

A dark blue vertical bar runs along the left edge of the page. A blue arrow-shaped banner points to the right from this bar, containing the date. In the bottom-left corner, there are several thin, curved lines in dark blue and light grey.

01/01/2021

PROGETTO TOTALE 2

Algoritmi e strutture dati

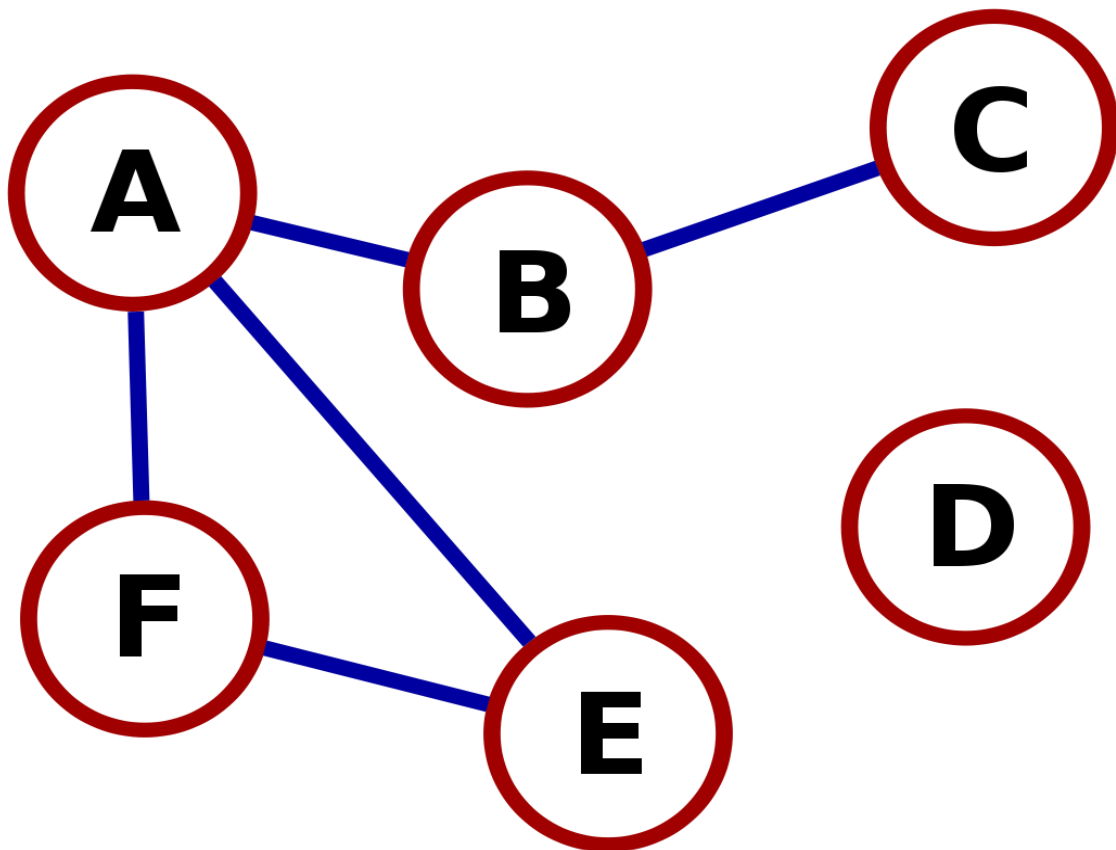
DAVIDE NAPPO 110157
UNICAM

Obbiettivi:

- Realizzare in Java una implementazione di un grafo generico non orientato con liste di adiacenza e una implementazione di un grafo generico orientato con matrice di adiacenza.
- Realizzare in Java un'implementazione di una coda con priorità implementata tramite uno heap binario.
- 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 precedente e realizzare in Java una implementazione dell'algoritmo di Bellman-Ford.
- Per il problema dei cammini minimi tra tutte le coppie di nodi: realizzare in Java una implementazione dell'algoritmo di Floyd-Warshall.
- Per il problema del calcolo di un albero minimo di copertura: realizzare in Java una implementazione dell'algoritmo di Kruskal e realizzare in Java una implementazione dell'algoritmo di Prim.
- Realizzare una suite di test JUnit 5 che controlli le funzionalità implementate.

