Damiano Pasquini

Docente Luca Tesei

Algoritmi e Strutture Dati Laboratorio (6 CFU)

Marzo 2022

Relazione Progetto Totale n. 2

GRAFO GENERICO NON ORIENTATO CON LISTE DI ADIACENZA E GRAFO GENERICO ORIENTATO CON MATRICE DI ADIACENZA

Grafo non orientato con liste di adiacenza. La struttura dati per rappresentare il grafo tramite liste di adiacenza si basa sulla struttura dati Map compresa nella libreria java.util. A differenza della rappresentazione standard, tale rappresentazione risulta più efficiente tramite l'utilizzo delle tabelle hash.

Per *aggiungere* e *eliminare* nodi in questo grafo sono stati utilizzati relativamente i metodi .put(key,value), .remove(key, value). Per la *ricerca* tramite l'oggetto è stato utilizzato il metodo .containsKey(key) mentre per la ricerca tramite l'attributo *label* del nodo vengono analizzati con un for le *label* dei nodi contenuti in adjacentLists.keySet() per poi restituire il nodo cercato.

Per *aggiungere* ed *eliminare* archi al grafo sono stati utilizzati relativamente i metodi .remove(Object) e .add(Object) agendo sui set di archi associati ad ogni nodo, come da struttura della lista di adiacenza. La *ricerca* di archi nel grafo è stata effettuata tramite il metodo .keySet() della lista di adiacenza, ricercando l'arco tra tutti i set di archi associati ad ogni nodo.

Grafo orientato con matrice di adiacenza. La struttura dati per rappresentare tale grafo è basata sull'utilizzo di due attributi; una mappa (Map contenuta in java.util) contenente i nodi (Key) con i relativi indici (Value), ed un arrayList di arrayList di GraphEdge<L>, ovvero una matrice quadrata contenente gli archi del grafo.

Per *aggiungere* nodi in questo grafo si agisce sulla Map *nodesIndex* aggiungendo il nodo con il metodo .put(key, value) con indice (value) assegnato equivalente al numero di nodi attuali dato da .nodeCount, per poi modificare la matrice *matrix* aggiungendo un arrayList con valori *null*. La *ricerca* di nodi invece è effettuata: tramite il .containsKey(key) se si ricerca con l'oggetto nodo, tramite *nodesIndex*.get(key) se si ricerca con la *label* del nodo, infine se si ricerca con l'indice si utilizza un forEach scorrendo *nodesIndex*.entrySet(), controllando node.getValue() per ogni nodo e in caso di corrispondenza viene restituito in output il nodo tramite node.getKey().

Per *aggiungere* un arco in questo grafo si ricavano gli indici dei nodi tramite il metodo .getLabel(), per poi aggiungere con il metodo .add(index, value) l'arco in input. Per *rimuovere* archi si opera sempre sulla matrice *matrix* però attraverso i metodi .get(index) e .set(index, value) passando su "index" gli indici dei due nodi corrispondenti all'arco, e su "value" *null*, per indicare la rimozione. Infine per la *ricerca* (containsEdge(edge)) di un arco in questa tipologia di grafo è utilizzata la stessa metodologia della rimozione e aggiunta, quindi attraverso il metodo .get(index) richiamato due volte sulla matrice di adiacenza *matrix* indicando gli indici dei due nodi associati all'arco.

CODA CON PRIORITA' IMPLEMENTATA TRAMITE UNO HEAP BINARIO

Questa implementazione della **coda con priorità tramite uno heap binario** è di tipo *Min Priority Queue* poiché nella sorgente dell'albero è collocato l'elemento che ha priorità minima, e la priorità di un elemento in questa implementazione è individuabile con il metodo .getPriority(). Questa coda di minima priorità rappresentata con heap binario permette le operazioni di inserimento, di estrazione dell'elemento con priorità minima (e di aggiornamento conseguente della coda) e di diminuzione della priorità di un elemento (e di aggiornamento conseguente della coda). La struttura dati utilizzata per rappresentare l'heap è un ArrayList<>().

