Università degli Studi di Camerino Scuola di Scienze e Tecnologie Corso di Laurea in Informatica Corso di Algoritmi e Strutture Dati 2021/2022 Studente: Michela Di Biase

RELAZIONE PROGETTO TOTALE N. 2

INDICE

Obiettivi del Progetto

Il progetto consiste nei seguenti Task:

1. Implementare in Java i metodi di una classe che rappresenta **strutture secondarie di RNA**.
2. Implementare in Java, usando la **programmazione dinamica**, l'algoritmo di **Nussinov-Jacobson** che trova il massimo numero di legami possibili data una sequenza di nucleotidi che forma una struttura secondaria di RNA senza pseudonodi.
3. **Valutare numericamente le prestazioni** dell'algoritmo implementato al punto precedente.
4. Realizzare in Java un'implementazione di una **coda con priorità** implementata tramite uno **heap binario**.
5. Realizzare in Java una implementazione di un **grafo generico orientato** con matrice di adiacenza.
6. Per il problema dei cammini minimi con sorgente singola, realizzare in Java una implementazione dell'**algoritmo di Dijkstra** che usi la coda di priorità implementata nel punto 4.

Parte 1: Strutture secondarie di RNA

L'acido ribonucleico (RNA) è un polimero lineare composto da quattro differenti tipi di nucleotidi che sono collegati tra loro da un legame forte, i quattro tipi di nucleotidi sono: Adenina (A), Guanina (G), Citosina (C) e Uracile (U).
Esistono anche tipi di legami deboli sono di base di Watson e Crick (G-C o C-G e A-U o U-A) o le coppie "oscillanti" - wobble in inglese - (G-U o U-G). Questi legami, detti coppie ammissibili di una struttura secondaria.

La classe SecondaryStructure rappresenta una struttura secondaria. E' possibile aggiungere dei legami deboli con il metodo addBond e controllare se sono presenti pseudonodi utilizzando IsPseudoknotted().

```
public boolean addBond(WeakBond b) {

    if (b == null) {
        throw  new NullPointerException();
    }

    if(b.getI() < 1 || primarySequence.length() < b.getJ()){
        throw  new IndexOutOfBoundsException();
    }

    char charI = primarySequence.charAt(b.getI()-1);
    char charJ = primarySequence.charAt(b.getJ()-1);

    if (!(isValid(charI, charJ, validN1: 'G', validN2: 'C') ||
            isValid(charI, charJ, validN1: 'A', validN2: 'U') ||
            isValid(charI, charJ, validN1: 'U', validN2: 'G'))
    ) {
        throw new IllegalArgumentException();
    }

    return this.bonds.add(b);

}
```

```java
public boolean isPseudoknotted() {

    List<WeakBond> bondList = new ArrayList<>(this.bonds);
    for(int i = 0; i < bondList.size(); i++) {
        WeakBond b1 = bondList.get(i);
        for(int j = i+1; j < bondList.size(); j++) {
            WeakBond b2 = bondList.get(j);
            /*
            i' < i < j < j', cioè la coppia (i', j') "contiene" la coppia (i, j);
            i < i' < j' < j, cioè la coppia (i, j) "contiene" la coppia (i', j');
            j' < i, cioè la coppia (i', j') "precede" la coppia (i, j);
            j < i', cioè la coppia (i', j') "segue" la coppia (i, j);
            */
            if (!((b2.getI() < b1.getI() && b1.getI() < b1.getJ() && b1.getJ() < b2.getJ()) ||
                    (b1.getI() < b2.getI() && b2.getI() < b2.getJ() && b2.getJ() < b1.getJ()) ||
                    (b2.getJ() < b1.getI())||
                    (b1.getJ() < b2.getI()))) {
                return true;
            }
        }
    }
    return false;
}
```

Grazie al metodo getDotBracketNotation è possibile ottenere una stringa che rappresenta la notazione dotBracket.

```java
public String getDotBracketNotation() {

    if(this.isPseudoknotted())
        throw new IllegalStateException();

    // AAGACCUGCACGCUAGUU
    //.(((..).(..))).....
    StringBuilder bondsNotation = new StringBuilder();
    for(int i = 0; i < this.primarySequence.length(); i++)
        bondsNotation.append(".");

    for (WeakBond b : this.bonds) {
        bondsNotation.setCharAt( index: b.getI()-1,  ch: '(');
        bondsNotation.setCharAt( index: b.getJ()-1,  ch: ')');
    }

    return this.primarySequence + "\n" + bondsNotation;
}
```

Parte 2: Algoritmo di Nussinov-Jacobson

E’ stato implementato l’algoritmo di Nussinov-Jacobson per il folding della struttura secondaria.