GRAFI

Si dice grafo una coppia ordinata $G=(V,E)$ di insiemi, con V insieme dei nodi ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di V . 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.



Rappresentazione di un grafo

GRAFO GENERICO NON ORIENTATO

Per la rappresentazione di un grafo non orientato abbiamo usato le liste di adiacenza. L'idea della rappresentazione è per ogni nodo u , la lista di adiacenza `adjacentLists[u]` contiene tutti i vertici v tali che esista un arco (u,v) appartenente al grafo.

```
49•  /*
50   * Le liste di adiacenza sono rappresentate con una mappa. Ogni nodo viene
51   * associato con l'insieme degli archi collegati. Nel caso in cui un nodo
52   * non abbia archi collegati Ã associato con un insieme vuoto. La variabile
53   * istanza Ã protected solo per scopi di test JUnit.
54   */
55   protected final Map<GraphNode<L>, Set<GraphEdge<L>>> adjacentLists;
56
```

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.

```
168
167•  @Override
168  public Set<GraphNode<L>> getAdjacentNodesOf(GraphNode<L> node) {
169      // TODO implementare
170      if(node==null) {
171          throw new NullPointerException();
172      }
173      if(!this.adjacentLists.containsKey(node)) {
174          throw new IllegalArgumentException("Il nodo passato non esiste");
175      }
176      Set<GraphNode<L>> nodeSet = new HashSet<GraphNode<L>>();
177      for(Set<GraphEdge<L>> objSet : this.adjacentLists.values()) {
178          for(GraphEdge<L> obj : objSet) {
179              if(obj.getNode2().equals(node)) nodeSet.add(obj.getNode1());
180              if(obj.getNode1().equals(node)) nodeSet.add(obj.getNode2());
181          }
182      }
183      return nodeSet;
184  }
185
186
```

Metodo `getAdjacentNodesOf`

GRAFO GENERICO ORIENTATO

Per la rappresentazione di un grafo generico orientato abbiamo usato una 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.

```
42
43 // Insieme dei nodi e associazione di ogni nodo con il proprio indice nella
44 // matrice di adiacenza
45 protected Map<GraphNode<L>, Integer> nodesIndex;
46
47 // Matrice di adiacenza, gli elementi sono null o oggetti della classe
48 // GraphEdge<L>. L'uso di ArrayList permette alla matrice di aumentare di
49 // dimensione gradualmente ad ogni inserimento di un nuovo nodo.
50 protected ArrayList<ArrayList<GraphEdge<L>>> matrix;
51
```

Insieme dei nodi e matrice di adiacenza

```
123
124• @Override
125 public boolean containsNode(GraphNode<L> node) {
126
127     if (node == null) {
128         throw new NullPointerException("Il nodo passato è nullo");
129     }
130     return nodesIndex.containsKey(node);
131 }
132
133
134• @Override
135 public GraphNode<L> getNodeOf(L label) {
136
137     if (label == null) {
138         throw new NullPointerException();
139     }
140     for (GraphNode<L> n : nodesIndex.keySet()) {
141         if (n.getLabel().equals(label)) {
142             return n;
143         }
144     }
145     return null;
146 }
147
148• @Override
```

Metodo containsNode e metodo getNodeOf

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:

