RELAZIONE SUL PROGETTO DI ALGORITMI E STRUTTURE DATI
AA 2016/2017

Alessandro Martin
N. Matricola 120651

Sommario:

ISTRUZIONI PER LA COMPILAZIONE

i sorgenti e i compilati sono presenti nella directory "progetto1617"

se si volesse ricompilare i sorgenti java:
Eliminare (qualora ce ne fossero) tutti i file .class lanciando il comando

rm *.class

entrare nella cartella progetto1617 e lanciare questo comando:

javac *

se si vuole eseguire il progetto
spostarsi sulla cartella superiore alla cartella progetto1617 e eseguire il seguente comando:

java -cp ./ progetto1617/Main "T"

dove T è il testo in input

DESCRIZIONE DEL PROBLEMA DA RISOLVERE:
Dato il testo T e il grafo G generato a partire dal testo, calcolare la massima lunghezza di un cammino tra due nodi in G

STRUTTURE DATI:
Per la gestione del Grafo è stato creato un oggetto apposito (chiamato Grafo), la cui implementazione è accessibile nel file Grafo.java.
L'oggetto **Grafo** è implementato memorizzando nel suo stato 3 elementi:

- Una Mappa (Nodi -> Archi) che viene usata per memorizzare i nodi del grafo come stringa e per ogni nodo una lista di archi che rappresenta la lista di adiacenza del nodo, in questa struttura i nodi sono chiavi e sono rappresentati come stringhe.
- Un vettore di archi che rappresenta la lista di adiacenza dei nodi mappati come numeri interi.
- Un intero che memorizza il numero di archi presenti in ogni momento nel grafo

Visto che gli archi sono dei collegamenti tra i nodi del grafo anch'essi sono stati implementati creando due oggetti, ***Arco*** **e** ***ArcoIntero***, che nello stato racchiudono una coppia di stringhe rappresentante i 2 nodi ordinati secondo il verso della freccia.
Il grafo G è generato tenendo conto la proprietà espressa dalla definizione 2 del progetto.
Esso è sempre un DAG, cioè un grafo che ha le seguenti caratteristiche:

1) G è un grafo diretto
2) G è un grafo che non ha alcun ciclo, quindi se A e B sono 2 nodi di G e ed esiste un cammino da A verso B allora non esiste un cammino che da B ci riporta ad A o viceversa.

L'oggetto Mappa mette in relazione una stringa a una lista di adiacenza implementata come Vector di Vector ed è utile per mantenere la struttura del Grafo con i nodi memorizzati come Stringa per effettuare il print del Grafo in formato dot.

L'oggetto Coda è stato implementato usando i Vector e l'implementazione comprende le operazioni di enqueue e dequeue, isEmpty e il pretty print.

INPUT

Questa sezione della relazione presenta la spiegazione dettagliata di come viene trattato l'input e come esso contribuisce alla costruzione del grafo che poi verrà usato negli algoritmi per risolvere il problema.

L'input viene inserito da linea di comando, il testo T non è altro che una stringa racchiusa tra doppi apici che viene trattata come un singolo elemento leggendo il vettore di input Args[].

Il testo viene parsato tenendo conto le regole stabilite dal progetto, pertanto le singole parole sono individuate tramite un semplice split usando come sepatore il carattere di spazio al fine di creare un array di parole.
Questo comporta la possibilità di utilizzare, nel testo, qualunque carattere rappresentabile sulla shell, anche se è consigliato non usare caratteri particolari (soprattutto quelli dedicati al linguaggio dot, come per esempio "{" oppure il ';', in quanto potrebbero creare problemi se si vuole disegnare il risultato).

Dall'array delle parole viene estrapolato l'alfabeto, che nel codice è semplicemente un array di caratteri (ordinati per codice ascii) che sono contenuti nel testo.

Dopo la determinazione dell'alfabeto, viene istanziato l'oggetto grafo ed esso viene riempito utilizzando l'array delle parole splittate come nodi, in questo passo viene anche effettuato un test per controllare che il nodo non sia già presente all'interno del grafo, se lo è non viene inserito. *Questo mi assicura che nel grafo non siano presenti nodi ripetuti, ogni stringa è inserita nel grafo solamente una volta*.