Prima di tutto creiamo la matrice di nussinov jacobson, con dimensioni pari al numero di nucleotidi (n) della sequenza primaria. Ogni cella in posizione (i, j) della matrice conterrà il numero massimo di legami deboli per una struttura secondaria relativa alla sottosequenza (i+1, j+1).
Per questo, il numero massimo di legami deboli per l’intera sequenza si troverà in posizione (0, n-1). La costruzione della matrice è implementata utilizzando la ricorsione definita nella traccia.

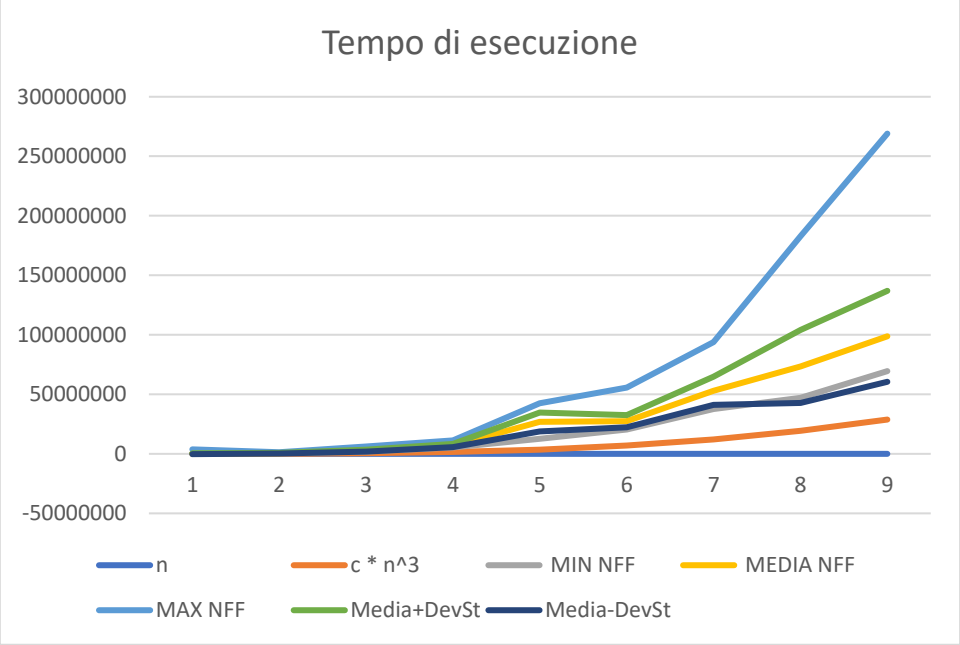
Un frammento di codice del metodo fold()

