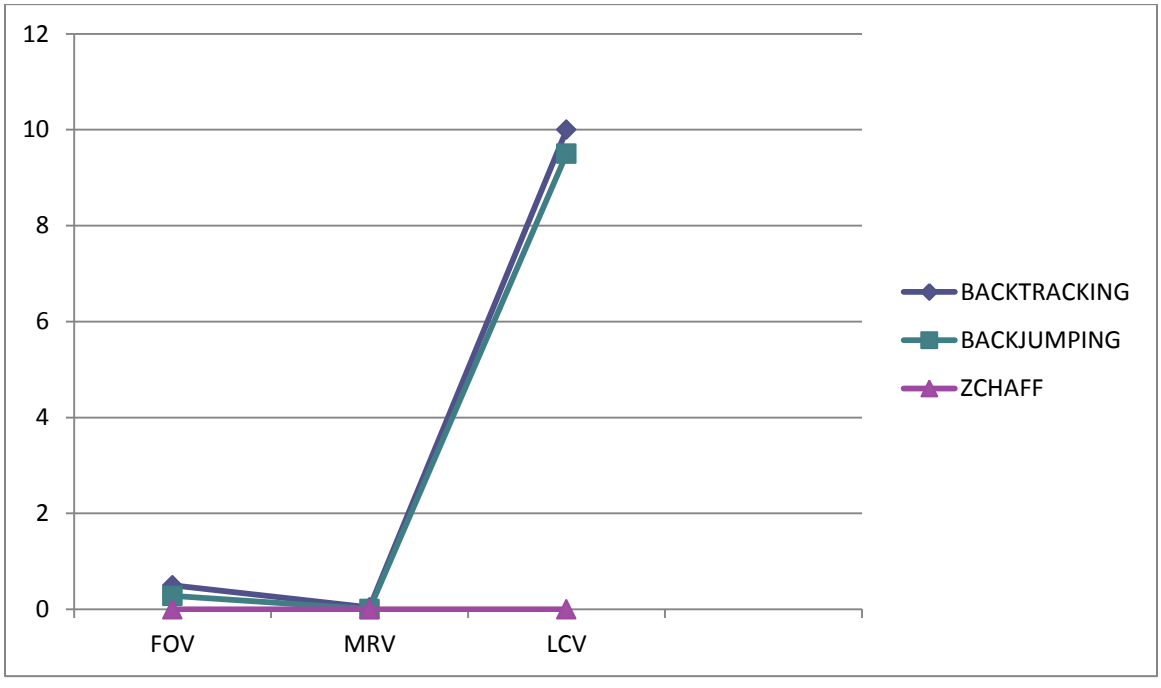
TEST 2 (UNSATISFIABLE)	PROBLEM: 45 Clauses and 28 Literals INPUT FILE: standard2.cnf EURISTICA: First order variables			
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	*4460*	*3012*	*4460*	3012
TIME	0.579094 *(sec)*	0.288017 *(sec)*	0.421989 *(sec)*	0.287017 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*4*	*N0*	*4*

TEST 2 (UNSATISFIABLE)	PROBLEM: 45 Clauses and 28 Literals INPUT FILE: standard2.cnf EURISTICA: Minimum Remaining Values (MRV)			
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	*350*	*28*	*350*	*28*
TIME	0.047057 *(sec)*	0.007106 *(sec)*	0.037189 *(sec)*	0.007002 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*12*	*N0*	*12*

	TEST 2 (UNSATISFIABLE)	PROBLEM: 45 Clauses and 28 Literals INPUT FILE: standard2.cnf EURISTICA: Least Constrained Variable or LCV Heuristics		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	90620	*90279*	90620	*90279*
TIME	12.3492 *(sec)*	11.92560 *(sec)*	8.9009 *(sec)*	8.76166 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*4*	*N0*	*4*

ZCHAFF	TIME	IMPLY	SAT/UNSAT
	0	355	UNSAT

Rappresentazione delle curve di prestazione temporale in base all'euristica scelta:



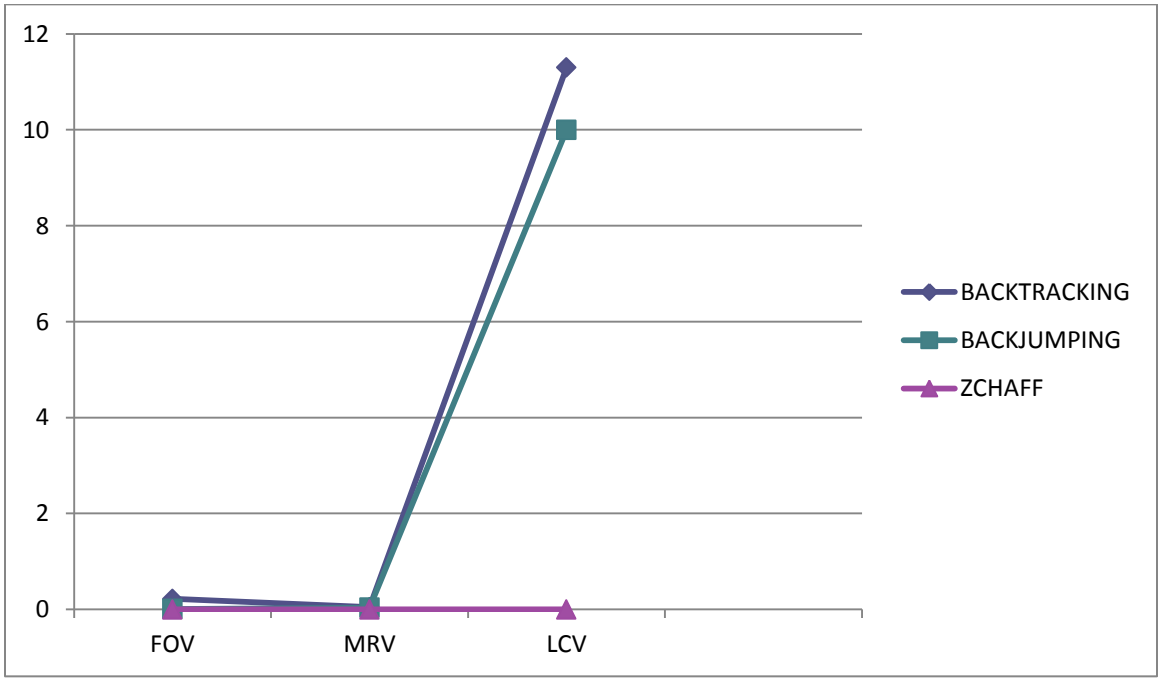
	TEST 3 (UNSATISFIABLE)	PROBLEM: 70 Clauses and 26 Literals INPUT FILE: standard3.cnf EURISTICA: First order variables		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	1402	25	1402	25
TIME	0.271808 *(sec)*	0.010637 *(sec)*	0.198902 *(sec)*	0.010865 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*11*	*N0*	*11*

	TEST 3 (UNSATISFIABLE)	PROBLEM: 70 Clauses and 26 Literals INPUT FILE: standard3.cnf EURISTICA: Minimum Remaining Values (MRV)		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	190	190	190	190
TIME	0.053187 *(sec)*	0.052353 *(sec)*	0.038652 *(sec)*	0.03699 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*0*	*N0*	*0*

	TEST 3 (UNSATISFIABLE)	PROBLEM: 70 Clauses and 26 Literals INPUT FILE: standard3.cnf EURISTICA: Least Constrained Variable or LCV Heuristics		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	70222	70222	70222	70222
TIME	13.1658 *(sec)*	13.1473 *(sec)*	9.78622 *(sec)*	9.91555 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*0*	*N0*	*0*

ZCHAFF	TIME	IMPLY	SAT/UNSAT
	0	140	UNSAT

Rappresentazione delle curve di prestazione temporale in base all'euristica scelta:



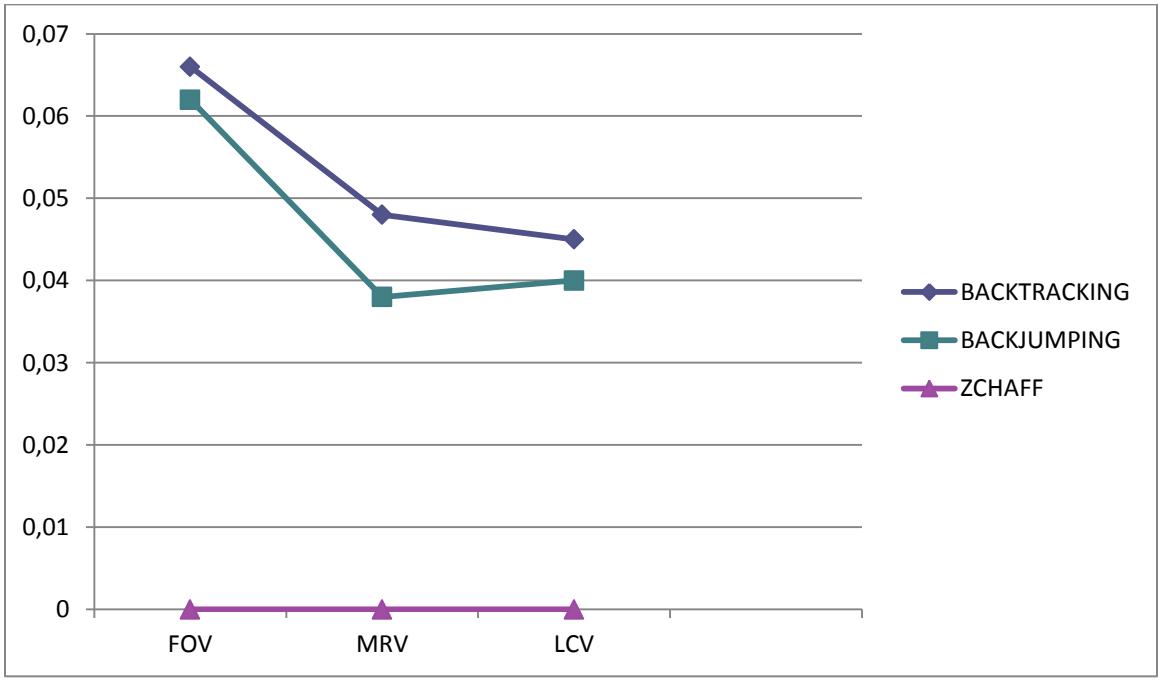
TEST 4 (UNSATISFIABLE)	PROBLEM: 84 Clauses and 36 Literals INPUT FILE: standard4.cnf EURISTICA: First order variables			
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	144	*127*	*144*	127
TIME	0.070104 *(sec)*	0.071555 *(sec)*	0.063383 *(sec)*	0.055671 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*1*	*N0*	*1*

TEST 4 (UNSATISFIABLE)	PROBLEM: 84 Clauses and 36 Literals INPUT FILE: standard4.cnf EURISTICA: Minimum Remaining Values (MRV)			
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	144	*127*	*144*	127
TIME	0.06814 *(sec)*	0.053612 *(sec)*	0.050395 *(sec)*	0.040395 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*1*	*N0*	*1*

	TEST 4 (UNSATISFIABLE)	PROBLEM: 84 Clauses and 36 Literals INPUT FILE: standard4.cnf EURISTICA: Least Constrained Variable or LCV Heuristics		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	144	*127*	*144*	127
TIME	0.0656 *(sec)*	0.063774 *(sec)*	0.05305 *(sec)*	0.050464 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*1*	*N0*	*1*

ZCHAFF	TIME	IMPLY	SAT/UNSAT
	0.00001	649	UNSAT

Rappresentazione delle curve di prestazione temporale in base all'euristica scelta:



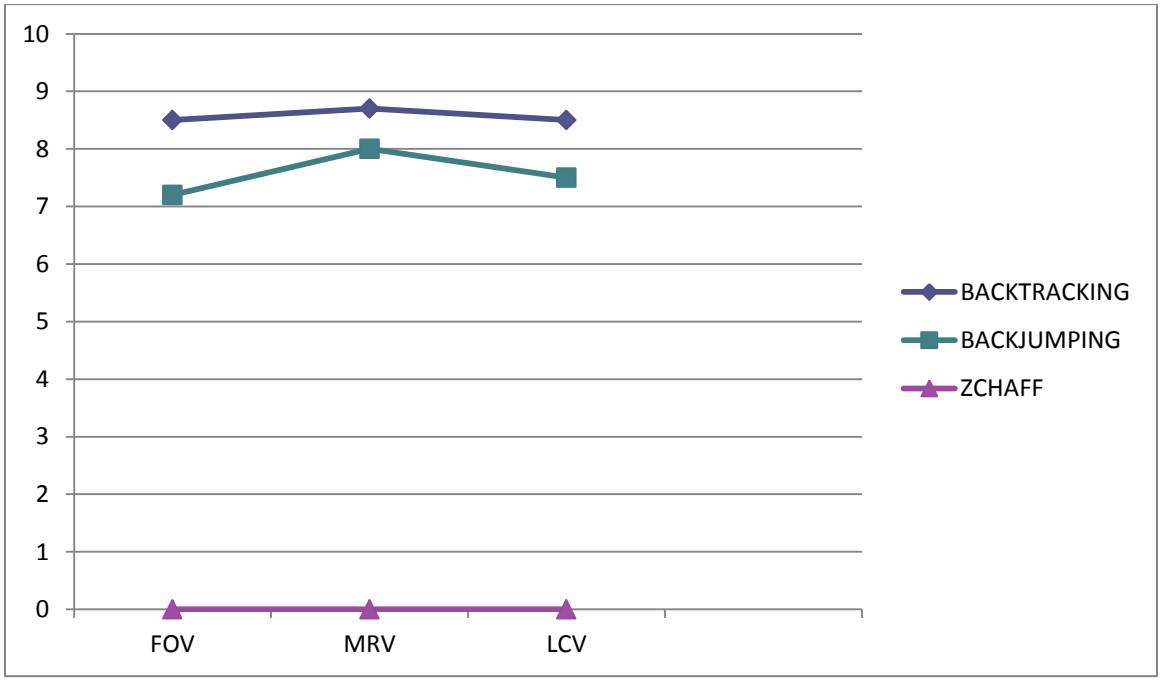
	TEST 5 (UNSATISFIABLE)	PROBLEM: 96 Clauses and 36 Literals INPUT FILE: standard5.cnf EURISTICA: First order variables		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	32764	28676	32764	28676
TIME	10.5826 *(sec)*	9.30019 *(sec)*	7.92221 *(sec)*	6.92943 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*2*	*N0*	*2*

	TEST 5 (UNSATISFIABLE)	PROBLEM: 96 Clauses and 36 Literals INPUT FILE: standard5.cnf EURISTICA: Minimum Remaining Values (MRV)		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	32764	28676	32764	28676
TIME	10.5996 *(sec)*	9.27259 *(sec)*	7.92562 *(sec)*	7.09168 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*2*	*N0*	*2*

	TEST 5 (UNSATISFIABLE)	PROBLEM: 96 Clauses and 36 Literals INPUT FILE: standard5.cnf EURISTICA: Least Constrained Variable or LCV Heuristics		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	32764	28676	32764	28676
TIME	10.589 *(sec)*	9.43782 *(sec)*	7.97126 *(sec)*	6.92822 *(sec)*
IMPLY	*N0*	*N0*	*N0*	*N0*
JUMP	*N0*	*2*	*N0*	*2*

ZCHAFF	TIME	IMPLY	SAT/UNSAT
	0.016001	10890	UNSAT

Rappresentazione delle curve di prestazione temporale in base all'euristica scelta:



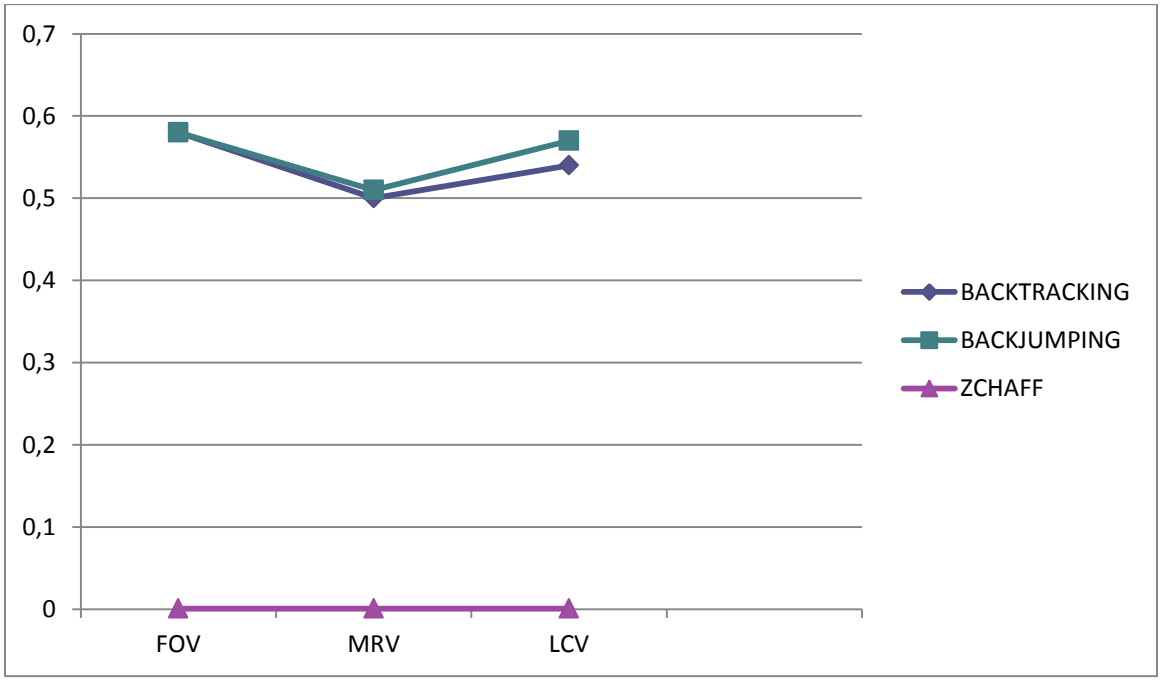
	TEST 6 (SATISFIABLE)	PROBLEM: 500 Clauses and 250 Literals INPUT FILE: standard6.cnf EURISTICA: First order variables		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	251	251	*251*	251
TIME	0.643405 *(sec)*	0.646288 *(sec)*	0.520696 *(sec)*	0.519617 *(sec)*
IMPLY	110	*110*	*110*	*110*
JUMP	*N0*	*0*	*N0*	*0*

	TEST 6 (SATISFIABLE)	PROBLEM: 500 Clauses and 250 Literals INPUT FILE: standard6.cnf EURISTICA: Minimum Remaining Values (MRV)		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	249	249	249	249
TIME	0.607938 *(sec)*	0.604488 *(sec)*	0.49505 *(sec)*	0.492655 *(sec)*
IMPLY	124	124	124	124
JUMP	*N0*	*0*	*N0*	*0*

	TEST 6 (SATISFIABLE)	PROBLEM: 500 Clauses and 250 Literals INPUT FILE: standard6.cnf EURISTICA: Least Constrained Variable or LCV Heuristics		
	COMPUTER M1		COMPUTER M2	
	BACKTRACKING	BACKJUMPING	BACKTRACKING	BACKJUMPING
SPLIT	250	250	250	250
TIME	0.687718 *(sec)*	0.738627 *(sec)*	0.571073 *(sec)*	0.570027 *(sec)*
IMPLY	105	105	105	105
JUMP	*N0*	*0*	*N0*	*0*