La creazione degli archi nel grafo è effettuata utilizzando il metodo *generaArchi()* che si occupa di controllare, per ogni coppia di nodi presenti nel grafo, se essa soddisfi la proprietà della Definizione 2 del progetto.

Quindi per tutte le stringhe viene generato il proprio vettore di caratteri che viene confrontato con gli altri per creare o meno l'arco tra i nodi.

Ogni nodo presente nel grafo G viene poi mappato su numeri interi attraverso la procedura *mappaNodi()*, riempiendo nell'oggetto grafo la lista di adiacenza con nodi rappresentati come interi.

Dopo questo passo, il grafo generato a partire dal testo T è sempre un grafo diretto aciclico ed esso può essere usato per l'applicazione degli algoritmi della soluzione che saranno spiegati nella sezione successiva.

SOLUZIONE DEL PROBLEMA

In questa sezione verrà trattata la soluzione al problema del progetto e verranno spiegati gli algoritmi utilizzati.

Sia G il grafo (diretto aciclico) determinato dall'input del testo T,
Per risolvere il problema del progetto ho utilizzato 2 algoritmi:

1- L'algoritmo di Kahn per la determinazione di un ordinamento topologico del grafo G
2- Un algoritmo che compie una visita in profondità del grafo G tenendo conto, come ordine di partenza, l'ordinamento topologico al fine per determinare il cammino più lungo tra due nodi di G.

Algoritmo di Kahn

L'algoritmo di Kahn è utilizzato per ottenere uno degli infiniti ordinamenti topologici dei nodi di un grafo diretto aciclico.
L'algoritmo di Kahn sfrutta il seguente teorema:
"*Se abbiamo un grafo G diretto aciclico allora esiste almeno un nodo che non ha archi entranti e almeno un nodo che non ha archi uscenti*".
Questo è vero perché se G è un DAG e quindi è aciclico per definizione, allora tutti i cammini tra i nodi di G hanno lunghezza finita. Prendiamo un cammino S tra due nodi di G che sia di lunghezza massima possibile e che questo cammino abbia come sorgente un nodo u e come destinazione un nodo v. Il nodo u non deve avere archi entranti in quanto se u avesse un arco entrante allora esisterebbe almeno un nodo con un arco verso u e quindi il cammino S non sarebbe stato di lunghezza massima. La stessa considerazione può essere estesa anche al nodo v che, essendo la destinazione, deve avere zero archi uscenti, infatti se questo avesse avuto archi uscenti avremo l'esistenza di un arco che collega v a un altro nodo, ma questo significherebbe che S non era un cammino di lunghezza massima.

Fissato questo teorema
L'algoritmo crea due code:

- **nodizero**: una coda contenente, durante l'esecuzione dell'algoritmo, tutti i nodi che non hanno alcun arco entrante
- **Topologico**: una coda che conterrà l'ordinamento topologico alla fine dell'esecuzione

E compie i seguenti passi:

- Calcolo tutti i gradi entranti di tutti i nodi presenti nel grafo e li inserisco in un vettore di interi chiamato *indegrees,* quindi inserisco nella coda **nodizero** tutti i nodi che nel vettore indegrees hanno zero archi entranti richiamando la procedura *nodiZero*(*indegrees*). Per il teorema visto sopra la coda, a questo primo passo, si riempirà con almeno un nodo.
- Per ogni nodo presente all'interno della coda **nodizero** svolgo i seguenti passi:
  1: Inserisco il nodo in **Topologico** ed estraggo la lista di adiacenza del nodo
  2: Per ogni arco presente in questa lista diminuisco nel vettore *indegrees* gli archi entranti del nodo cui punta l'arco di 1, se questa diminuzione fa diventare l'indegree del nodo 0 lo inserisco immediatamente in coda ai nodi zero.

Il risultato dopo questo ciclo è un ordinamento topologico del grafo

Algoritmo di determinazione del cammino più lungo

L'algoritmo di determinazione del cammino più lungo si basa su una variante semplificata (senza colori e senza tempi) di visita in profondità del grafo al fine di calcolarla ricorsivamente. Analogamente a DFS anche l'algoritmo scritto per risolvere il problema è composto principalmente da due procedure, la prima chiamata *lunghezzaMassima()* che si occupa di istanziare le strutture dati utilizzate poi nella visita vera e propria *visitaDistanze().*

Idea dell'algoritmo:
Supponiamo di avere il grafo diretto aciclico $G = (V, E)$ i cui nodi sono numeri interi e il suo ordinamento topologico è all'interno di una coda.