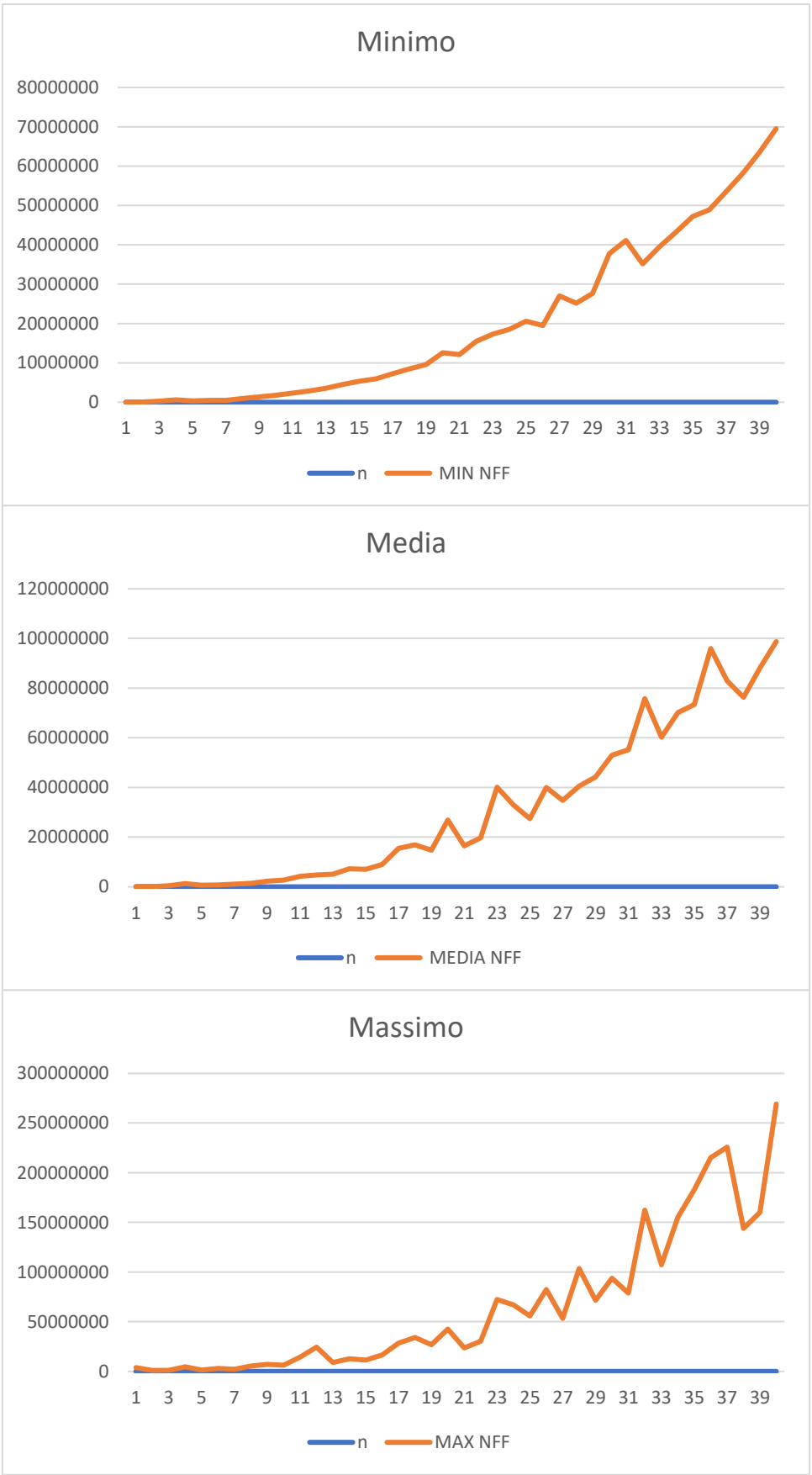
```
for(int n = 1; n < primarySequence.length(); n++) {// prende la stringa per iniziare a controllare i
    // neuclotidi (Si noti che il primo nucleotidi  ha posizione 1)
    for(int i = 0; i < primarySequence.length()-n; i++) {// inizio a controllare le colonne
        int j = i+n;//La matrice di Nussinov-Jacobson N e` una matrice di dimensione n × (n + 1)
        // assegna come massimo il i, j-1 della matrice
        int max = nussinovJacobson[i][j-1];
        // in questo for troviamo il valore di val
        for(int k = i; k < j; k++) {
            int val = -1;//il valore di val quando non ci sono legami
            //controlla se ci sono legami
            if(areValid(primarySequence.charAt(k), primarySequence.charAt(j)))
                if(k > 0)
                    val = nussinovJacobson[i][k-1] + nussinovJacobson[k+1][j-1] + 1;
                else
                    val = nussinovJacobson[k+1][j-1] + 1;
            if (val > max) {// se e piu grande val
                max = val;// allora val diventa il nuovo massimo
            }
        }
        nussinovJacobson[i][j] = max;// inserisco il nuovo massimo sulla matrice in posizione i e j
    }
```

