Introduzione

Nel 1994 Leonard Adleman descrive nella sua pubblicazione *Molecular Computation of Solutions to Combinatorial Problems* come sia possibile, attraverso l'uso della biologia molecolare, risolvere problemi matematici attraverso l'uso della biologia. In particolare, viene risolta un'istanza del problema del ciclo hamiltoniano su grafo orientato, problema notoriamente NP-completo. Così, codificando archi e vertici in sequenze casuali di DNA e sfruttando gli strumenti offerti dalla biologia molecolare, Adleman ha per la prima volta risolto un problema matematico attraverso l'uso della biologia. Le critiche a tale esperimento non tardarono ad arrivare, ed infatti già nel 1996 Hartmanis, in *On the Weight of Computations*, spiega quali siano i limiti di tale processo dovuti ad una nuova grandezza da tenere in considerazione: la complessità molecolare. Il peso delle molecole utilizzate per la computazione, infatti, ha un costo che cresce in modo esponenziale rendendo di fatto impraticabili tali esperimenti per grandi istanze dei problemi. Un altro aspetto importantissimo da tenere in considerazione è il fatto che il DNA non è error-free. La scelta effettuata da Adleman di utilizzare codifiche casuali per archi e vertici del grafo si è rivelata corretta solo perchè l'istanza su cui ha effettuato l'esperimento era particolarmente piccola. In generale, a causa di possibili ibridazioni non volute, che invalidano il processo di computazione, bisogna prestare molta attenzione nella scelta della molecole utilizzate per la codifica. Così, oltre all'approccio sperimentale, che consiste nel perfezionamento delle tecniche biologiche utilizzate negli esperimenti attraverso l'analisi di risultati sperimentali, si è sviluppato un ulteriore approccio, via Linguaggi Formali, in cui i filamenti di DNA sono visti come parole sull'alfabeto $\Delta = \{A, C, G, T\}$, che rappresenta le basi del DNA, e le involuzioni sono identificate attraverso proprietà definite su involuzioni sull'alfabeto. Questo problema è noto in letteratura come *"DNA code word design problem"*. Seguendo quest'ultimo approccio sono state formalizzate le proprietà che l'insieme di partenza deve rispettare per evitare le possibili ibridazioni che si potrebbero verificare quando tali proprietà non sono rispettate. La prima proprietà riguarda la codicità dell'insieme di partenza ed è la più importante. Essa consente di avere un'unica fattorizzazione per parole ottenute attraverso la concatenazione di parole dell'insieme di

partenza. Se tale proprietà non fosse verificata, infatti, al termine dei processi di computazione non si avrebbe la possibilità di risalire all'esatta configurazione di partenza. Le altre proprietà di base sono la proprietà θ-*compliant* e la proprietà θ-*freedom*. La prima prevede che la involuzione di un elemento dell'insieme di partenza non deve comparire come fattore di un elemento dell'insieme di partenza stesso. La seconda, invece, prevede che la involuzione di un elemento dell'insieme di partenza non deve comparire come fattore della concatenazione di due elementi dell'insieme di partenza. Se una di queste proprietà non fosse rispettata, durante il processo di computazione potrebbe accadere che, a causa del comportamento che intercorre tra molecole, esse possano legarsi e quindi essere inutilizzabili per la computazione stessa, invalidando quindi il processo di computazione effettuato fino a quel momento. Successivamente sono state definite ulteriori proprietà a dimostrazione del fatto che i possibili errori a causa di ibridazioni non volute in cui è possibile incorrere sono numerosi.

Oggetto di studio di questo lavoro di tesi è stato in particolare [11], in cui le proprietà sopra elencate sono verificate attraverso l'uso di automi. In particolare, viene utilizzato un importante teorema della teoria dei codici [3] che, utilizzando strutture quali Flower automaton e Square automaton, consente di verificare la proprietà di codicità di un insieme. Attraverso una visita sullo Square automaton, infatti, è possibile identificare eventuali path, definiti ambigui, che rendono non verificata la proprietà di codicità. Utilizzando lo stesso principio, gli autori in [11] definiscono due nuovi automi, Theta Flower ed Involution Square, definendo nuovi path ambigui con particolari caratteristiche la cui presenza all'interno degli automi indica che le proprietà che si stanno verificando non sono rispettate dall'insieme che gli automi rappresentano.

