Progetto 2 Algoritmi e Strutture Dati

FRANCESCO MOSCHELLA, Università degli Studi di Camerino, Italia
TOUFIK MELLOUL, Università degli Studi di Camerino, Italia

In questo documento viene illustrato il lavoro svolto dagli studenti Moschella e Melloul per la realizzazione del progetto di Laboratorio di Algoritmi e Strutture Dati 2021.
Dopo una breve introduzione vengono descritti i punti salienti di ognuna delle classi principali; la sezione finale è dedicata ai test JUnit e i riferimenti ai sorgenti e alle basi teoriche utilizzate durante la fase di implementazione degli algoritmi e delle strutture dati illustrate.

1 INTRODUZIONE

In questo documento illustriamo il contenuto del progetto svolto per Laboratorio di Algoritmi e Strutture Dati.
In ordine troviamo i **Grafi** (Sezione 2), l'**Heap** (Sezione 3), gli **Algoritmi per il Calcolo del Cammino Minimo** (Sezione 4), una breve descrizione riguardante i **Test** JUnit (Sezione 6) e i **Riferimenti** al nostro codice e al materiale teorico usato per l'implementazione degli algoritmi e delle strutture dati presenti nel documento (Sezione 7).

2 GRAFI

I grafi sono strutture matematiche utilizzate per modellare relazioni tra coppie di oggetti. Un grafo è composto da Vertici e Archi. Esiste una distinzione tra Archi orientati e Archi non orientati: gli Orientati collegano 2 Vertici in modo asimmetrico, quelli Non-Orientati li collegano in modo simmetrico.

2.1 Adjacency Matrix Directed Graph

Un grafo Orientato è rappresentato dalla coppia $G = (V, E)$, dove V è un insieme di Vertici ed E è un insieme di Archi Orientati che collega coppie di Vertici.
La classe (Riferimento: 2) implementa un grafo Orientato mediante la matrice di adiacenza.
Nell'implementazione la matrice è rappresentata da una *ArrayList di ArrayList* contenente gli Archi del Grafo e accompagnata da una *Map* contenente i Vertici del Grafo e la relativa posizione nella matrice. L'utilizzo di *ArrayList* ha permesso di aggiungere e rimuovere in modo semplice e dinamico gli Archi.
L'uso di *ArrayList* in coppia con la *Map* ha, inoltre, consentito di effettuare: aggiunta, rimozione, ricerca e restituzione di Archi in tempo costante $O(1)$.

2.2 Map Adjacent List Undirected Graph

Un grafo Non-Orientato è rappresentato dalla coppia $G = (V, E)$, dove V è un insieme di Vertici ed E è un insieme di Archi Non-Orientati che collega coppie di Vertici.
La classe (Riferimento: 3) implementa un grafo Non-Orientato mediante la lista di adiacenza.
Per la rappresentazione del grafo si è usata la lista di adiacenza creata tramite l'utilizzo di una *Map* contenente i Nodi e un *Set* di Archi del grafo.
La classe permette aggiunta e ricerca di nodi in tempo costante $O(1)$, la rimozione di archi in tempo costante $O(1)$, mentre le restanti operazioni vengono fatte in tempo lineare rispetto alla quantità di nodi $O(|V|)$ o in tempo $O(|V| \cdot |E|)$.

3 HEAP

Gli *Heap* sono strutture ad albero e sono la convenzione per rappresentare le code di priorità.
Un *Heap* possiede tutti i vincoli di un albero con la stessa k-arietà, ai quali aggiunge le seguenti 2 proprietà:

(1) Proprietà di forma: un *Heap* è un albero completo con la stessa *arietà*, dove tutti i livelli, eventualmente ad esclusione dell'ultimo, sono completi e, se l'ultimo livello non è completo, i nodi sono disposti da sinistra a destra.
(2) Proprietà di Heap: la chiave di un nodo è maggiore o uguale ($\geq$) o minore o uguale ($\leq$) alla chiave di ognuno dei nodi figlio.

