1 Analisi dello pseudocodice nella versione 3

La versione 3 si occupa dell'implementazione degli algoritmi descritti nella consegna del progetto quindi di *EspressioneRegolare* ed *EspressioniRegolari*: per comodità ed evitare ridondanza entrambi i procedimenti sono stati implementati nello stesso pseudocodice discriminando le varie righe con l'uso della variabile *split*.

1.1 Ottimizzazioni e gestioni delle rimozioni

I punti critici da gestire per questa versione saranno le varie ricerche da effettuare per controllare gli stati e le transizioni della rete comportamentale: per attenuare il carico computazionale nell'algoritmo *unifyParallelTransition* è stata implementata un'hashMap per il controllo delle transizioni chiamata *trHashMap* (riga 3).

Per questa versione la gestione delle rimozioni di stati e transizioni ottimale diventa fondamentale infatti, a causa della strutture dati implementate in questo progetto, la rimozione di un nodo o transizione della rete è formata da una serie continua di ricerche in varie liste, visto che dovranno essere chiamate un *O(s)* volte il carico computazionale di queste rimozioni rischia di diventare oneroso.

Per facilitare la ricerca è stata implementata una HashMap in *automaton* chiamata *sttrHashMap*: questa contiene tutti i puntatori di stati e transizioni alle varie liste ad esempio, se una transizione t sarà uscente in uno stato s_{out} ed entrante in uno stato s_{in}, t si troverà puntato sia nella lista $out[s_{out}]$ che nella lista $in[s_{in}]$ oltre che nella lista dell'automa $transitions[automa]$, quindi troveremo tre item in *sttrHashMap* che punteranno ai seguenti puntatori, stessa cosa vale per gli stati. Al posto di cercare in singole liste ogni singolo puntatore facente rifermiento al determinato oggetto che stiamo rimuovendo basterà cercare in *sttrHashMap* l'item desiderato e rimuovere dalla lista. La gestione delle singole ricerche sarà affidato da un metodo specifico che nello pseudocodice è definito da varie funzioni (*statePointerSearch,transitionInPointerSearch,transitionPointerSearch*) specificate per aumentare la leggibilità

Il miglioramento è di un ordine di grandezza infatti $removeTheState = O(s)$ mentre $removeTransition = O(1)$ contro rispettivamente i $O(s^2 + ts)$ e $O(t + s)$ precedenti: valgono le stesse ipotesi fatte per $ctrHashMap$ visto che sia $trHashMap$ e sia $sttrHashMap$ sono a concatenazione e usano la stessa funzione di hash quindi effettuano una ricerca in un tempo medio $O(1)$

1.2 Analisi delle complessità in *diagnosis*

1.2.1 Calcolo della complesstià computazionale

Il maggior carico che troviamo in questo algoritmo è rappresentato dal ciclo while (righe 7-13) e dagli algoritmi che contiene (*replaceOneToOneState,unifyParallelTransitions,replaceManyToManyStates*.
Consideriamo prima il ciclo while che continuerà a ripetersi finchè non si troverà una sola transizione nella rete contente l'espressione regolare calcolata: sia s_i e t_i rispettivamente il numero di stati e transizioni della rete comportamentale in input nell'algoritmo nell'iesima iterazione del ciclo. Per semplificare consideriamo s_i e t_i costanti per tutta l'iterazione e che varino solo alla fine di quest'ultima. nella prima iterazione:

$$s_1 = s + 2, t_1 = t + f$$

dove s, t ed f sono rispettivamente il numero di nodi, transizioni e stati finali nella rete, grazie agli algoritmi *replaceInitialState* e *replaceEndState* (righe 4-5) aggiungiamo uno stato iniziale e finale (*initState* e *endState*) oltre che aggiungere una transizione da ogni stato finale ad *endState*. Nell'iterazione successiva $s_2 < s_1$ e $t_2 < t_1$ per costruzione stessa cosa per le iterazioni successive:

$$\forall i = 1, 2..., m : s_i < s_{i-1}, t_i < t_{i-1}$$

Possiamo quindi omettere la complessità delle iterazioni successive alla prima perchè poco significanti per il calcolo complessivo.

replaceOneToOneStates pesa $O(s_1^2)$ per costruzione, *parallelTransitions* controlla tutte le transizioni della rete quindi $t_1 = O(s_1^2)$ nel caso peggiore di rete completa ed infine *replaceManyToManyStates* a causa del'algoritmo *unifyAllTransitionState* (riga 7) ($O(s_1^3)$) arriva a pesare $O(s_1^4)$ sovrastando gli altri due. Possiamo quindi indicare la complessità computazionale di *diagnosis* come:

$$O(s_1^4) \simeq O(s^4) = O((a^3 n^{a+1} l^k o)^4)$$

1.2.2 Considerazioni spaziali sull'implementazione di *sttrHashMap*

Per quanto *sttrHashMap* ottimizzi al meglio i tempi di rimozione la dimensione dell'hashMap risulta molto grande:
sttrHashMap contiene tutti i puntatori inerenti alle seguenti liste:

- *transitions[automa]* di cardinalità t (considerando un solo automa come nelle reti comportamentali)
- le liste *trIn[stato]* che sommate portano ad una cardinalità totale t
- le liste *trOut[stato]* che sommate portano anche loro ad una cardinalità t
- *states[automa]* di cardinalità s (considerando l'automa della rete comportamentale)

Quindi l'HashMap conterrà esattamente $d = 3t + s$ elementi e quindi:

$$d = O(3s^2)$$

per quanto questo valore, di una struttura dati non necessaria per l'implementazione del programma, possa pesare sulla memoria occupata preferiamo concentrarci sulla velocità di esecuzione considerando le specifiche base di un calcolatore medio capace ancora almeno, per gli esempi considerati, di contenere questo problema.