Per ogni nodo presente all'interno della coda dell'ordinamento topologico viene fatta partire una visita in profondità, e questo mi assicura di visitare tutti i nodi del grafo anche se non connesso.
La visita in profondità, nel codice definita come *visitaDistanze(),* è una visita che parte da un nodo indicato come parametro, e durante l'attraversamento in profondità aggiorna tutti i valori nel vettore delle distanze dei nodi attraversati, tenendo conto della distanza dal nodo di partenza se essa è superiore a quella indicata nel vettore.
Ad ogni chiamata ricorsiva della visita a un nodo adiacente la distanza di partenza della visita (indicata dalla variabile *start)* viene aumentata di 1.
Grazie ad un test, all'inizio della *visitaDistanze()* viene comparata la distanza *start* alla distanza presente nel vettore delle distanze per il nodo che stiamo attraversando.
La conseguenza di ciò è che la visita non viene effettuata se la distanza già presente nel vettore è maggiore della distanza attuale della visita facendo risparmiare chiamate ricorsive che provocherebbero attraversamenti inutili che non modificherebbero le distanze nel vettore.
Infatti leggere una distanza maggiore a *start* nel vettore distanze significa non solo aver già effettuato una visita in profondità da quel nodo, ma anche che tutte le distanze nei nodi che sarebbero visitati in profondità sono state precedentemente aggiornate con distanze maggiori a quelle che otterrei facendo la visita attuale.

Calcolo della complessità

In questa sezione saranno analizzate le complessità dei due algoritmi utilizzati per risolvere il problema determinato dal progetto.

Algoritmo di Kahn

1a) Calcolo grado in entrata di tutti i nodi del grafo, equivale a una scansione completa del grafo, complessità pari a $\Theta(|V| + |E|)$
1b) selezione tra tutti i nodi di quelli con grado in entrata pari a zero, equivale a una scansione di tutti i nodi $\Theta(|V|)$
2) Il ciclo while esterno viene eseguito per ogni nodo che non ha archi entranti, ogni volta che un nodo di questa coda viene preso vengono diminuiti i gradi in entrata dei nodi che avevano un arco a partire dal nodo estratto. Quindi prima o poi ogni nodo nel grafo entrerà nella coda e ci rimarrà fintantò non sarà estratto, dopo l'estrazione dalla coda il nodo non può più rientrare.
Pertanto questo ciclo viene eseguito $\Theta(|V|)$ volte
3) il ciclo while interno viene eseguito per ogni arco nella lista di adiacenza del nodo estratto dalla coda, pertanto viene eseguito al più $\Theta(|adj[nodo]|)$ volte.
4) complessivamente while interno ed esterno hanno costo pari a
$\sum_{nodo \in G} \Theta(|adj[nodo]|) = \Theta(|E|)$

la complessità di Kahn è pari a $\mathrm{O}(V + E)$

Algoritmo di visita

lunghezzaMassima()
1) l'inizializzazione presente in *lunghezzaMassima()*, richiede di istanziare un vettore che contiene, per ogni nodo, le distanze attuali delle visite, pertanto questa richiede un tempo lineare al numero di nodi $O(|V|)$.
2) il ciclo while viene eseguito per ogni nodo nella coda dell'ordinamento topologico, pari a tutti i nodi del grafo, quindi $|V|$ volte

3) La *visitadistanze*() come per la visita in profondità classica viene lanciata esattamente una volta per ogni nodo presente nel grafico, quindi $\Theta(|V|)$ e l'esecuzione della visita ha un costo pari alla cardinalità lista di adiacenza del nodo sul quale viene chiamata, con un tempo totale di $\Theta(|E|)$

la complessità per la visita è $\Theta(V + E)$

la complessità totale dell'algoritmo per calcolare le distanze è pertanto di $O(V + E)$

Calcolo dei tempi

Questa sezione è dedicata alla generazione dei casi di test e al calcolo dei tempi per gli algoritmi che risolvono il problema.
Per il calcolo dei tempi di esecuzione sono stati utilizzati gli algoritmi delle dispense: *calcolarip(), tempomedionetto(), misurazione()*.
Per calcolare la granularità è stata utilizzata la funzione *System.currentTimeInMillis(),* la quale estrae il tempo del sistema attuale in millisecondi.
L'errore massimo impostato nella misurazione è del 5% quindi la chiamata a *misurazione()* avviene con α=1,96.
Per calcolare il tempo di esecuzione, l'algoritmo è stato eseguito su un portatile con CPU Core i7 6700HQ@2.4Ghz e sistema operativo MacOs.