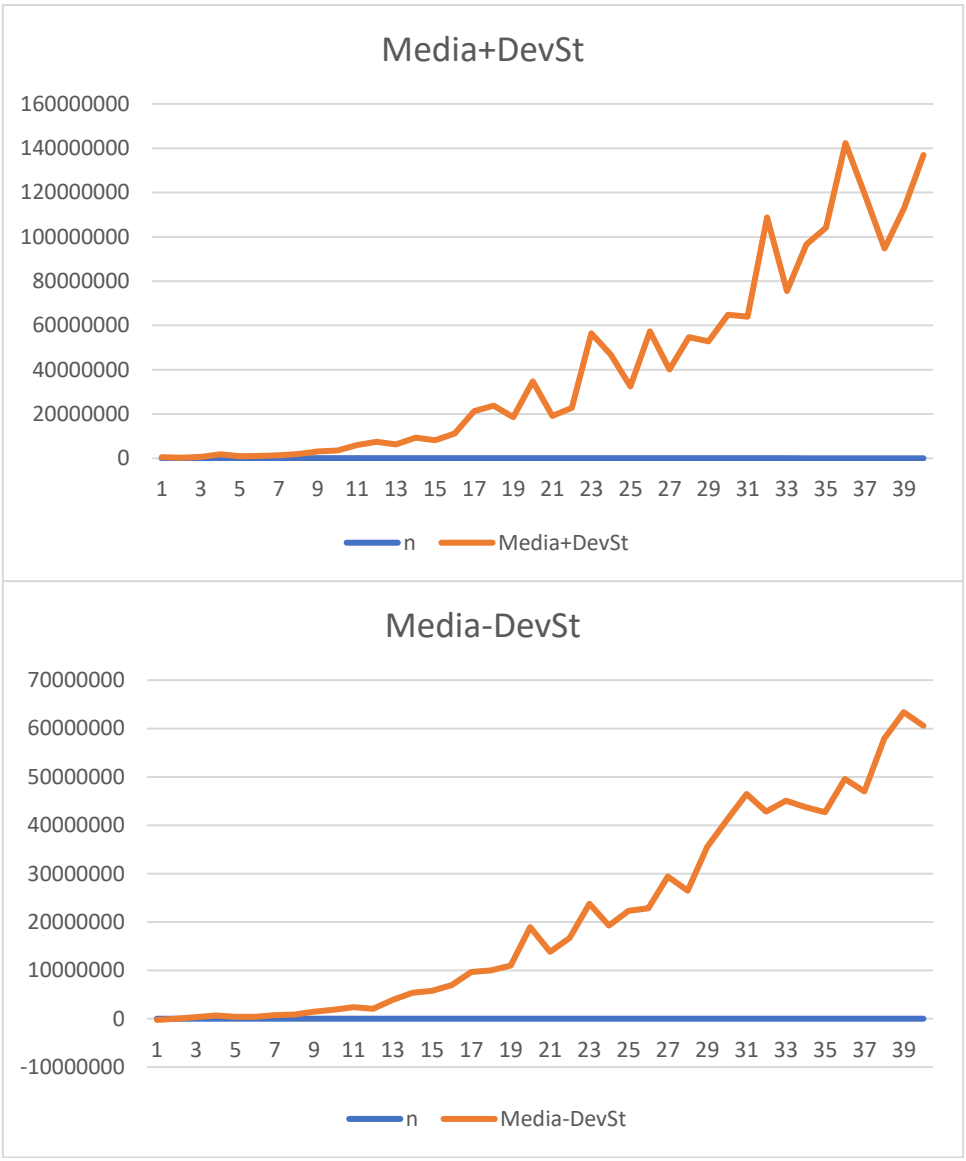
Il metodo traceback(i, j) ci permette ricorsivamente di ricostruire una struttura ottima partendo dalla cella in posizione (0, n-1). La ricorsione in questo caso si muove al contrario rispetto a quella della costruzione della matrice, e si ferma quando i > j. Ad ogni passo se possibile viene aggiunto un legame debole alla struttura secondaria.

```java
private void traceback(int i, int j) {
    if(j <= i)
        return;
    if(nussinovJacobson[i][j] == nussinovJacobson[i][j-1]) {// se in valore in posizione [i][j]
        // e uguale a quello [i][j-1]
        traceback(i,  j: j-1);// faccio la ricorsione e diminuisco la j
        return;
    }
    for(int k = i; k < j; k++) {// a k gli viene assegnatoa  il valore della colonna i
        int val = 0;//assegnamo 0 a val
        if(k > 0) { // se k e maggiore di zero
            val = nussinovJacobson[i][k-1];// val prende il valore della posizione della matrice
            // che stavamo controllando
        }
        if(areValid(primarySequence.charAt(k), primarySequence.charAt(j)) &&
                nussinovJacobson[i][j] == val + nussinovJacobson[k+1][j-1]+1) {
            optimalSubstructure.addBond(new WeakBond( i: k+1,  j: j+1));
            traceback(i,  j: k-1);
            traceback( i: k+1,  j: j-1);
            return;
        }
    }
}
```