Per effettuare l'operazione di ***insert***, dopo aver controllato che l'elemento inserito non sia *null*, viene aggiunto l'elemento in coda all'heap, e viene aggiornato il suo indice con la dimensione corrente dell'heap sottratta di 1. Successivamente, tramite un ciclo while, l'elemento in input viene scambiato con il suo genitore nel caso in cui la priorità del nodo figlio sia minore della priorità del genitore, fino al corretto posizionamento dell'elemento in questione.

L'operazione di ***estrazione del minimo extractMinimum()*** posiziona l'ultimo elemento della coda nella radice assegnandogli l'handle corretto (0) per poi cancellarlo dall'ultima posizione (la dimensione dell'*heap* diminuisce di 1), imposta l'indice dell'elemento estratto a -1 e se la coda è vuota restituisce l'elemento estratto, imposta l'indice della radice a 0 ed esegue il metodo minHeapify(indice) sull'indice 0 per aggiornare i nodi sotto alla radice basandosi sulle loro priorità. Tramite il metodo minHeapify() viene utilizzata la tecnica della *programmazione dinamica* che divide il problema (di posizionare l'elemento nella giusta posizione dell'heap) in sottoproblemi (utilizzo del metodo .minHeapify(indice)). Inoltre come è possibile notare il metodo minHeapify(indice) è ricorsivo ed applica le stesse condizioni ad ogni esecuzione.

```
public PriorityQueueElement extractMinimum() {
   if(heap.isEmpty())throw new NoSuchElementException("This priority queue
      must contain at least an element");
   PriorityQueueElement elementToReturn = this.heap.get(0);
   this.heap.set(0, this.heap.get(this.size()-1));
   this.heap.remove(this.size()-1);
   elementToReturn.setHandle(-1);
   if(this.heap.isEmpty()) return elementToReturn;
   this.heap.get(0).setHandle(0);
   this.minHeapify(0); <-- programmazione dinamica
   return elementToReturn;
}

private void minHeapify(int index) {
   int smallestChildIndex = index;
   int leftChildIndex = (index*2)+1;
   int rightChildIndex = (index*2)+2;

   if(leftChildIndex < this.size())
      if(this.heap.get(leftChildIndex).getPriority() <
      this.heap.get(index).getPriority())
          smallestChildIndex = leftChildIndex;
   if(rightChildIndex < this.size())
      if(this.heap.get(rightChildIndex).getPriority() <
      this.heap.get(smallestChildIndex).getPriority())
          smallestChildIndex = rightChildIndex;
   if(smallestChildIndex != index) {
      swap(index, smallestChildIndex);
      minHeapify(smallestChildIndex); <-- ricorsivo
   }
}
```

L'operazione di **decremento della priorità** *decreasePriority(PriorityQueueElement, Priority)* viene effettuata tramite l'assegnamento della nuova priorità all'elemento in input, e con un ciclo while che scambia l'elemento (attraverso il metodo privato .swap(indicePrimoElemento,indiceSecondoElemento)) con il suo genitore fino al raggiungimento della corretta posizione nell'albero binario (rappresentato con un ArrayList di elementi con priorità). Più precisamente ad ogni operazione di swap l'elemento con priorità viene scambiato con il suo genitore, ottenuto tramite il metodo privato .getParent(indiceElemento).

Infine viene utilizzato il metodo **min heapify** *minHeapify(indiceElemento)* in due parti del codice scritto: nell'estrazione del minimo al fine di riordinare gli elementi nella coda in base alla loro priorità, e nel metodo minHeapify stesso (ricorsivamente) nel caso in cui l'elemento da riordinare non si trovi nella giusta posizione in base alla sua priorità.