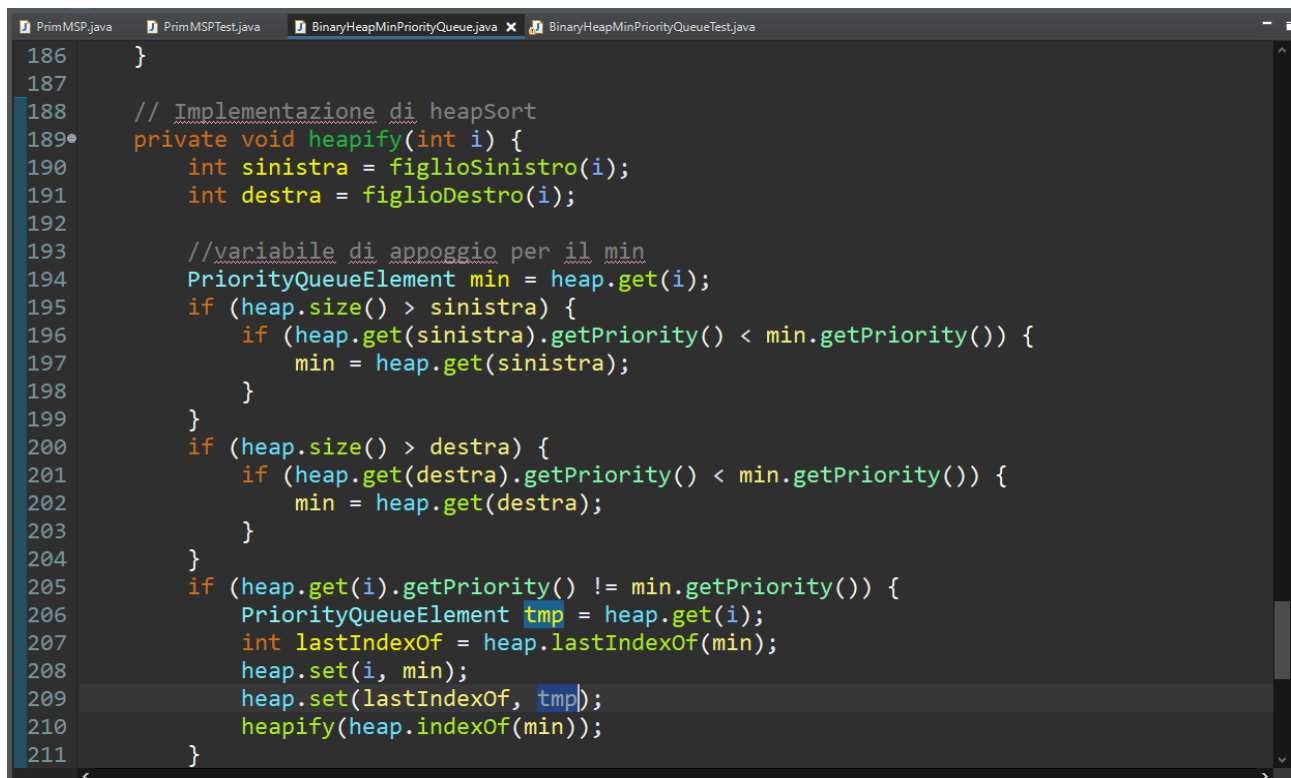
(1) Proprietà di forma: un Heap è un albero completo con la stessa arietà, dove tutti i livelli, eventualmente ad esclusione dell'ultimo, sono completi e, se l'ultimo livello non è completo, i nodi sono disposti da sinistra a destra.

(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.

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. Nel progetto è rappresentato da un ArrayList che permette aggiunta e rimozione dinamica di nodi dall'Heap. Una coda di priorità è una struttura dati che serve a mantenere in ordine un insieme S di elementi, ciascuno con un valore associato detto chiave.



```
186     }
187
188     // Implementazione di heapSort
189     private void heapify(int i) {
190         int sinistra = figlioSinistro(i);
191         int destra = figlioDestro(i);
192
193         //variabile di appoggio per il min
194         PriorityQueueElement min = heap.get(i);
195         if (heap.size() > sinistra) {
196             if (heap.get(sinistra).getPriority() < min.getPriority()) {
197                 min = heap.get(sinistra);
198             }
199         }
200         if (heap.size() > destra) {
201             if (heap.get(destra).getPriority() < min.getPriority()) {
202                 min = heap.get(destra);
203             }
204         }
205         if (heap.get(i).getPriority() != min.getPriority()) {
206             PriorityQueueElement tmp = heap.get(i);
207             int lastIndexOf = heap.lastIndexOf(min);
208             heap.set(i, min);
209             heap.set(lastIndexOf, tmp);
210             heapify(heap.indexOf(min));
211         }
212     }
```

Metodo Heapify

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. Il problema della ricerca del cammino minimo può essere definito sia su grafi orientati che su grafi non orientati. Di questo problema esistono alcune varianti in cui, partendo da un dato vertice, può essere richiesto di trovare i cammini minimi verso tutti gli altri vertici; oppure trovare i cammini minimi per ogni coppia di vertici del grafo. Un problema simile è quello del commesso viaggiatore, in cui si cerca il percorso più breve che attraversi tutti i nodi del grafo una sola volta, per poi ritornare al punto di partenza. Questo problema è però NP-Completo, per cui una soluzione efficiente potrebbe non esistere. Nel campo delle telecomunicazioni, questo problema viene a volte chiamato min-delay path problem. Nelle sotto-sezioni successive verranno mostrati alcuni di essi.