Parte 3: Valutare numericamente le prestazioni




Minimo
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
n
MIN NFF
Media
120000000
100000000
80000000
60000000
40000000
20000000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
n
MEDIA NFF
Massimo
300000000
250000000
200000000
150000000
100000000
50000000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
n
MAX NFF


Media+DevSt
160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
n
Media+DevSt
Media-DevSt
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
-10000000
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
n
Media-DevSt

Parte 4: Heap binario

La coda con priorità, necessaria per l'algoritmo di Dijkstra, è stata implementata con un heap binario. Di seguito le operazioni della coda di priorità implementate:

```java
public void insert(PriorityQueueElement element) {
    if(element == null) {
        throw new NullPointerException();        }
    heap.add(element);
    int i = heap.size()-1;
    PriorityQueueElement current = element;
    if(i <= 0) {// significa che c'e solo un l'emento
        return;}
    int fatherI = (i-1)/2;
    PriorityQueueElement father = heap.get(fatherI);
    while(i > 0 && current.getPriority() < father.getPriority()) {// confronta se il padre
        // e minore del figlio se si entra nel while
        heap.set(i, father); // modifica nella posizione del figlio ci mette il valore del padre
        father.setHandle(i);
        heap.set(fatherI, current);// nella posizione del padre ci va il valore del figlio
        current.setHandle(fatherI);
        i = fatherI; // i ora prende la posizione del padre perche deve controllare i nodi successivi
        if(i > 0) {
            fatherI = (i-1)/2;
            current = heap.get(i);// restituisce la posizione del elemrnt
            father = heap.get(fatherI);// restituisce la posizione del padre
        }
    }
}
```

```java
public void decreasePriority(PriorityQueueElement element,double newPriority) {

    if(!heap.contains(element)) {// se non e presente l'elemnto
        throw new NoSuchElementException();// lancia l'eccezione
    }
    if(newPriority >= element.getPriority()) {// la nuova priorita deve essere minore
        throw new IllegalArgumentException();// altrimenti lancia l'eccezione
    }

    element.setPriority(newPriority);// modifica la prioria vecchia con quella nuova
    int i = element.getHandle();//la posizione dell'elemento con nuova priorita
    int fatherI = (i-1)/2; // posizione del padre
    PriorityQueueElement father = heap.get(fatherI); // l'elemento che si trova nella posizione del padre
    while(i > 0 && element.getPriority() < father.getPriority()) {// fin quando la nuova priorita e minore
        heap.set(i, father);// modifica l'elemnto in posizione i con l'elemento del father
        father.setHandle(i);
        heap.set(fatherI, element);// modifica l'elemnto in posizione del father con element
        element.setHandle(fatherI);
        i = fatherI; // i  prende la posizione del padre
        if(i > 0) {
            fatherI = (i-1)/2;
            element = heap.get(i);// restituisce la posizione del elemrnt
            father = heap.get(fatherI);// restituisce la posizione del father
        }
    }
}
```

```java
public PriorityQueueElement extractMinimum() {
    if(heap.isEmpty()) {
        throw new NoSuchElementException();
    }
    int i = 0;
    PriorityQueueElement root = heap.get(i);// la radice che dobbiamo estrarre
    while(2*i+1 < heap.size()) {// se la posizione del figlio sinistro e minore della lunghezza del heap
        PriorityQueueElement left = heap.get(2*i+1);
        if(2*i+2 < heap.size()) {
            PriorityQueueElement right = heap.get(2*i+2);
            if(left.getPriority() < right.getPriority()) { // se il dodo di sinistra e piu piccolo di
                // quello di destra
                heap.set(i, left);// modifico la radice con il valore di left
                left.setHandle(i);
                i = 2*i+1; // modifico la i mettrndogli la posizione del figlio di sinistra
            } else {
                heap.set(i, right);// modifico la radice con il valore di right
                right.setHandle(i);
                i = 2*i+2;// modifico la i mettrndogli la posizione del figlio di destra
            }
        } else {// se non trovo niente a destra prendo quelo di sinistra
            heap.set(i, left);// modifico la radice con il valore di left
            left.setHandle(i);
            i = 2*i+1;// modifico la i mettrndogli la posizione del figlio di sinistra
        }

    }
    heap.remove(i);// rimuoviamo la radice
    return root;// ritorniamo la radice che avevamo posizionato sulla variabile d'appoggio
}
```