ALGORITMO DI DIJKSTRA (problema del cammino minimo con sorgente singola)

L'algoritmo di Dijkstra permette di computare tutti i cammini minimi da una singola sorgente verso tutti i nodi di un grafo orientato e pesato. Questo algoritmo si basa sulle proprietà di un heap binario collocato in una cosa di minima priorità. Per la risoluzione dell'inizializzazione della coda e della computazione del cammino minimo è stato seguito lo pseudocodice dell'algoritmo stesso.

L'operazione di **computazione del percorso minimo da un nodo sorgente** *computeShortestPathFrom(GraphNode<L> sourceNode)* viene eseguita inizializzando un albero binario rappresentato con una coda di minima priorità, vengono impostati i nodi predecessori a null, il nodo sorgente avrà priorità 0.0 mentre tutti gli altri nodi avranno una priorità infinita. La computazione viene eseguita tramite un ciclo while che esegue l'algoritmo di Dijkstra fino a che la coda ha uno o più elementi. Nella computazione dei percorsi minimi viene utilizzata la programmazione dinamica nell'utilizzo del metodo .decreasePriority(nodo,distanza) il quale a sua volta imposta la nuova priorità all'elemento e con un ciclo while lo scambia con il nodo genitore al fine di riordinare l'elemento in base alla sua priorità.

Parte di codice relativa all'esecuzione dell'algoritmo Dijkstra:

```
while(!queue.isEmpty()){                    programmazione dinamica ↴
        GraphNode<L> minimum = (GraphNode<L>) queue.extractMinimum();
        for (GraphNode<L> node : this.graph.getAdjacentNodesOf(minimum)) {
                double dist = minimum.getFloatingPointDistance() +
                        this.graph.getEdge(minimum,node).getWeight();
                if(dist < node.getFloatingPointDistance()) { ← distanza dal nodo sorgente
                        node.setPrevious(minimum);
                        queue.decreasePriority(node, dist); ← programmazione dinamica
                }
        }
}
this.isComputed = true;
```

Per la **ricostruzione del cammino minimo**, calcolato in precedenza dall'algoritmo, si utilizza il metodo getShortestPath(GraphNode<L> targetNode) indicando il nodo destinazione. Per ricostruire il percorso formato da archi, il metodo scorre inizialmente tutti i nodi per individuare il nodo corrispondente al nodo target contenuto nel grafo, al fine di utilizzare il nodo target con l'hashcode effettivo poiché per la ricerca nel set di nodi è necessario il nodo effettivo e non una sua copia non corrispondente. Successivamente con un ciclo while si scorrono i nodi partendo dal nodo target tramite l'utilizzo dei nodi predecessori di ogni nodo, fino a risalire al nodo radice. Infine si effettua il "reverse" dell'insieme di nodi precedentemente ottenuti, in modo tale che l'insieme di archi che verrà poi ricostruito sarà ordinato dal nodo sorgente al nodo destinazione.

Si è così ottenuto l'insieme di archi ordinati corrispondenti al percorso minimo. In questo metodo per la ricostruzione del cammino minimo composto da archi non è presente programmazione dinamica (che è invece presente nella computazione del cammino minimo con algoritmo di Dijkstra).

ALGORITMO DI BELLMAN-FORD (calcolo dei cammini minimi a sorgente singola in un grafo pesato che può contenere anche pesi negativi ma non cicli negativi)

L'algoritmo di Bellman-Ford calcola tutti i percorsi minimi da un nodo sorgente ad un nodo destinazione in un grafo orientato e pesato che può contenere pesi negativi ma non cicli di peso negativo.

Per la risoluzione dell'implementazione del metodo che calcola tutti i cammini minimi da un nodo sorgente **computeShortestPathsFrom(GraphNode<L> sourceNode)** è stata utilizzata una coda di minima priorità come nell'algoritmo di Dijkstra, e con un for sono state assegnate le priorità per ogni nodo ad infinito tranne che per il nodo sorgente, inserendo infine ogni nodo nella coda di min priorità. Successivamente è stato realizzato l'algoritmo di Bellman-Ford seguendo il relativo pseudocodice.