Algoritmo di Dijkstra

L'algoritmo di Dijkstra risolve il problema dei cammini minimi da sorgente unica in un grafo orientato pesato, nel caso in cui tutti i pesi degli archi non siano negativi. L'algoritmo mantiene un insieme S di nodi i cui pesi finali dei cammini minimi dalla sorgente s sono stati già determinati. L'algoritmo seleziona ripetutamente il nodo u appartenente all'insieme dei nodi del grafo $- S$ con la stima minima del cammino minimo, aggiunge u a S e rilassa tutti gli archi che escono da u . Manteniamo anche una coda di min-priorità Q di nodi, utilizzando come valore per le loro chiavi la `floatingDistance`. 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.

```
DijkstraShortestPathComputer.java x DijkstraShortestPathComputerTest.java
94
95 @Override
96 public void computeShortestPathsFrom(GraphNode<L> sourceNode) {
97
98     if(sourceNode == null){
99         throw new NullPointerException("Il nodo passato è nullo");
100     }
101     if(!this.grafo.containsNode(sourceNode)){
102         throw new IllegalArgumentException("Il nodo passato non è contenuto nel grafo");
103     }
104     //inizializzazione
105     BinaryHeapMinPriorityQueue queue = new BinaryHeapMinPriorityQueue();
106     for (GraphNode<L> node : grafo.getNodes()) {
107         if (sourceNode.equals(node)) {
108             node.setFloatingPointDistance(0.0);
109             queue.insert(node);
110         } else {
111             //distanza iniziale sconosciuta
112             //Imposto tutte le distanze dei nodi con un valore che non potrà mai assum
113             node.setFloatingPointDistance(Double.POSITIVE_INFINITY);
114         }
115     }
116     while (!queue.isEmpty()) {
117
118         @SuppressWarnings("unchecked")
119         GraphNode<L> nodeCurrent = (GraphNode<L>) queue.extractMinimum();
```

```
116     while (!queue.isEmpty()) {
117
118         @SuppressWarnings("unchecked")
119         GraphNode<L> nodeCurrent = (GraphNode<L>) queue.extractMinimum();
120         for (GraphEdge<L> edge : grafo.getEdgesOf(nodeCurrent)) {
121             double distance = nodeCurrent.getFloatingPointDistance() + edge.getWeight();
122             //relax
123             if (distance < edge.getNode2().getFloatingPointDistance()) {
124                 edge.getNode2().setFloatingPointDistance(distance);
125                 edge.getNode2().setPrevious(nodeCurrent);
126                 queue.insert(edge.getNode2());
127             }
128         }
129     }
130     this.source = sourceNode;
131     this.isComputed = true;
132
133 }
134 /**
```

