1 Analisi dello pseudocodice nella versione 1

prima di calcolare le complessità dobbiamo soffermarci su alcune considerazioni necessarie per calcolare la complessità temporale.

1.1 cardinalità dello spazio comportamentale

Ogni nodo è identificato esattamente da un contesto: dato un esatto momento dell'esecuzione di una rete definiamo come contesto l'insieme formato dallo stato in cui ogni automa è presente e dagli eventi contenuti in ogni link della rete. Possiamo definire ogni conesto in questo modo:

$$n_1, n_2, ..., n_a, l_1, l_2, ..., l_k$$

Dove per n_i intendiamo il numero di stati presenti nell'automa i, l_j il numero di eventi presenti nel link j, a il numero di automi nella rete e k il numero di link

Vista questa caratterizzazione ogni singola variazione di uno dei valori del contesto (stato o evento) implica un possibile nuovo contesto e quindi un nuovo nodo. Consideriamo allora tutte le possibili permutazioni:

$$\prod_{i=1}^{a} n_i \prod_{j=1}^{k} l_j$$

Questo valore rappresenta il numero massimo possibile di nodi che possono essere generati in ogni rete comportamentale, semplificando:

$$\prod_{i=1}^{a} n_i \prod_{j=1}^{k} l_j \leq n^a l^k$$

con $n = \max(n_1, n_2, ..., n_a)$ e $l = \max(l_1, l_2, ..., l_k)$.
Quindi se s è il numero esatto di nodi nella rete comportamentale possiamo indicare:

$$s = O(n^a l^k)$$

Questa crescita esponenziale del numero di nodi si può faclmente notare anche con i semplici esempi giocattolo lasciati per il testing per il progetto, sarà necessario tenere in considerazione questo valore s perchè risulterà critico nelle versioni successive.

1.2 Ottimizzazione

A causa della crescita esponenziale dei nodi nello spazio comportamentale abbiamo ritenuto opportuno l'uso di un' hashmap per la gestione dei contesti: infatti dalla riga 8 alla riga 14 della funzione *step* si controlla che il nodo considerato non sia già stato visitato in precedenza, per poterlo fare è necessaria una ricerca tra tutti i possibili contesti; una semplice lista porterebbe ad un alto costo computazionale. La tabella di hash considerata risolve le collisioni per concatenazione e sfrutta una funzione di hash che segue il metodo di divisione:

$$h(k) = k * mod * m$$

dove $m = 307$ qundi un valore primo non troppo vicino a potenze esatte di 2, grazie a questa accortezza possiamo assumere la condizione di uniformità semplice e visto che la funzione calcola il risultato in un $O(1)$ per sua costruzione possiamo considerare il tempo medio di ricerca t della funzione di hash come:

$$t = O(1)$$

nello pseudocodice l'hashMap è chiamata come *ctHashMap*, quando sarà necessario nelle versioni successive verranno definite altre hashMap nei punti più critici.

1.3 Gestione del pruning

La rimozione dei nodi avviene subito dopo al creazione dell'ambiente comportamentale nella funzione *prune*, dopo l'esecuzione di *step*: prenendo in considerazione il codice di *prune* l'algoritmo prima individua i nodi attraverso una strategia backward richiamando la funzione /emphdfsVisit sui nodi finali (righe 5-6), una variante della visita in profondità che individua i nodi raggiungibili (a cui viene associato il colore nero) e quelli irraggiungibili (nodi rimasti bianchi); se un nodo non è raggiungibile dagli stati finali allora porta sicuramente ad un vicolo cieco quindi può essere rimosso (righe 11-13). Nel calcolo della complessità temporale la funzione *prune* non verrà considerata perché non risulta critica.

2 Analisi delle complessità

2.1 Calcolo della complessità temporale nella funzione step

Consideriamo ora lo pseudocodice di *step*, vista la natura ricorsiva prima troviamo il costo di ogni ricorsione: il cuore dell'algoritmo si trova nel ciclo while di riga 4 dove avviene un controllo di tutte le transizioni in uscita, quindi il ciclo si ripeterà esattamente t_{ji} volte dove t_{ji} rappresenta il numero delle transizioni uscenti dal j-esimo stato presente nell'i-esimo automa.

Controllando le singole funzioni definite le più costose risultano *isBufferFree* (riga 5) e *createNewContext* (riga 6); entrambe le funzioni ciclano sul numero di azioni prodotte in uscita dalla transizione presa in esame, definiremo l_{ki} come il numero di link attivati dalla k-esima transizione presente nell i-esimo automa.

Visto che questo controllo delle transizioni viene ripetuto per ogni stato "puntato" da ogni automa possiamo calcolare il costo complessivo di ogni ricorsione:

$$\sum_{i=1}^{a} t_{ij} l_{ik} \leq \sum_{i=1}^{a} (n_i - 1) l_{ki}$$

Considerando la rete completamente connessa (che rappresenta il caso peggiore) da ogni stato possono uscire al massimo $n_i - 1$ transizioni:

$$\leq (\sum_{i=1}^{a} (n_i - 1))(a-1)a$$

Visto che in ogni automa al massimo possiamo avere $a - 1$ link:

$$\leq a(n-1)(a-1) = O(a^3 n)$$

2.2 calcolo della complessità temporale nella ricorsione di step

step segue una strategia in profondità e non ha una condizione specifica di stop, si ferma solo quando ha calcolato tutto lo spazio comportamentale e quindi trovato tutti i nodi raggiungibili, una ricorrenza che modellizza lo pseudocodice può essere:

$$\begin{cases} T(1) = O(a^3 n) \\ T(s) = T(s-1) + O(a^3 n) \end{cases}$$

Visto che ogni ricorrenza è indipendente dalle altre il costo totale diventa:

$$\sum_{i=1}^{s} T(i) = O(a^3 n^{a+1} l^k)$$

La complessità quindi risulta molto sensibile alla crescita del numero degli automi e del numero di link questo a causa della crescia esponenziale del possibile numero di nodi nello spazio comportamentale

2.3 calcolo della complessità spaziale

I punti più critici nel calcolo della complessità spaziale risultano le funzioni *step* e *dfsVisit*, per quanto entrambi gli algoritmi non hanno ricorsioni in coda non è possibile usufruire di un compilatore ottimizzante per questo abbiamo ovviato per una strategia in profondità: essendo il numero di nodi nella rete comportamentale un valore sempre elevato rispetto alle dimensioni delle reti da cui derivano una strategia in ampiezza avrebbe occupato molta più memoria.

Consideriamo come r il numero massimo di record di attivazione presenti nello stack durante l'esecuzione del programma, i nostri algoritmi seguendo la strategia in profondità continueranno a scendere nella rete finchè troveranno almeno un nodo non ancora esplorato (come avviene in *step* nelle righe 9-14 e in *dfsVisit* nella 5-6), quindi l'esplorazione avviene considerando la rete come se fosse un grafo aciclico e quando è presente una transizione rientrante in un nodo già visitato la ricorsione si ferma: dato uno stato iniziale definiamo d come la massima profondità raggiunta dagli algoritmi. Possiamo quindi dire che:

$$r = \Theta(d)$$

Con una strategia in ampiezza invece:

$$r = O(s)$$

Usare una strategia in profondità quindi porta ad un risparmio della memoria.