Gli *Heap* dove la chiave del nodo Padre è maggiore o uguale ($\geq$) a quella del nodo Figlio sono chiamati *Max-Heap*. Gli *Heap* dove la chiave del nodo Padre è minore o uguale ($\leq$) a quella del nodo Figlio sono chiamati *Min-Heap*

3.1 Binary Heap Min Priority Queue

L'*Heap* binario a priorità minore (*Min-Heap*) è stato usato nell'implementazione di una Coda a priorità Minima nel file (Riferimento: 4).
Nel progetto è rappresentato da un *ArrayList* che permette aggiunta e rimozione dinamica di nodi dall'*Heap*.
Assumendo che le operazioni sull'*ArrayList*, i controlli e le operazioni algebriche impieghino tempo costante ($O(1)$), l'inserimento di un nodo, l'estrazione del nodo minore e la diminuzione di priorità possono essere fatte in tempo logaritmico ($O(log_2(N))$).
Tutte le altre operazioni sono fatte in tempo costante ($O(1)$). I frammenti di codice successivi (Codici 3.1 e 3.1) mostrano il *cuore* del codice, rappresentato dalle operazioni di *Heapify-Up* e *Heapify-Down*, volte a mantenere le proprietà del *Min-Heap* sopra definite.

```
private void heapifyUp(int startPosition) {
    int parentIndex = getParentNode(startPosition);
    PriorityQueueElement parent = heap.get(parentIndex);
    PriorityQueueElement element = heap.get(startPosition);
    while(parent.getPriority()>element.getPriority()){
        heap.set(element.getHandle(), parent);
        heap.set(parent.getHandle(), element);

        parent.setHandle(element.getHandle());
        element.setHandle(parentIndex);

        parentIndex = getParentNode(parentIndex);
        if(parentIndex<0){
            //root reached
            break;
        }
        parent = heap.get(parentIndex);
    }
}

private void heapifyDown(int startPosition) {
```

```
        int childIndex = getBestChildIndex(startPosition);
        if(childIndex==-1){
            return;
        }
        PriorityQueueElement child = heap.get(childIndex);
        PriorityQueueElement element = heap.get(startPosition);
        while (child.getPriority()<element.getPriority()){
            heap.set(element.getHandle(), child);
            heap.set(child.getHandle(), element);

            child.setHandle(element.getHandle());
            element.setHandle(childIndex);

            childIndex = getBestChildIndex(element.getHandle());
            if(childIndex==-1){
                break;
            }
            child = heap.get(childIndex);
        }
}
```

Nei frammenti di codice precedenti vengono chiamate le funzioni *getParentNode(...)* e *getBestChildIndex(...)*. Tali funzioni sono volte ad ottenere rispettivamente l'indice del nodo Padre e l'indice del nodo Figlio con priorità minore.

4 ALGORITMI PER IL CAMMINO MINIMO

Il problema del *cammino minimo* consiste nel trovare il cammino tra 2 Vertici di un Grafo tale che la somma degli archi che costituiscono il cammino sia minima.
Un esempio di utilizzo del problema è rappresentato dal percorso ottimale da seguire per arrivare in un certo luogo avendo una mappa e conoscendo sia la propria posizione che quella del luogo di destinazione.
In questo esempio le strade possono essere modellate come Archi e gli incroci come Nodi del Grafo. Ogni arco può essere pesato considerando come peso il tempo necessario a percorrere la strada compresa tra i 2 incroci.
Esistono vari algoritmi per il calcolo del cammino minimo. Nelle sotto-sezioni successive verranno mostrati alcuni di essi.