ComputeShortestPathsFrom

Algoritmo di Bellman-Ford

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. L'algoritmo restituisce un valore booleano che indica se esiste oppure no un ciclo di peso negativo che è raggiungibile dalla sorgente. Se un tale ciclo non esiste, l'algoritmo fornisce i cammini minimi e i loro pesi.

```
cs/asdi2021/totalproject2/BellmanFordShortestPathComputer.java - Eclipse IDE
Run Window Help
BellmanFordShortestPathComputer.java x BellmanFordShortestPathComputerTest.java DijkstraShortestPathComputer.java
89 @Override
90 public void computeShortestPathsFrom(GraphNode<L> sourceNode) {
91
92     if (sourceNode == null) {
93         throw new NullPointerException();
94     }
95
96     if (!grafo.containsNode(sourceNode)) {
97         throw new IllegalArgumentException();
98     }
99
100     initSingleSource(sourceNode);
101     //processo gli archi ripetutamente
102     for (int i = 0; i < grafo.getNodes().size(); ++i) {
103         for (GraphEdge<L> edge : grafo.getEdges()) {
104             GraphNode<L> source = edge.getNode1();
105             GraphNode<L> dest = edge.getNode2();
106             //relax
107             if (dest.getFloatingPointDistance() > source.getFloatingPointDistance() + edge.getWeight()) {
108                 dest.setFloatingPointDistance(source.getFloatingPointDistance() + edge.getWeight());
109                 dest.setPrevious(source);
110             }
111         }
112     }
113     // Mi accerto che non ci siano cicli negativi
114     for (GraphEdge<L> e : grafo.getEdges()) {
115         if (e.getNode2().getFloatingPointDistance() > e.getNode1().getFloatingPointDistance() + e.getWeight()) {
116             throw new IllegalStateException(" ciclo negativo");
117         }
118     }
119     /* Inizializzo le informazioni necessarie associate ai nodi del grafo
120     associato a questo calcolatore*/
121     this.lastSource = sourceNode;
122     this.isComputed = true;
123 }
124
125
126
```

computeShortestPathsFrom

Algoritmo di Floyd-Warshall

Risolve il problema dei cammini minimi tra tutte le coppie di nodi. Possono essere presenti archi di peso negativo, ma si suppone che non ci siano loop negativi. L'algoritmo considera i nodi intermedi di un cammino minimo, dove un nodo intermedio di un cammino semplice $p = \langle v_1, \dots, v_l \rangle$ è un nodo qualsiasi di p diverso da v_1 e v_l . 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.

matrice C					matrice P			
1	2	3	4		1	2	3	4
∞	∞	∞	∞	1	1	2	3	4
∞	∞	∞	∞	2	1	2	3	4
∞	∞	∞	∞	3	1	2	3	4
∞	∞	∞	∞	4	1	2	3	4

nodo di partenza (da)

nodo di arrivo (a)

Esempio di matrice dei costi e matrice dei predecessori dei nodi

```
23
24• /*
25 * Matrice dei costi dei cammini minimi. L'elemento in posizione i,j
26 * corrisponde al costo di un cammino minimo tra il nodo i e il nodo j, dove
27 * i e j sono gli interi associati ai nodi nel grafo (si richiede quindi che
28 * la classe che implementa il grafo supporti le operazioni con indici).
29 */
30 private double[][] costMatrix;
31
32• /*
33 * Matrice dei predecessori. L'elemento in posizione i,j Ã -1 se non esiste
34 * nessun cammino tra i e j oppure corrisponde all'indice di un nodo che
35 * precede il nodo j in un qualche cammino minimo da i a j. Si intende che i
36 * e j sono gli indici associati ai nodi nel grafo (si richiede quindi che
37 * la classe che implementa il grafo supporti le operazioni con indici).
38 */
39 private int[][] predecessorMatrix;
40
```

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.

Algoritmo di Kruskal

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.

```
52 public Set<GraphEdge<L>> computeMSP(Graph<L> g) {
53     // TODO implementare
54     // In caso di parametro null
55     if (g == null) {
56         throw new NullPointerException("Parametro null!");
57     }
58     // Se il grafo è orientato
59     if (g.isDirected()) {
60         throw new IllegalArgumentException("Il grafo è orientato");
61     }
62     // Se il grafo ha archi non pesati o negativi
63     for (GraphEdge<L> edge : g.getEdges()) {
64         if ((!edge.hasWeight()) || (edge.getWeight() < 0)) {
65             throw new IllegalArgumentException("Arco non pesato o negativo!");
66         }
67     }
68     ArrayList<GraphEdge<L>> archiOrdinati = sortEdges(g.getEdges());
69     Set<GraphEdge<L>> archiIdonei = new HashSet<>();
70
71     //creo gli insiemi disgiunti, uno per ogni nodo
72     for (GraphNode<L> node : g.getNodes()) {
73         HashSet<GraphNode<L>> lista = new HashSet<>();
74         lista.add(node);
75         this.disjointSets.add(lista);
76     }
77     //per ogni arco
78     for (GraphEdge<L> arco : archiOrdinati) {
79         //se i vertici non appartengono allo stesso insieme (quindi evito i cicli)
80         if (findSet(arco.getNode1()) != findSet(arco.getNode2())) {
81             // L'arco viene aggiunto all'insieme degli archi che effettuano il Kruskal
82             archiIdonei.add(arco);
83             //unisco i due insiemi
84             unionSet(arco.getNode1(), arco.getNode2());
85         }
86     }
87     return archiIdonei;
88 }
89
90
91 // TODO implementare: inserire eventuali metodi privati per fini di
92 // implementazione
93
```