Esecuzione dell'algoritmo di Bellman-Ford

```
for(int i = 0; i < this.graph.nodeCount()-1; i++){
        for (GraphEdge<L> edge : this.graph.getEdges()) {
                if((edge.getNode1().getFloatingPointDistance()+edge.getWeight()) <
                        < edge.getNode2().getFloatingPointDistance()){
                        edge.getNode2().setFloatingPointDistance(
                                edge.getNode1().getFloatingPointDistance()+
                                edge.getWeight());
                        edge.getNode2().setPrevious(edge.getNode1());
                }
        }
}
```

Controllo dei cicli di peso negativi

```
for (GraphEdge<L> edge : this.graph.getEdges()) {
        if((edge.getNode1().getFloatingPointDistance() + edge.getWeight()) <
                edge.getNode2().getFloatingPointDistance())
                throw new IllegalStateException("This graph contains negative-weigh cycle");
}
```

Da notare l'utilizzo dell'operazione *insert(PriorityQueueElement)* nella fase di inizializzazione al fine di inserire tutti gli elementi nella coda di minima priorità; questa operazione viene eseguita con un costo asintotico $O(\log n)$.

La **ricostruzione del cammino minimo nell'algoritmo di Bellman-Ford** per ottenere il cammino minimo verso un nodo target (destinazione) viene eseguita individuando inizialmente il nodo target all'interno del grafo tramite un ciclo for; successivamente con un ciclo while si scorrono i nodi prendendo come nodo corrente il suo nodo predecessore fino a che il nodo corrente non corrisponde al nodo sorgente. Infine viene effettuata l'operazione di *.reverse(list)* (che fa parte della classe "Collections") per restituire il percorso di archi nel giusto ordine (partendo dal nodo sorgente fino al nodo destinazione)

Ricostruzione del cammino minimo verso un nodo destinazione

```
public List<GraphEdge<L>> getShortestPathTo(GraphNode<L> targetNode) {
        if(targetNode == null) throw new NullPointerException("targetNode can't be null");
        if(!this.graph.containsNode(targetNode)) throw new IllegalArgumentException("This
                graph doesn't contain this node");
        if(!this.isComputed) throw new IllegalStateException("This graph isn't yet computed");
        ArrayList<GraphEdge<L>> listToReturn = new ArrayList<>();
        GraphNode<L> currNode = null;
        for(GraphNode<L> node : this.graph.getNodes()){
                if(node.equals(targetNode)){
                        currNode = node;
                        break;
                }
        }
        while(!currNode.equals(lastSourceNode)) {
                listToReturn.add(this.graph.getEdge(currNode.getPrevious(), currNode));
                currNode = currNode.getPrevious();
        }
        Collections.reverse(listToReturn);
        return listToReturn;
}
```

Individuazione del nodo target corretto (hash code del nodo realmente contenuto nel set di nodi)

ALGORITMO DI FLOYD-WARSHALL (per il calcolo dei cammini minimi tra tutte le coppie di nodi in un grafo orientato)

L'algoritmo di Floyd-Warshall calcola tutti i cammini minimi tra tutte le coppie di nodi di un grafo orientato e pesato che può contenere anche archi con peso negativo, ma non cicli di peso negativo. L'algoritmo di Floyd-Warshall utilizza la tecnica della programmazione dinamica poiché per il calcolo del cammino tra una coppia di nodi presuppone che sia stato già eseguito il calcolo della matrice dei costi e della matrice dei nodi predecessori, suddividendo appunto il problema del calcolo del cammino minimo in più sottoproblemi. In questa implementazione vengono verificati eventuali cicli di peso negativi successivamente alla computazione, con la stessa logica dell'algoritmo di Bellman-Ford.

Nell'**inizializzazione del calcolatore** vengono impostati tutti i costi a infinito (nella matrice dei costi) tramite *Double.POSITIVE_INFINITY*, mentre tutti i predecessori vengono impostati a -1 (nella matrice dei predecessori).