Il lavoro di tesi si è concentrato sullo studio dell'algoritmo proposto in [11], con l'obiettivo di una implementazione in Java come suggerito dagli autori stessi. Nella fase implementativa sono stati riscontrati inizialmente problemi relativi all'utilizzo di memoria dell'algoritmo proposto. Particolare attenzione è stata dedicata quindi alla ricerca di una nuova implementazione che ha avuto come obiettivo principale la minimizzazione degli sprechi di memoria senza tralasciare il tempo di esecuzione. Il lavoro ha previsto infine una fase di testing che, attraverso l'esecuzione su istanze particolarmente grandi, ha dimostrato come l'implementazione adottata offre una buona scalabilità sia in termini di spazio necessario che di tempo di esecuzione.

Il lavoro di tesi è organizzato con un primo capitolo in cui vengono esposti i dettagli dell'esperimento effettuato da Adleman. Verrà quindi presentato l'algoritmo da egli stesso proposto in [2] e sarà mostrata una possibile esecuzione dell'algoritmo stesso su un'istanza semplice del problema. Saranno quindi analizzati i pro e i contro di tale esperimento e saranno presentate le due direzioni di ricerca sviluppate per questa problematica: l'approccio sperimentale e quello via linguaggi formali. Verrà seguito

quest'ultimo.

Il secondo capitolo offre una serie di definizioni di base sui linguaggi che saranno utilizzate nel prosieguo del lavoro di tesi. Verranno quindi formalizzate le proprietà che gli insiemi scelti devono rispettare per evitare alcune delle possibili cattive ibridazioni corredandole di esempi. Infine, verranno introdotte, sempre con relativi esempi, le strutture Flower automaton e Square automaton, così come definite in [3].

Il terzo capitolo è dedicato alle dimostrazioni di correttezza delle varie proprietà. Per quanto riguarda la proprietà di codicità, come già detto, viene utilizzato un importante teorema dalla teoria dei codici [3]. Successivamente vengono definite le strutture Theta Flower automaton ed Involution Square automaton e vengono formalizzate le dimostrazioni per la verifica delle proprietà così come fatto in [11] per le rispettive versioni *strictly*. In questo capitolo vengono lasciate in sospeso le dimostrazioni di correttezza relative alle due proprietà, θ-*compliant* e θ-*freedom*, nella loro forma non *strictly*. Per esse non si è giunti ad una dimostrazione formale ma piuttosto ad una serie di esempi che cercano di coprire tutti i possibili casi in cui è necessario effettuare delle verifiche. Per la proprietà θ-*compliant*, invece, non c'è bisogno di nessuna dimostrazione essendo un caso banale. Questi esempi saranno ripresi in un apposito paragrafo nel quarto capitolo.

Nel quarto capitolo viene riportato lo pseudocodice dell'algoritmo, leggermente modificato rispetto a quanto riportato in [11]. Oltre alla procedura generica, che definisce una visita sull'automa prodotto, vengono analizzati i controlli effettuati per la verifica delle varie proprietà. Per la proprietà θ-*freedom*, nella sua forma non strictly, sono stati trovati degli esempi che cercano di coprire tutti i casi di interesse. Non avendo comunque espresso una dimostrazione formale relativa alla correttezza di tale processo di verifica, si è preferito non implementare i controlli relativi a tale proprietà.