I tempi sono stati analizzati generando casi di test:
- Su stringhe di DNA, cioè stringhe formate da sequenze casuali di lunghezza compresa tra 1 e 25 dei 4 nucleotidi, quindi $\Sigma = \{A, C, T, G\}$.
- Su grafi generati casualmente indicando numero di archi e nodi da generare.

Generazione casuale:

Sequenze di DNA:
Il metodo *sequenzeDNA()* si occupa di generare *numseq* sequenze di DNA casuali lunghe tra 1 e 25 caratteri.
Per numseq volte viene generato un numero casuale tra 1 e 25, che determina la lunghezza di volta in volta di ogni sequenza, quindi per ogni carattere fino a questa lunghezza viene scelto casualmente un carattere nell'insieme formato dall'alfabeto dei 4 nucleotidi $\Sigma = \{A, C, T, G\}$
Viene prima generato un vector di sequenze, poi il vector viene trasformato in stringa, ogni sequenza è separata da uno spazio, e questa stringa rappresenta un testo che può essere dato in pasto agli algoritmi per la soluzione.

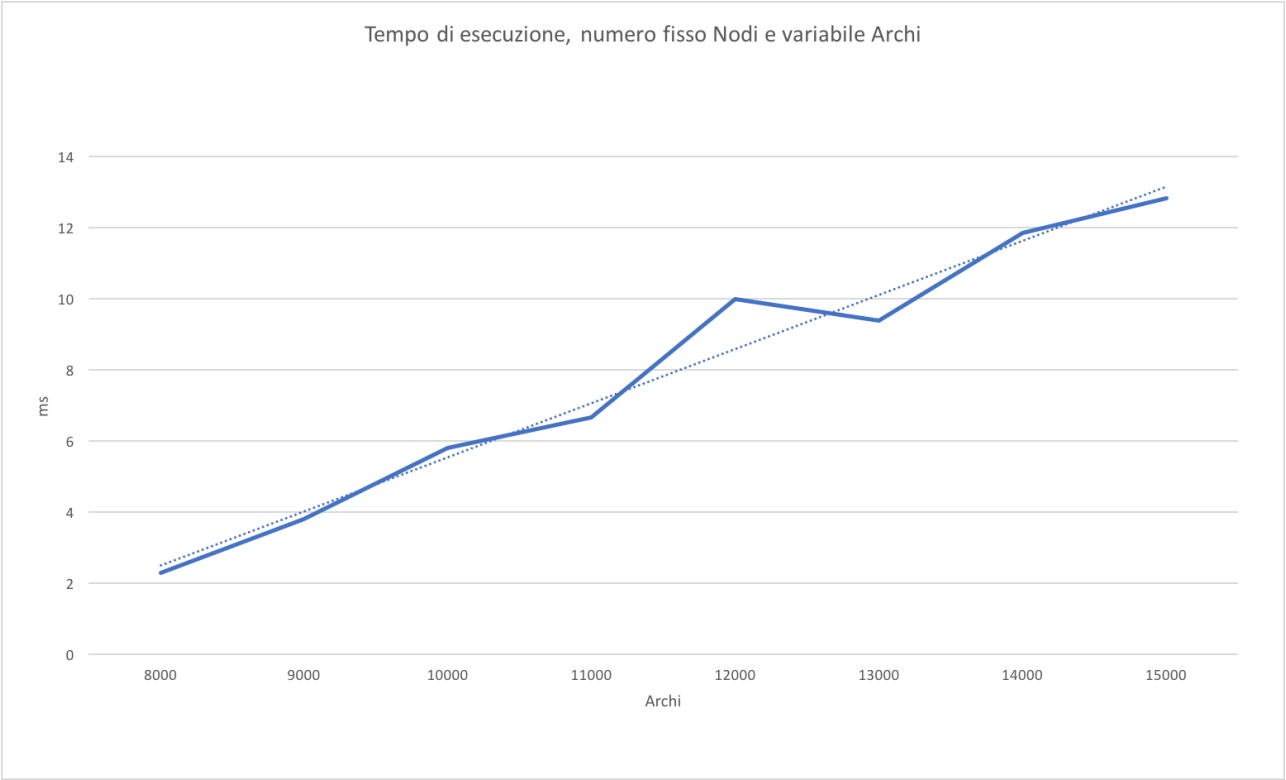
Grafi generati casualmente:

Il metodo *dagCasualeArchi()* si occupa di generare in maniera diretta grafi diretti aciclici indicando un numero di nodi e numero di archi.
L'unico vincolo che deve essere soddisfatto per la generazione è che dato il numero n di nodi da generare, il numero di archi indicato in input non deve superare $\frac{n(n-1)}{2}$ archi, valore equivalente al massimo numero di archi di un DAG.

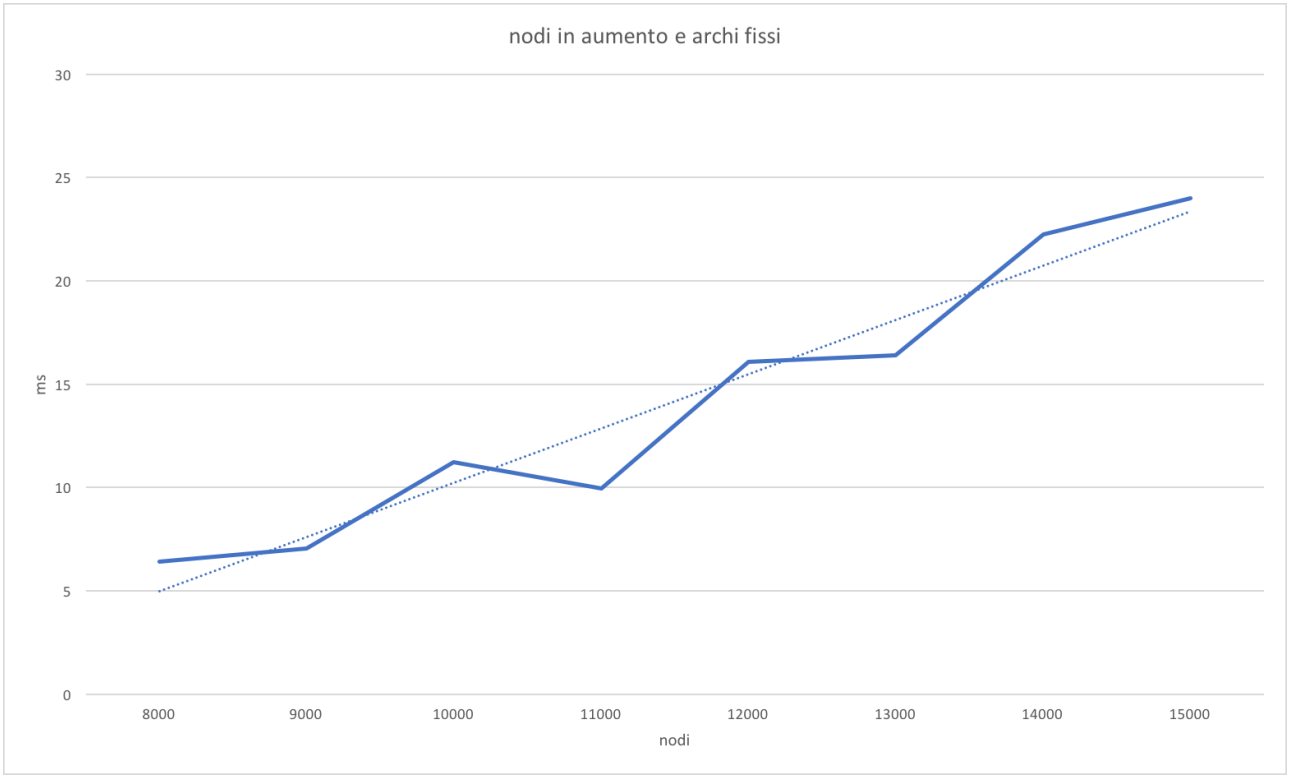
Questa generazione sfrutta la proprietà fondamentale secondo la quale i grafi diretti aciclici sono grafi aventi matrice di adiacenza triangolare inferiore, pertanto generare un DAG si riconduce al problema più semplice di:

1. Dato il numero di nodi n da generare, creare una matrice di adiacenza n*n con 0 su ogni cella
2. Dato il numero di archi, per ogni elemento presente nella matrice triangolare inferiore, generare casualmente 1 fino a raggiungere la quantità di archi desiderati tra i nodi

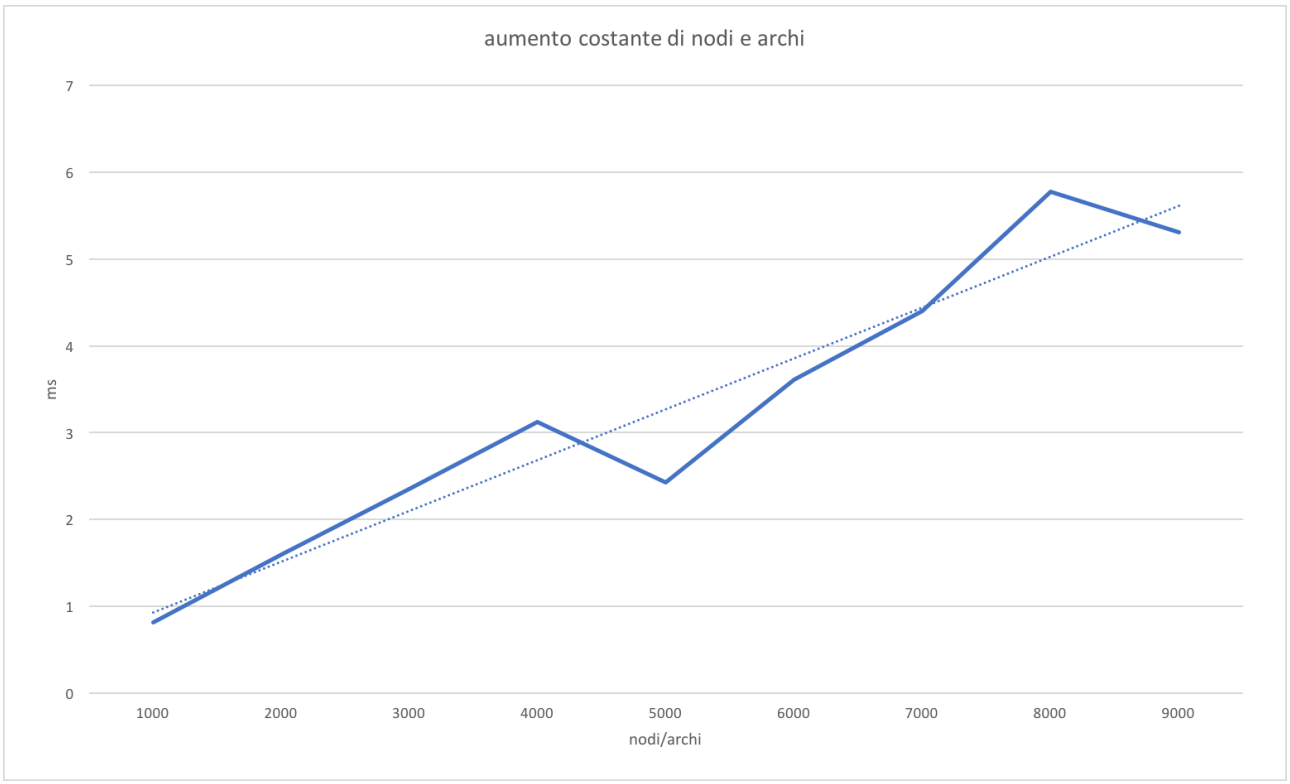
Grafici sul calcolo dei tempi



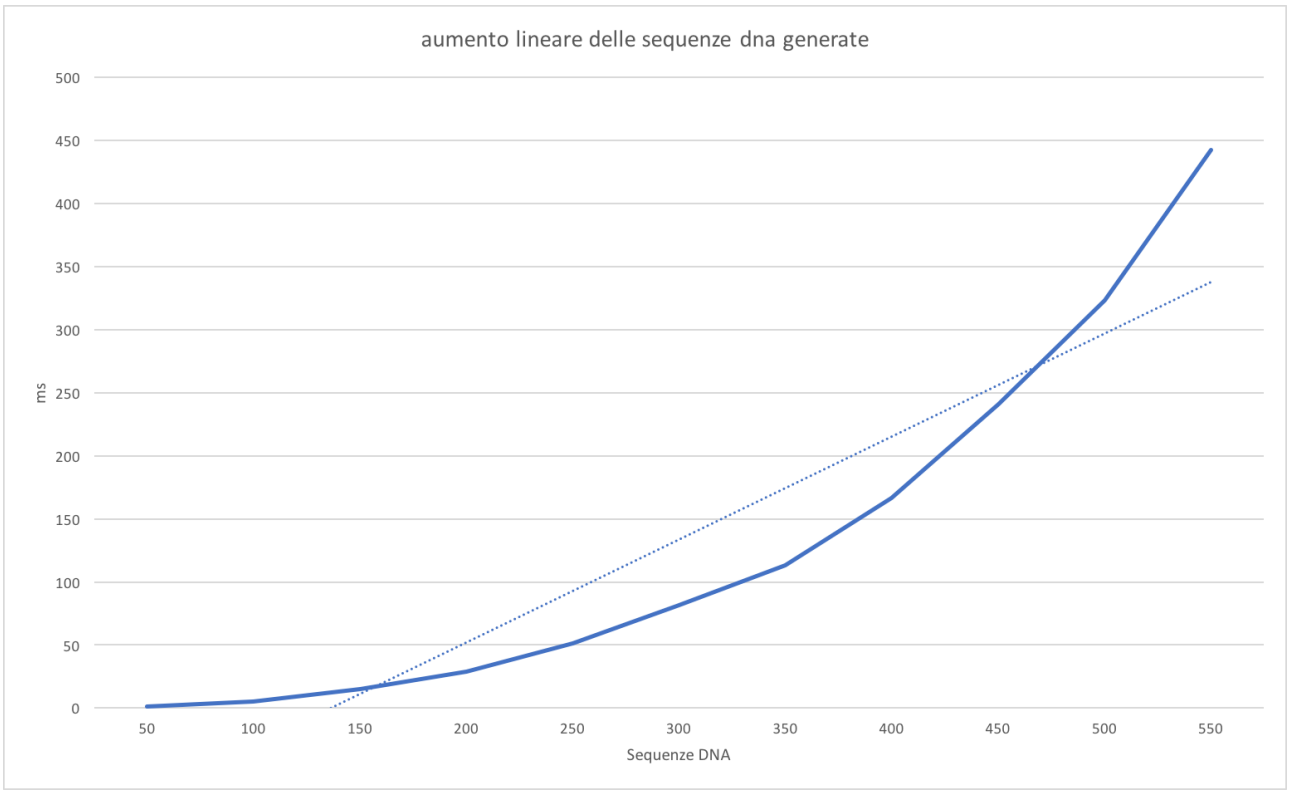
nodi	archi	ms
500	8000	2,300
500	9000	3,800
500	10000	5,820
500	11000	6,675
500	12000	10,000
500	13000	9,400
500	14000	11,850
500	15000	12,833