4.1 Dijkstra Shortest Path Computer

Tra gli algoritmi per il calcolo del percorso minimo, l'algoritmo di Dijkstra è uno tra i più famosi. Appartiene alla *Famiglia* dei *Single-source*, cioè a quella famiglia di algoritmi che considera un solo nodo come sorgente.
Lo scopo dell'algoritmo di Dijkstra è trovare il cammino minimo in un Grafo ($G = (V, E)$) dove ogni Arco (E) è pesato e il peso è Non-Negativo.
La parte più importante della classe (Riferimento: 5) è la funzione *computeShortestPathsFrom(...)* mostrata nel frammento di codice seguente (4.1).

```
public void computeShortestPathsFrom(GraphNode<L> sourceNode) {
    if(sourceNode==null){
        throw new NullPointerException("Passed␣parameter␣must␣"
            +"be␣not␣null");
    }
    if(!graph.getNodes().contains(sourceNode)){
        throw new IllegalArgumentException("Invalid␣Node␣passed");
    }
    BinaryHeapMinPriorityQueue queue = new BinaryHeapMinPriorityQueue();
    for(GraphNode<L> node: graph.getNodes()){
        if(sourceNode.equals(node)){
            node.setFloatingPointDistance(0.0);
            queue.insert(node);
        }else{
            node.setFloatingPointDistance(Double.POSITIVE_INFINITY);
        }
    }
    while(!queue.isEmpty()){
        GraphNode<L> current = (GraphNode<L>) queue.extractMinimum();
        for(GraphEdge<L> edge: graph.getEdgesOf(current)){
            double distance = current.getFloatingPointDistance()+
                edge.getWeight();
            if(distance<edge.getNode2().getFloatingPointDistance()){
                edge.getNode2().setFloatingPointDistance(distance);
                edge.getNode2().setPrevious(current);
                queue.insert(edge.getNode2());
            }
        }
    }
    lastSource=sourceNode;
    solved=true;
}
```

Nel frammento di codice 4.1 viene utilizzata la coda di priorità descritta in precedenza (Sottosezione 3.1). La funzione implementa l'algoritmo di Dijkstra spiegato su Wikipedia (Riferimento: 13). Dato un grafo $G = (V, E)$, orientato e i cui Archi abbiano pesi non negativi, sia $N = |V|$ e $M = |E|$. Assumendo che i controlli e le operazioni aritmetiche siano svolte in tempo costante $O(1)$, che il grafo sia completo e senza *self-loop* ($M = N \cdot (N-1)$), la complessità temporale dell'algoritmo è:
$O(N + log_2 \cdot N + M(log_2 N + \frac{M}{N} \cdot log_2 N)) = O(N + log_2 N + M \cdot (log_2 N + (N-1) \cdot log_2 N)) =$
$O(N+log_2 N+M \cdot log_2 N+M \cdot (N-1) \cdot log_2 N) = O(N+log_2 N+N \cdot (N-1) \cdot log_2 N+N \cdot (N-1)^2 \cdot log_2 N) =$
$O(N + log_2 N \cdot (1 + N \cdot (N-1) + N \cdot (N-1)^2)) = O(N + log_2 N \cdot (N \cdot (N-1) \cdot (1 + (N-1)))) =$
$O(N + log_2 N \cdot (N \cdot (N-1) \cdot (N-1))) = O(N + N \cdot (N-1)^2 \cdot log_2 N) \approx O(N^2 log_2 N)$
Tuttavia, nel *average case scenario* (dove $|M| \ll |N|^2$), la complessità è di gran lunga minore, infatti:
$O(N + log_2 N + M \cdot (log_2 N + \frac{M}{N} \cdot log_2 N)) = O(N + log_2 N + M \cdot log_2 N + \frac{M^2}{N} \cdot log_2 N) =$

$O(N + log_2N \cdot (1 + M + \frac{M^2}{N}) = O(N + log_2N \cdot (M \cdot (1 + \frac{M}{N})) =$
$O(N + M \cdot log_2N + \frac{M}{N} \cdot log_2N) \approx O(N + M \cdot log_2N)$