Nel quinto capitolo vengono riportate le principali classi e i principali metodi di interesse dell'algoritmo. Avendo definito una interfaccia che definisce le operazioni di base necessarie all'esecuzione dell'algoritmo stesso, vengono proposte e discusse le varie implementazioni realizzate. Brevemente, la scelta più ovvia di rappresentare tali automi attraverso grafi (nella loro rappresentazione tramite liste di adiacenza) verrà discussa, ma scartata in favore di un'altra rappresentazione che rappresenta un buon compromesso tra tempo di esecuzione ed utilizzo di memoria. Infatti, in qualunque modo si tentino di rappresentare gli automi, il costo relativo ai riferimenti ad oggetti gestiti della JVM non consente di avere una buona scalabilità per istanze particolarmente grandi dell'input. La soluzione adottata, quindi, è stata quella di evitare l'utilizzo di oggetti ed utilizzare tipi primitivi per la realizzazione delle strutture che rappresentano gli automi. Per fare questo è stata adattata una delle strutture realizzate durante il corso di Strutture Dati, un array gestito in modo circolare per memorizzare

gi archi dell'automa, utilizzando i tipi primitivi come elementi di tale classe. Oltre all'efficienza nel salvataggio delle strutture in memoria è stata realizzata anche un'altra implementazione in cui le strutture che rappresentano gli automi prodotto sono salvate direttamente su disco. Per poter decidere direttamente a tempo di esecuzione quale effettiva implementazione utilizzare, è stata scritta una interfaccia che definisce i metodi necessari all'esecuzione dell'algoritmo e la scelta dell'implementazione si basa sul numero di archi dell'automa prodotto.

Infine, il sesto ed ultimo capitolo, riporta i dati raccolti durante la fase di testing: inizialmente sono state realizzate piccole istanze del problema cercando di riportare in particolare casi limite in modo da poter escludere errori su queste istanze (cioè, casi particolari che non venissero trattati dall'implementazione). Successivamente il testing è stato effettuato su istanze particolarmente grandi così da poter osservare tempo di esecuzione e consumo di memoria dell'algoritmo. Il lavoro di tesi si conclude con l'introduzione di alcuni possibili sviluppi futuri dell'algoritmo e del progetto.

Bibliografia

[1] Leonard Adleman. Wikipedia, L'enciclopedia libera. Tratto il 23 gennaio 2012, 15:29 da `it.wikipedia.org/w/index.php?title=Leonard_Adleman&oldid=45885393`.

[2] Adleman L. M., *Molecular Computation of Solutions to Combinatorial Problems*, 1994, Science, Vol.266, pp.1021-1024.

[3] Bertsel J., Perrin D. *Theory of Codes*, 1985, Academic Press, Orlando Florida.

[4] Braich R.S., Chelyapov N., Johnson C., Rothemund P.W.K., Adleman L. M., *Solution of a 20-Variable 3-SAT Problem on DNA Computer*, Sciencexpress, 2002.

[5] Deaton R., Garzon M., Rose J., Franceschetti D.R., Stevens S.E. Jr., *Good Encodings for DNA-based Solutions to Combinatorial Problems*, DNA Based Computers II, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, Vol.44, American Mathematical Society, pp.131-140, 1996.

[6] Garzon M., Deaton R., Neathery P., Murphy R.C., Franceschetti D.R., Stevens S.E. Jr., *On the Encoding Problem for DNA Computing*, DNA Based Computers III, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, Vol.48, American Mathematical Society, pp.230-237, 1997.

[7] Garzon M., Neathery P., Deaton R., Murphy R.C., Franceschetti D.R., Stevens S.E. Jr., *A New Metric for DNA Computing*, Second Annual Conference on Genetic Programming, 1997.

[8] Hartmanis J., *On the Weight of Computations*, 1995, Bulletin of European Association on Theoretical Computer Science, n.55 pp.136-138.

[9] Hopcroft J.E., Motwani R., Ullman J.D., *Automi, Linguaggi e Calcolabilità*, Addison Wesley, 2009.

[10] Kari L., Kitto B., Thierrin G., *Codes, Involutions and DNA Encodings*, Formal and Natural Computing LNCS 2300, 2002.

[11] Kephart D.E., LeFevre J. *CodeGen: The Generation and Testing of DNA Code Words*, 2004, to appear in IEEE.

[12] `http://www.cs.virginia.edu/kim/publicity/pldi09tutorials/memory-efficient-java-tutorial.pdf`