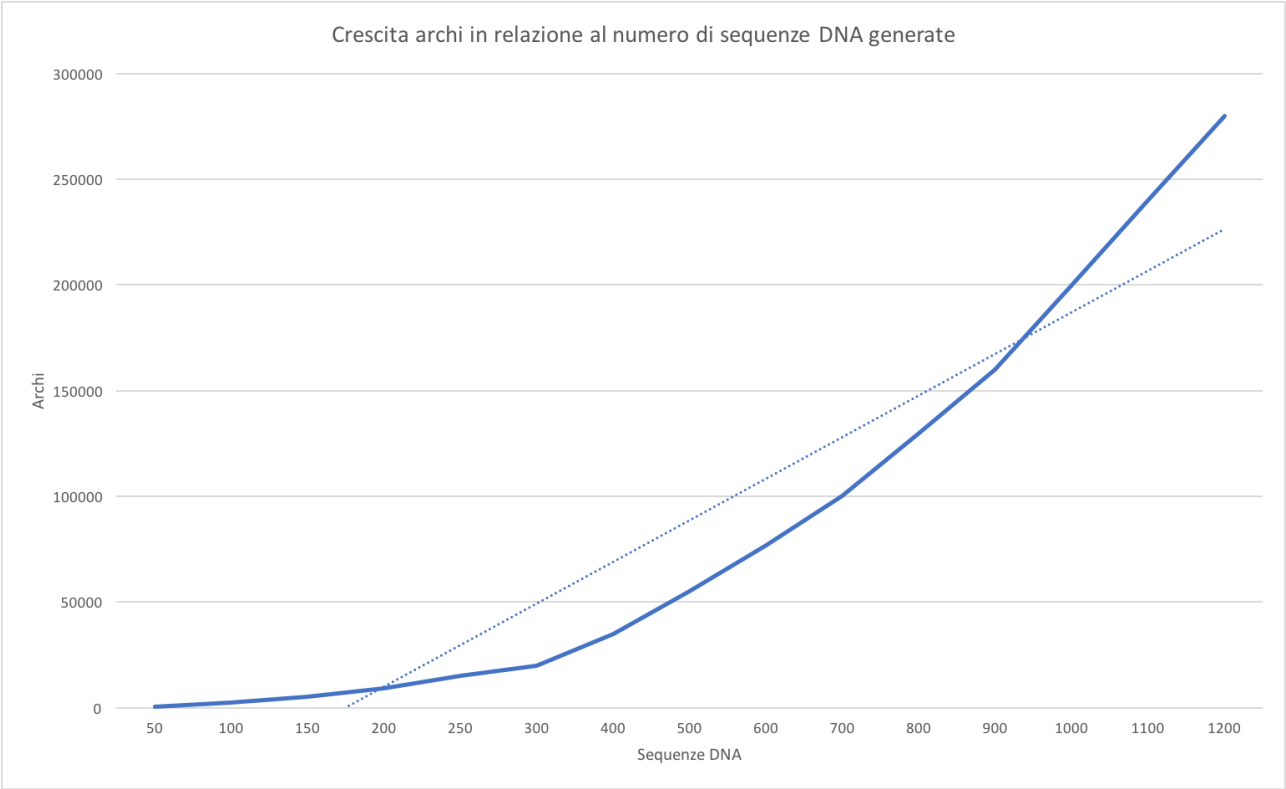
nodi	archi	ms
8000	500	6,40
9000	500	7,06
10000	500	11,22
11000	500	9,96
12000	500	16,08
13000	500	16,40
14000	500	22,25
15000	500	24,02



nodi	archi	ms
1000	1000	0,81
2000	2000	1,6
3000	3000	2,35
4000	4000	3,1212
5000	5000	2,43
6000	6000	3,6146
7000	7000	4,406
8000	8000	5,78
9000	9000	5,312



sequenze	tempo (ms)
50	1
100	5,2
150	14,8
200	28,8
250	51,6
300	81,4
350	113,4
400	166,6
450	240,6
500	323,1518
550	442,6617



Crescita archi con dna	
sequenze	archi
50	600
100	2300
150	5000
200	9000
250	15000
300	20000
400	35000
500	55000
600	77000
700	100000
800	130000
900	160000
1000	200000
1100	240000
1200	280000

Considerazioni sulle misurazioni del tempo

I primi 3 grafici che fanno riferimento a aumenti lineari solo di nodi, oppure solo di archi, oppure in contemporanea di archi e nodi mostrano una sostanziale linearità dell'algoritmo per la determinazione della soluzione in accordo con l'analisi di complessità.

Più interessante il caso che riguarda la generazione casuale di sequenze di DNA
Si nota come l'aumento lineare delle sequenze generate produca una crescita polinomiale nel tempo di esecuzione.
Questo è dovuto al metodo di generazione casuale delle sequenze di DNA.
A causa del basso numero di caratteri nell'alfabeto e della lunghezza massima delle sequenze generate, un aumento del numero di sequenze da generare lineare produce un aumento molto rapido del numero di archi presenti nel grafo che cresce in accordo alla crescita temporale dell'algoritmo della soluzione.