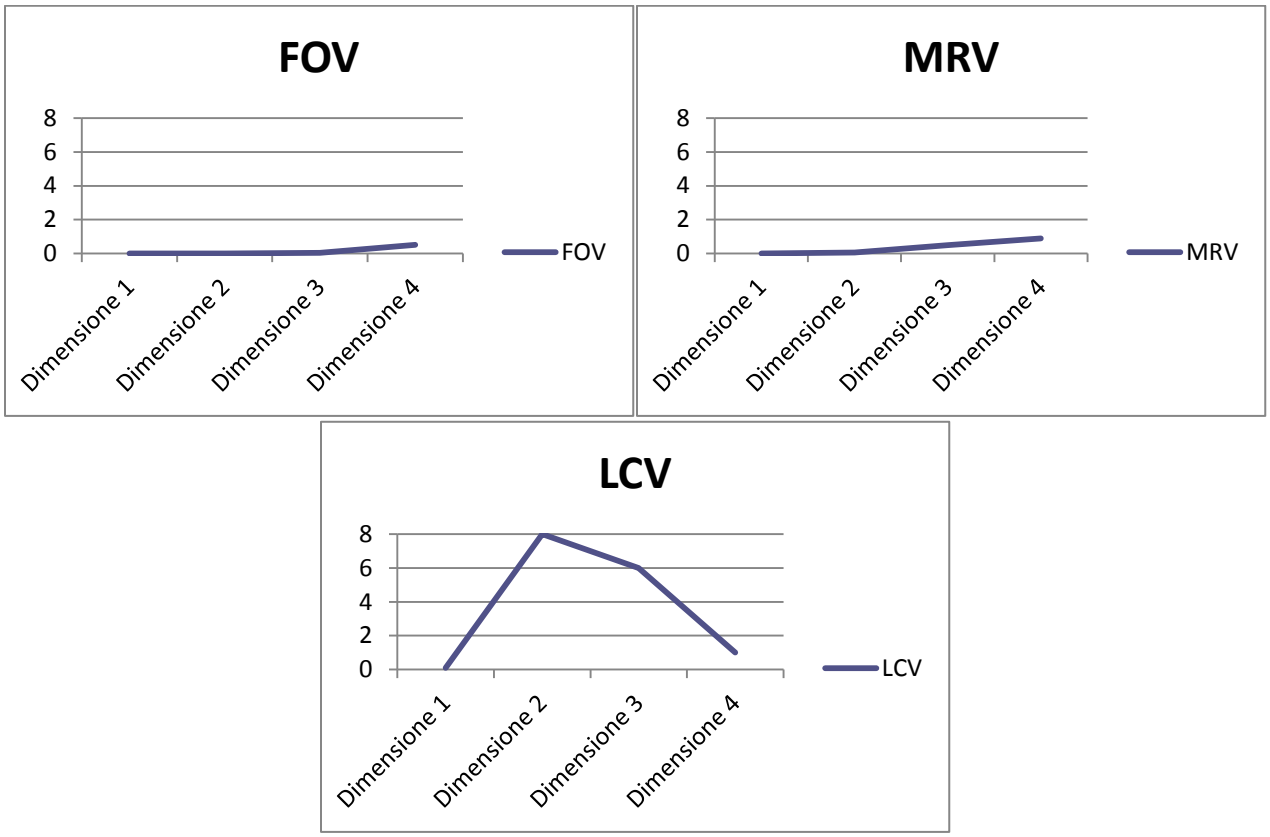
ZCHAFF	TIME	IMPLY	SAT/UNSAT
	0.00101	250	SAT

Rappresentazione delle curve di prestazione temporale in base all'euristica scelta:



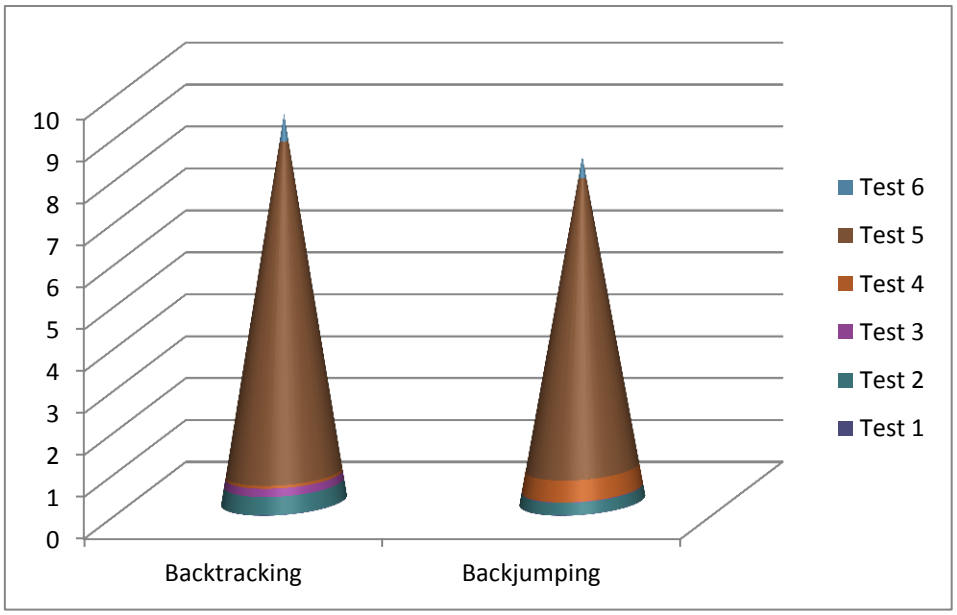
CONSIDERAZIONI CONCLUSIVE

Grafico dei **tempi medi** tra le macchine utilizzate al crescere della dimensione del problema (numero clausole e/o numero letterali):



(Grafici Tempo X Dimensione)

L'euristica LCV si è rivelata la peggiore i suoi tempi sono nettamente superiori alle altre due, le euristiche FOV e MRV hanno mantenuto un comportamento simile al crescere della dimensione del problema, in alcuni casi migliore la MRV in altri la FOV.