Nel metodo che effettua la **computazione dei cammini minimi** viene eseguita questa inizializzazione:

```
for (GraphEdge<L> edge : this.graph.getEdges()) {
        int u = edge.getNode1().getHandle();
        int v = edge.getNode2().getHandle();
        this.costMatrix[u][v] = edge.getWeight();
        this.predecessorMatrix[u][v] = u;
}
for (GraphNode<L> node : this.graph.getNodes()) {
        int v = node.getHandle();
        this.costMatrix[v][v] = 0;
        this.predecessorMatrix[v][v] = v;
}
```

Successivamente all'inizializzazione vengono **computati i cammini minimi** tra tutte le coppie di nodi con la seguente implementazione basata sullo pseudocodice dell'algoritmo di Floyd-Warshall.

Implementazione della computazione dei cammini minimi (calcolo bottom-up dei pesi di cammino minimo)

```
for (int h = 0; h < this.graph.nodeCount(); h++)
        for (int i = 0; i < this.graph.nodeCount(); i++)
                for (int j = 0; j < this.graph.nodeCount(); j++)
                        if(this.costMatrix[i][j] > this.costMatrix[i][h] + this.costMatrix[h][j]){
                                this.costMatrix[i][j] = this.costMatrix[i][h] + this.costMatrix[h][j];
                                this.predecessorMatrix[i][j] = this.predecessorMatrix[h][j];
                        }
```

Per ottenere infine il **cammino minimo tra due nodi** viene utilizzato il metodo **getShortestPath(sourceNode, targetNode)** che ricostruisce il percorso minimo tra i due nodi. La ricostruzione del percorso minimo (e l'algoritmo di Floyd-Warshall più in generale) utilizza la programmazione dinamica dovendo individuare il nodo intermedio tra *u* e *v* dato dalla matrice dei predecessori, la quale è stata computata precedentemente come sottoproblema. Il percorso viene ricostruito secondo questa implementazione:

Ricostruzione del percorso minimo tra due nodi (il codice evidenziato rappresenta la ricostruzione vera e propria)

```
ArrayList<GraphNode<L>> nodePath = new ArrayList<>();
int u = this.graph.getNodeIndexOf(sourceNode.getLabel());
int v = this.graph.getNodeIndexOf(targetNode.getLabel());
while(u != v) {
        nodePath.add(this.graph.getNodeAtIndex(u));
        u = predecessorMatrix[u][v];
}
nodePath.add(this.graph.getNodeAtIndex(v));
ArrayList<GraphEdge<L>> edgePath = new ArrayList<>();
for (int i = 0; i < nodePath.size()-2; i++)
        edgePath.add(this.graph.getEdge(nodePath.get(i), nodePath.get(i+1)));
return edgePath;
```

ALGORITMO DI KRUSKAL (per il calcolo di un albero minimo di copertura)

Questo algoritmo calcola il Minimum Spanning Tree di un grafo non orientato, pesato e con pesi negativi. La struttura dati utilizzata per rappresentare gli insiemi disgiunti è formata da: un ArrayList di HashSet di GraphNode, ovvero ArrayList<HashSet<GraphNode<L>>>, e per rappresentare il Minimum Spanning Tree (albero minimo di copertura) di archi viene utilizzato Set<GraphEdge<L>>. Come per tutti gli algoritmi, è stata suddivisa la computazione in inizializzazione e esecuzione dell'algoritmo. Anche in questo algoritmo la logica per individuare cicli di peso negativo è la stessa espressa precedentemente.

Per **inizializzare la computazione** sono state inizializzate le due strutture dati: *disjointSets* e *mspEdgeSet* nel seguente modo:

Inizializzazione KruskalMSP

```
mspEdgeSet.clear();
        for (GraphNode<L> node : g.getNodes()) {
                HashSet<GraphNode<L>> hashSet = new HashSet<>();
                hashSet.add(node);
                disjointSets.add(hashSet);
        }
```

Per la **computazione dell'algoritmo** invece è stato seguito lo pseudocodice di riferimento, che è stato riscritto nel seguente modo.