4.2 Bellman Ford Shortest Path Computer

Anche l'algoritmo di Bellman-Ford è uno tra i più famosi per il calcolo del percorso minimo in un grafo e appartiene anch'esso alla *Famiglia* dei *Single-source*.
Lo scopo dell'algoritmo di Bellman-Ford è trovare il cammino minimo in un Grafo ($G = (V, E)$) dove ogni Arco (E) è pesato e il peso può essere Negativo, tuttavia nel grafo non devono essere presenti cicli negativi.
La parte più importante della classe (Riferimento: 6) è la funzione *computeShortestPathsFrom(...)* mostrata nel frammento di codice seguente (4.2).

```
public void computeShortestPathsFrom(GraphNode<L> sourceNode) {
    if(sourceNode==null){
        throw new NullPointerException("Passed␣parameter␣must␣" +
            "be␣not␣null");
    }
    if(!graph.getNodes().contains(sourceNode)){
        throw new IllegalArgumentException("Invalid␣Node␣passed");
    }
    for(GraphNode<L> node: graph.getNodes()){
        node.setFloatingPointDistance(Double.POSITIVE_INFINITY);
        node.setPrevious(null);
        if(sourceNode.equals(node)){
            node.setFloatingPointDistance(0.0);
        }
    }
    for(int i=0;i<graph.getNodes().size();++i){
        for(GraphEdge<L> edge:graph.getEdges()){
            GraphNode<L> source = edge.getNode1();
            GraphNode<L> dest = edge.getNode2();
            if(dest.getFloatingPointDistance()>source.
                getFloatingPointDistance()+edge.getWeight()){
                dest.setFloatingPointDistance(source.
                    getFloatingPointDistance()+edge.getWeight());
                dest.setPrevious(source);
            }
        }
    }
    for(GraphEdge<L> edge: graph.getEdges()){
        GraphNode<L> source = edge.getNode1();
        GraphNode<L> dest = edge.getNode2();
        if(dest.getFloatingPointDistance()>source.
                getFloatingPointDistance()+edge.getWeight()){
```

```
                throw new IllegalStateException("Graph contains "+
                    "negative-weight cycle");
            }
        }

        lastSource=sourceNode;
        solved=true;
    }
```

La funzione implementa l'algoritmo di Bellman-Ford spiegato su Wikipedia (Riferimento: 14). Dato un grafo $G = (V, E)$, orientato e in cui non esistano cicli negativi, sia $N = |V|$ e $M = |E|$. Assumendo che i controlli e le operazioni aritmetiche siano svolte in tempo costante $O(1)$, la complessità temporale dell'algoritmo è:
$O(N + N \cdot M + M) \approx O(N \cdot M)$

4.3 Floyd Warshall All Pairs Shortest Path Computer

L'algoritmo di Floyd-Warshall calcola il cammino minimo per tutte le coppie di un grafo pesato e orientato.
L'idea che sta alla base di questo algoritmo è un processo iterativo per cui, scorrendo tutti i nodi, ad ogni passo h si ha (data una matrice A), nella posizione [i,j], la distanza - pesata - minima dal nodo di indice i a quello j, attraversando solo nodi di indice minore o uguale a h. Se non vi è alcun collegamento allora nella cella troviamo infinito.
Ovviamente alla fine (con h = numero di nodi), leggendo la matrice, si ricava la distanza minima fra i vari nodi del grafo.
La parte più importante della classe (Riferimento: 7) è la funzione *computeShortestPaths(...)* mostrata nel frammento di codice seguente (4.3).

```
public void computeShortestPaths() {
    int totalepeso = 0;
    for (GraphEdge<L> g : graph.getEdges()) {
        totalepeso += g.getWeight();
    }
    if (totalepeso < 0)
        throw new IllegalArgumentException();
    for (int k = 1; k < costMatrix.length; k++) {
        for (int i = 1; i < costMatrix.length; i++) {
            for (int j = 1; j < costMatrix.length; j++) {
                if (costMatrix[i][j] > costMatrix[i][k] +
                    costMatrix[k][j]) {
                    costMatrix[i][j] = costMatrix[i][k] +
                        costMatrix[k][j];
                    predecessorMatrix[i][j] = predecessorMatrix[k][j];
                }
            }
        }
```

```
        }
        computato = true;
    }
```

È un algoritmo di programmazione dinamica con complessità temporale $O(|V|^3)$

5 ALGORITMI PER L'ALBERO DI COPERTURA MINIMO

Un albero di copertura di un grafo è un albero che contiene

- tutti i vertici del grafo
- un sottoinsieme degli archi del grafo: quelli necessari per connettere tra loro tutti i vertici con uno e un solo cammino

Infatti ciò che differenzia un grafo da un albero è che in quest'ultimo non sono presenti cammini multipli tra due nodi.