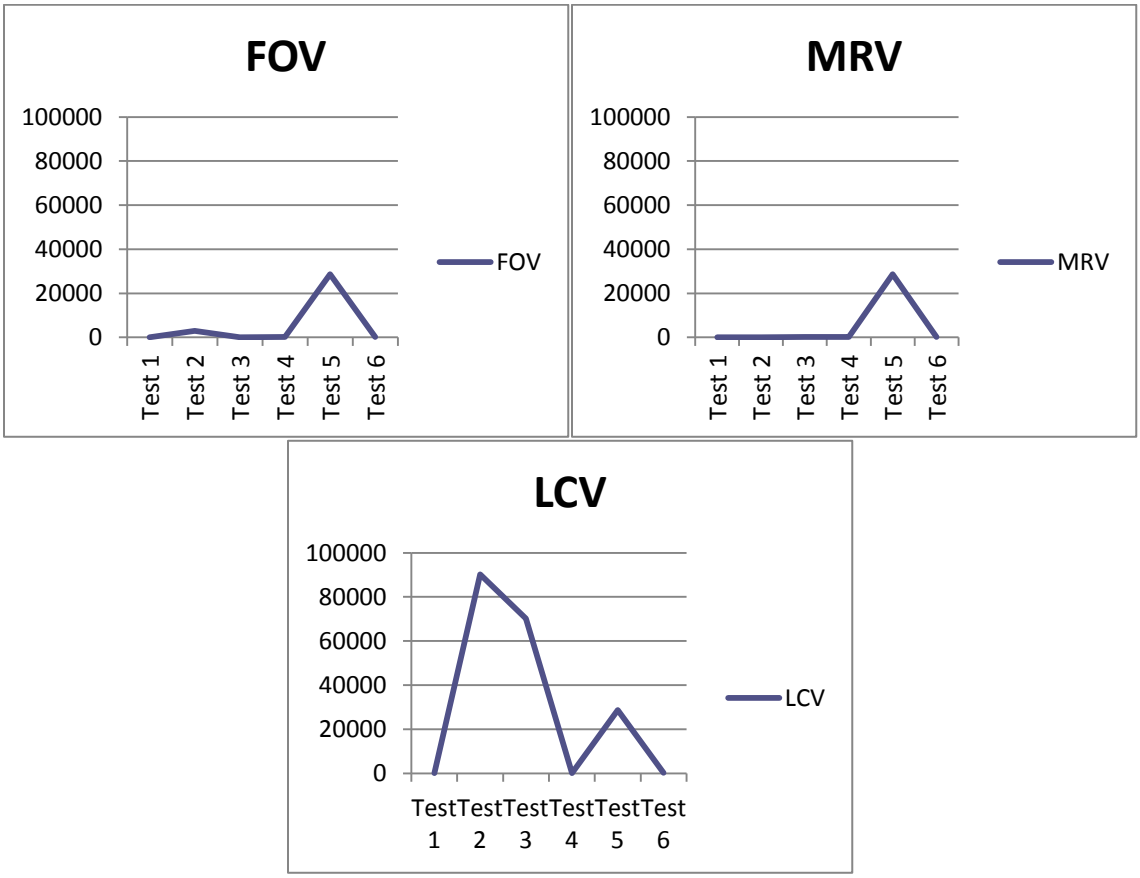


Nel grafico a sinistra sono messi a confronto i tempi ottenuti nei diversi test per le due procedure Backtracking e Backjumping. Il tempo complessivo del backtracking risulta maggiore di quello del backjumping.

(Grafico su Test Euristica FOV Tempi complessivi)

Di seguito nei grafici è mostrato l'andamento del numero di **Split** usando l'algoritmo di Backjumping con euristiche FOV, MRV ed LCV nei diversi test effettuati:

(Grafici Numero Split X Test)



(Grafici Normalizzati in ordinate a 100000 SPLIT)

E' interessante notare che l'euristica LCV è quella che genera il numero di split più alto nelle diverse prove conseguendo anche tempi più lunghi per arrivare alla soluzione.

Il programma zChaff ha realizzato tempi molto inferiori per risolvere i problemi, dal confronto tra i due metodi Backtracking e Backjumping è emerso un discreto vantaggio per la tecnica di Backjumping che ha conseguito in media tempi più bassi.