Computazione dell'algoritmo di Kruskal

```
for (GraphEdge<L> edge : getOrderedEdges(g)) {
        int i = getIndexOf(edge.getNode1());
        int j = getIndexOf(edge.getNode2());
        if (i != j){
                mspEdgeSet.add(edge);
                this.union(i,j);
```

Il **metodo privato *getOrderedEdges(Graph)*** è stato implementato per ottenere un Set di archi ordinati in base al loro peso in ordine crescente (come richiesto dall'algoritmo).

Il **metodo privato *union(i,j)*** è stato implementato per unire due Set disgiunti, e per eliminare uno dei due Set non più utili; più precisamente aggiunge il Set con indice *j* al Set con indice *i*, per poi rimuovere il Set con indice *j*.

ALGORITMO DI PRIM (per il calcolo di un albero minimo di copertura)

L'algoritmo di Prim calcola il Minimum Spanning Tree in un grafo non orientato, pesato e con pesi non negativi. Questo algoritmo utilizza come struttura dati una Binary Heap Min Priority Queue (albero binario rappresentato con una coda con priorità minima). L'implementazione di questo algoritmo è composta dalla sola computazione ***computeMSP(graph, node)***.

La computazione **computeMSP(Graph<L> g, GraphNode<L> s)** viene eseguita secondo due fasi successive al controllo della validità del grafo inserito e del nodo inserito: inizializzazione ed esecuzione dell'algoritmo stesso. Nell'inizializzazione vengono impostate le priorità dei nodi ad infinito (tramite Double.POSITIVE_INFINITY), i nodi predecessori vengono impostati con valore *null* e i colori di ogni nodo vengono impostati a GraphNode.COLOR_WHITE. Il nodo sorgente sarà l'unico con priorità 0.0 e tutti i nodi saranno inseriti nella coda di min. priorità tramite l'operazione *insert(element)*.

L'**esecuzione dell'algoritmo di Prim** viene eseguita tramite il ciclo *while* fino a che la coda non diventa vuota. Inizialmente viene estratto il nodo con priorità minima e il suo colore diventerà GraphNode.COLOR_BLACK, ovvero nodo fuori dalla coda con priorità; con un ciclo *for* vengono presi tutti i nodi adiacenti al nodo precedentemente estratto, e se il colore del nodo attuale è GraphNode.COLOR_WHITE (presente nella coda con priorità) e se l'arco che lo collega al nodo estratto ha peso minore della priorità del nodo attuale, allora il nodo estratto diventerà il predecessore del nodo attuale, e verrà decrementata la priorità del nodo attuale. Il codice implementato è il seguente:

Inizializzazione algoritmo di Prim

```
for (GraphNode<L> node : g.getNodes()) {
        node.setPriority(Double.POSITIVE_INFINITY);
        node.setPrevious(null);
        node.setColor(GraphNode.COLOR_WHITE);
}
s.setPriority(0.0);
for (GraphNode<L> node : g.getNodes())
        this.queue.insert(node);
```

Esecuzione algoritmo di Prim

```
while(!queue.isEmpty()){
        GraphNode<L> u = (GraphNode<L>) queue.extractMinimum();
        u.setColor(GraphNode.COLOR_BLACK);
        for (GraphNode<L> v : g.getAdjacentNodesOf(u)) {
                if(v.getColor() == GraphNode.COLOR_WHITE &&
                        && g.getEdge(u,v).getWeight() < v.getPriority()) {
                        v.setPrevious(u);
                        this.queue.decreasePriority(v,g.getEdge(u,v).getWeight());
                        }
                }
        }
```

Testo di riferimento utilizzato

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduzione agli algoritmi 3/ED*. McGraw- Hill, 2010.