ComputeMSP

Algoritmo di Prim

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. Questa strategia è golosa perché l'albero cresce includendo a ogni passo un arco che contribuisce con la quantità più piccola possibile a formare il peso dell'albero.

```
58 public void computeMSP(Graph<L> g, GraphNode<L> s) {
59     |
60     //Controllo se gli elementi passati siano nulli
61     if(g == null || s == null){
62         throw new NullPointerException("Uno dei due valori passati è nullo");
63     }
64     //Controllo che il nodo sorgente passato sia contenuto nel grafo
65     if(!g.containsNode(s)){
66         throw new IllegalArgumentException("Il nodo passato non è nel grafo");
67     }
68     //Controllo se i dati degli elementi siano idonei
69     if(g.isDirected() || !hasWeight(g)){
70         throw new IllegalArgumentException("Il grafo ha archi con pesi non idonei oppure è orientato");
71     }
72     //Ciclo tutti i nodi del grafo
73     for(GraphNode<L> nodo : g.getNodes()){
74         //Imposto tutte le priorità dei nodi ad infinito
75         nodo.setPriority(Double.POSITIVE_INFINITY);
76         //Imposto la priority del nodo sorgente con un valore più basso in modo che sarà il primo ad
77         //essere preso durante l'estrazione dallo heap
78         if(nodo.equals(s)){
79             nodo.setPriority(0);
80         }
81         //Setto tutti i padri dei nodi a null
82         nodo.setPrevious(null);
83         //Inserisco i nodi all'interno dello heap
```

```

83         //Inserisco i nodi all'interno dello heap
84         this.queue.insert(nodo);
85     }
86     //Scorro lo heap fino a che non è vuoto
87     while(this.queue.size() != 0){
88         //Creo una variabile di appoggio
89         GraphNode<L> appoggio = null;
90         //Ciclo tutti i nodi del grafo
91         int handle = this.queue.extractMinimum().getHandle();
92         for(GraphNode<L> nodo : g.getNodes()) {
93             //Cerco il nodo corrispondente all'handle del valore minimo dello heap
94             if (nodo.getHandle() == handle) {
95                 appoggio = nodo;
96                 break;
97             }
98         }
99         //Scorro tutti i nodi adiacenti al nodo precedentemente estratto dallo heap
100        for(GraphNode<L> adj : g.getAdjacentNodesOf(appoggio)){
101            //Controllo se lo heap contiene questo nuovo nodo
102            if(!this.queue.contains(adj)) {
103                //In caso affermativo ciclo tutti gli archi uscenti del nodo estratto dallo heap
104                for (GraphEdge<L> edge : g.getEdgesOf(appoggio)) {
105                    //Controllo quale arco collega i due nodi interessati
106                    if(correctEdge(edge, appoggio, adj)){
107                        //Controllo se il nuovo nodo ha una priority più alta del peso dell'arco o ar
108                        // è stato visitato
109                        if(adj.getPriority() > edge.getWeight()){
110                            //In caso affermativo imposto il padre del nodo nuovo

```

src/it/unicam/cs/asdl2021/totalproject2/PrimMSP.java - Eclipse IDE

```

107                //Controllo se il nuovo nodo ha una priority più alta del peso dell'arco o ar
108                // è stato visitato
109                if(adj.getPriority() > edge.getWeight()){
110                    //In caso affermativo imposto il padre del nodo nuovo
111                    // con il nodo estratto dallo heap
112                    adj.setPrevious(appoggio);
113                    //Imposto la nuova priority del nodo scoperto con il peso dell'arco
114                    adj.setPriority(edge.getWeight());
115                }
116            }
117        }
118    }
119 }
120 }
121 }
122

```

computeMSP

