Il Terzo Modulo si occupa del completamento delle funzioni sviluppate nel Secondo Modulo: deve gestire la scoperta di un assegnamento contraddittorio, tornare indietro (backtracking) nell'albero (se possibile) ed iniziare l'esplorazione di un altro ramo, dopo aver memorizzato la contraddizione in una clausola learnt.

Il Secondo Modulo si è occupato della Propagazione unitaria (Unit Propagation) che assegna valori 'forzati' a letterali nella Coda (messi lì perchè ultimi "superstiti" di clausole unitarie), ed analizza, tramite la watcher list, l'effetto della propagazione su alcune clausole, che possono a loro volta diventare unitarie e così via.
Il processo si ripete fino a quando

1. tutte le variabili sono state settate (clausola soddisfatta dall'assegnamento attuale)
2. la Pcoda è vuota, ma ci sono ancora variabili con valore UNDEF.
3. si arriva ad una contraddizione (tentativo di inserimento in Pcoda di un letterale in contraddizione con uno già presente o con l'assegnamento attuale)

Nel primo caso, l'esecuzione termina con successo.
Nel secondo caso, si possono utilizzare delle euristiche (quali il numero di volte nelle quali appare in una clausola o quante volte è stata assegnata) per scegliere la variabile cui assegnare un valore per riprendere la propagazione. Per semplificare questo modulo, useremo invece un valore aleatorio ottenuto con `select` .
Nel terzo caso, si chiama la funzione analyze per costruire la learnt clause, che costituisce il 'cuore' del terzo modulo.

Nel Terzo Modulo quindi bisogna completare l'implementazione del metodo `solver_search` per trattare il caso in cui la chiamata al metodo `solver_propagate` restituisce un puntatore a clausola (trovato un conflitto). In questo caso, bisogna chiamare una nuova funzione `solver_analyze` e poi effettuare il backtracking fino al livello restituito da `solver_analyze`.
Se `solver_propagate` non restituisce un puntatore a clausola (restituisce NULL) verifichiamo che i parametri appropriati non siano stati superati (`nof_learnts`, `nof_conflicts`) e ci sono ancora variabili da assegnare, nel qual caso chiamiamo `solver_select` ecc. come implementato nel Secondo Modulo.
Poichè non abbiamo aggiornato il valore di activity, ignoriamo l'uso di `nof_learnts` per ridurre il numero di clausole imparate.
Se il numero di conflitti supera il limite indicato da `nof_conflicts`, si "azzera" tutto (backtracking fino al livello 0) e si ricomincia daccapo.

Il metodo `solver_analyze` viene chiamato da `solver_search` quando il metodo `solver_propagate` restituisce un puntatore non nullo. La funzione

`int` **`solver_analyze`**`(solver *,clause *,vec_i *)`

riceve in input la clausola che ha causato il conflitto, e costruisce un `vec_i` che verrà poi passato al `new-clause` per costruire la nuova clausola (`learnt` clause) che "ricorda" il conflitto, aggiungendola alla lista delle clausole imparate finora (parte di `solver`).

Per costruire la nuova clausola, supponiamo che la contraddizione sia stata raggiunta perchè abbiamo tentato di inserire in Pcoda il letterale p quando –p è già presente. In questo caso, la (prima bozza della) clausola imparata è la disgiunzione delle clausole `reason` di p e –p, senza ovviamente i letterali stessi p e –p.
Si ottiene lo stesso risultato come segue:

1. prendere la clausola c1 che ha creato il conflitto,
2. rimuovere il letterale p che veniva propagato quando si è raggiunto il conflitto, ottenendo la clausola c2
3. prendere la clausola q1 'reason' di –p e rimuovere il letterale –p, ottenendo la clausola q2
4. costruire la clausola c2 OR q2 (letterali di ciascuno, non ripetuti)

A questo punto, si semplifica la clausola (se, per esmpio, ci sono ripetizioni) per ottenere la clausola imparata. Si "ordinano" i letterali della nuova clausola, inserendo in prima e seconda posizione i letterali di livello di decisione piu' alto. Questa nuova clausola va poi inserita nella watcher list dei letterali appropriati (negazione dei primi due)
In aggiunta alla costruzione di un vec_i , la funzione `solver_analyze` restituisce come livello di backtracking il livello `lev` più piccolo tra i letterali della clausola (quello decisionale, non forzato, più vicino alla radice).
A questo punto, la `solver_search` effettua il backtracking, direttamente, o tramite chiamate a funzioni ausiliari apposite

`Void` **`Solver_Undo`** `(solver, // input+output)`
/*Post: elimina dal solver le informazioni relative al letterale p che si trova nell'ultima posizione di trail */
`Void` **`Solver_Cancel`**`(solver, // input+output)`
/*Post: elimina dal solver le informazioni relative al livello di decisione corrente chiamando solver_Undo() tante volte quanto il valore di (dimensione(trail) – ultimo elemento di trail_lim()) */

settando ad UNDEF (e ricordandosi di aggiornarne il corrispondente `reasons`, `level` ecc.) tutte le variabili presenti nel `trail` fino a `lev`, e si inserisce il letterale corrispondente in coda.
La coda non è più vuota, e si ricomincia con la propagazione.