5.1 Kruskal MSP

Kruskal è uno degli algoritmi utilizzati per individuare l'albero di copertura minimo di un grafo. In particolare quest'ultimo ordina gli archi secondo costi crescenti e costruisce un insieme ottimo di archi T scegliendo di volta in volta un arco di peso minimo che non forma cicli con gli archi già scelti.
La parte più importante della classe (Riferimento: 8) è la funzione *cmputeMSP(...)* mostrata nel frammento di codice seguente (5.1).

```
public Set<GraphEdge<L>> computeMSP(Graph<L> g) {
    if (g == null)
        throw new NullPointerException(
                "Calcolo dell'albero minimo di copertura su un" +
                    "grafo nullo");
    // controllo le condizioni sul grafo
    checkGraph(g);
    // creo l'insieme risultato
    Set<GraphEdge<L>> risultato = new HashSet<>();
    // creo gli insiemi disgiunti, uno per ogni nodo
    this.disjointSets.clear();
    for (GraphNode<L> n : g.getNodes()) {
        HashSet<GraphNode<L>> s = new HashSet<>();
        s.add(n);
        this.disjointSets.add(s);
    }
    // Ordino gli archi in senso crescente in una lista in modo da
    // evitare problemi con il comparator che e' incompatibile
    // con equals.
    List<GraphEdge<L>> archi = new ArrayList<>(g.getEdges());
    archi.sort(this.edgesComparator);
    for (GraphEdge<L> e : archi) {
        int i = setOf(e.getNode1());
```

```
            int j = setOf(e.getNode2());
            if (i != j) {
                risultato.add(e);
                union(i, j);
            }
        }
        return risultato;
    }
```

Il costo computazionale dell'algoritmo è nel caso peggiore $O(V \cdot E)$ dove E è il numero di archi ed V il numero di vertici.

5.2 Prim MSP

Prim è essenzialmente un algoritmo di visita che, partendo da un nodo iniziale u (scelto arbitrariamente), esamina tutti i nodi del grafo.
Ad ogni iterazione, partendo da un nodo v, visita un nuovo nodo w (scelto secondo opportuni criteri) e pone l'arco (v, w) nell'insieme T che, al termine dell'esecuzione, conterrà una soluzione ottima.
La differenza sostanziale rispetto ad un algoritmo di visita "standard" è che la scelta del prossimo nodo da visitare viene fatta introducendo un concetto di priorità tra nodi e che l'insieme Q dei nodi da visitare viene gestito come una coda di priorità.
La parte più importante della classe (Riferimento: 9) è la funzione *computeMSP(...)* mostrata nel frammento di codice seguente (5.2).

```
public void computeMSP(Graph<L> g, GraphNode<L> s) {
    if (g == null || s == null)
        throw new NullPointerException();
    if (!g.containsNode(s))
        throw new IllegalArgumentException();
    if (g.isDirected())
        throw new IllegalArgumentException(
                "Tentativo di applicare l'algoritmo di Prim " +
                "su un grafo orientato");
    // Determina se tutti gli archi del grafo hanno un peso
    // assegnato e, se si', positivo o nullo
    Set<GraphEdge<L>> edges = g.getEdges();
    for (GraphEdge<L> e : edges) {
        if (!e.hasWeight())
            throw new IllegalArgumentException(
                "Tentativo di applicare l'algoritmo di Prim su " +
                "un grafo con almeno un arco con peso " +
                "non specificato");
        if (e.getWeight() < 0)
            throw new IllegalArgumentException(
                "Tentativo di applicare l'algoritmo di Prim " +
```

```
                "su un grafo con almeno un arco con peso negativo");
    }

    boolean vAppartieneQ;

    GraphEdge<L> arcoUV = null;

    for (GraphNode<L> v : g.getNodes()) {
        v.setPriority(Double.MAX_VALUE);
        v.setPrevious(null);
    }
    s.setPriority(0);
    for (GraphNode<L> nodo : g.getNodes()) {
        this.queue.insert(nodo);
    }
    GraphNode<L> u;
    while (this.queue.size() != 0) {
                u = (GraphNode<L>) this.queue.extractMinimum();
        for (GraphNode<L> v : g.getAdjacentNodesOf(u)) {
            vAppartieneQ = quequeContains(v);

            for (GraphEdge<L> arco : g.getEdgesOf(u)) {
                //poiche' il grafo non e' orientato bisogna controllare
                //ambo i nodi dell'arco per vedere se esso e' l'arco
                //che collega il nodo u con il nodo v
                if (arco.getNode1().equals(v) || arco.getNode()
                        .equals(v)) {
                    arcoUV = arco;
                    break;
                }
            }
            if (vAppartieneQ && arcoUV != null && arcoUV.getWeight()
                    < v.getPriority()) {
                v.setPrevious(u);
                v.setPriority(arcoUV.getWeight());
            }
        }
    }
}
```