Parte 5: AdjacencyMatrixDirectedGraph

Per rappresentare il grafo diretto, è stata implementata una matrice di adiacenza: è una struttura dati costituita da una matrice quadrata che ha come indici di righe e colonne il numero dei vertici del grafo. Nel posto (i, j) della matrice si trova un oggetto GraphEdge<L> se e solo se esiste nel grafo un arco che va dal vertice i al vertice j, altrimenti vi si trova NULL. L'indice del vertice del grafo è determinato dall'ordine di inserimento e viene mappato al vertice nella HashMap<GraphNode<L>>, Integer> nodesIndex. Di seguito i metodi per aggiungere un vertice e un arco al grafo:

- addNode

```java
public boolean addNode(GraphNode<L> node) {
    // prima controlliamo tutte le possibile eccezioni
    if(node == null)// se il nodo e null lancia l'eccezione
        throw new NullPointerException();
    if(getNode(node) != null)// se il nodo e gia prensete return false
        return false;
    ArrayList<GraphEdge<L>> newRow = new ArrayList<>();
    for(int i = 0; i < matrix.size(); i++)
        newRow.add(null);// creiamo la riga e gli assenniamo tutti valori nulli
    this.matrix.add(newRow);// aggiungiamo una riga
    for(int i = 0; i < matrix.size(); i++) {// aggiungiamo una colonna
        this.matrix.get(i).add(null);
    }
    this.nodesIndex.put(node, matrix.size()-1); // aggiungiamo il nodo con il suo indice su nodesIndex
    return true;
}
```

- addEdge

```java
public boolean addEdge(GraphEdge<L> edge) {

    if(edge == null)
        throw new NullPointerException();

    GraphNode<L> node1 = edge.getNode1();
    GraphNode<L> node2 = edge.getNode2();
    //index1 e index2 indicano la posizione map nodesIndex
    Integer index1 = nodesIndex.get(node1);
    Integer index2 = nodesIndex.get(node2);

    if(index1 == null || index2 == null) {
        throw new IllegalArgumentException();
    }

    if(!edge.isDirected())
        throw new IllegalArgumentException();
    //trovo l'arco
    GraphEdge<L> foundEdge = matrix.get(index1).get(index2);
    if(foundEdge != null)
        return false;
    matrix.get(index1).set(index2, edge);
    return true;
}
```

Parte 6: Cammini Minimi con Sorgente Singola

E' stato implementato l'algoritmo di Dijkstra per effettuare il calcolo dei cammini minimi a partire da un vertice sorgente, con grafi diretti e con pesi non negativi. Di seguito l'implementazione, che fa utilizzo dell'Heap binario implementato come coda di priorità.

Un frammento di codice del metodo computeShortestPathsFrom.

```
// algoritmo di Dijkstra per i cammini minimi
while(!queue.isEmpty()) {
    GraphNode<L> el = (GraphNode<L>) queue.extractMinimum();// possiamo fare il cast perche abbiamo in
    //GraphNode
    for(GraphEdge<L> edge : graph.getEdgesOf(el)) {
        GraphNode<L> neighbour = edge.getNode2();
        if(queue.getBinaryHeap().contains(neighbour)) {
            double newDistance = el.getFloatingPointDistance() + edge.getWeight();
            if(newDistance < neighbour.getFloatingPointDistance()) {
                neighbour.setPrevious(el);
                queue.decreasePriority(neighbour, newDistance);
            }
        }
    }
}
```

Una volta calcolati tutti i cammini minimi, è possibile risalire, dato un certo vertice destinazione, al rispettivo cammino minimo a partire dalla sorgente.

```
public List<GraphEdge<L>> getShortestPathTo(GraphNode<L> targetNode) {
    if(targetNode == null)
        throw new NullPointerException();
    if(this.graph.getNode(targetNode) == null)
        throw new IllegalArgumentException();
    if(!this.isComputed)
        throw new IllegalStateException();

    GraphNode<L> currNode = targetNode;
    List<GraphEdge<L>> path = new ArrayList<>();
    // ricostruisco il percorso dal nodo target fino alla sorgente
    while(currNode != null && !currNode.equals(this.lastSource)) {
        if(currNode.getPrevious() != null) {
            GraphEdge<L> newEdge = this.graph.getEdge(currNode.getPrevious(), currNode);
            path.add(newEdge);
        }
        currNode = currNode.getPrevious();
    }
    if(currNode == null)
        return null;
    Collections.reverse(path);
    return path;
}
```