La complessità dell'algoritmo di Prim dipende dall'implementazione della struttura dati ausiliaria utilizzata per contenere i nodi.
Se la struttura dati usiliaria è implementata con una coda di priorità e si assume che il test di presenza o meno nella coda abbia una complessità costante, allora il tempo totale di esecuzione dell'algoritmo sarà di $O(E \cdot logV)$.

Se la coda con priorità è realizzata con Heap di Fibonacci il tempo di esecuzione può essere ulteriormente migliorato. L'implementazione dell'algoritmo di Prim con Fibonacci Heap è la più efficiente ottenibile, infatti il costo di esecuzione è $O(E + V \cdot logV)$.

6 TEST

Per verificare la correttezza delle classi create sono stati implementati dei test JUnit.
JUnit è un framework per *unit testing*, cioè permette di effettuare test su singole sezioni di codice (Riferimento: 15).
Per ognuna delle classi sopra descritte è stato implementato un apposito file per JUnit Testing, in modo da verificare il corretto funzionamento di ognuno dei metodi. I test creati sono reperibili nella cartella del progetto, contenuta nella Repository Github (Riferimento: 1), e sono facilmente riconoscibili poiché ogni file JUnit è stato nominato con il formato *<Nome della Classe da Verificare>Test.java*.

7 RIFERIMENTI

(1) Repository Github del Progetto, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati, Accessed 2021-02-18
(2) Classe AdjacencyMatrixDirectedGraph.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/AdjacencyMatrixDirectedGraph.java, Accessed 2021-02-18
(3) Classe MapAdjacentListUndirectedGraph.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/MapAdjacentListUndirectedGraph.java, Accessed 2021-02-18
(4) Classe BinaryHeapMinPriorityQueue.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/BinaryHeapMinPriorityQueue.java, Accessed 2021-02-18
(5) Classe DijkstraShortestPathComputer.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/DijkstraShortestPathComputer.java, Accessed 2021-02-18
(6) Classe BellmanFordShortestPathComputer.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/BellmanFordShortestPathComputer.java, Accessed 2021-02-18
(7) Classe FloydWarshallAllPairsShortestPathComputer.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/FloydWarshallAllPairsShortestPathComputer.java, Accessed 2021-02-18
(8) Classe KruskalMSP.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/KruskalMSP.java, Accessed 2021-02-18
(9) Classe PrimMSP.java, [Online]. Available: https://github.com/HarlockOfficial/ST0853_Progetto_Algoritmi_E_Strutture_Dati/blob/main/src/it/unicam/cs/asdl2021/totalproject2/PrimMSP.java, Accessed 2021-02-18
(10) Pagina Wikipedia riguardante i grafi, [Online]. Available: https://en.wikipedia.org/wiki/Graph_theory, Accessed 2021-02-18
(11) Pagina Wikipedia riguardante gli Heap, [Online]. Available: https://en.wikipedia.org/wiki/Binary_heap, Accessed 2021-02-18

(12) Pagina Wikipedia riguardante il problema del cammino minimo, [Online]. Available: https://en.wikipedia.org/wiki/Shortest_path_problem, Accessed 2021-02-18
(13) Pagina Wikipedia riguardante l'algoritmo di Dijkstra, [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra's_algorithm, Accessed 2021-02-18
(14) Pagina Wikipedia riguardante l'algoritmo di Bellman-Ford, [Online]. Available: https://en.wikipedia.org/wiki/Bellman-Ford_algorithm, Accessed 2021-02-18
(15) Pagina Wikipedia riguardante JUnit, [Online]. Available: https://en.wikipedia.org/wiki/JUnit, Accessed 2021-02-18