RELAZIONE PROGETTO TOTALE II

ALGORITMI E STRUTTURE DATI LABORATORIO

10 SETTEMBRE 2021

MICHELE PIOLI

Matricola: 110388

ALESSIO TOZZI

Matricola: 109376

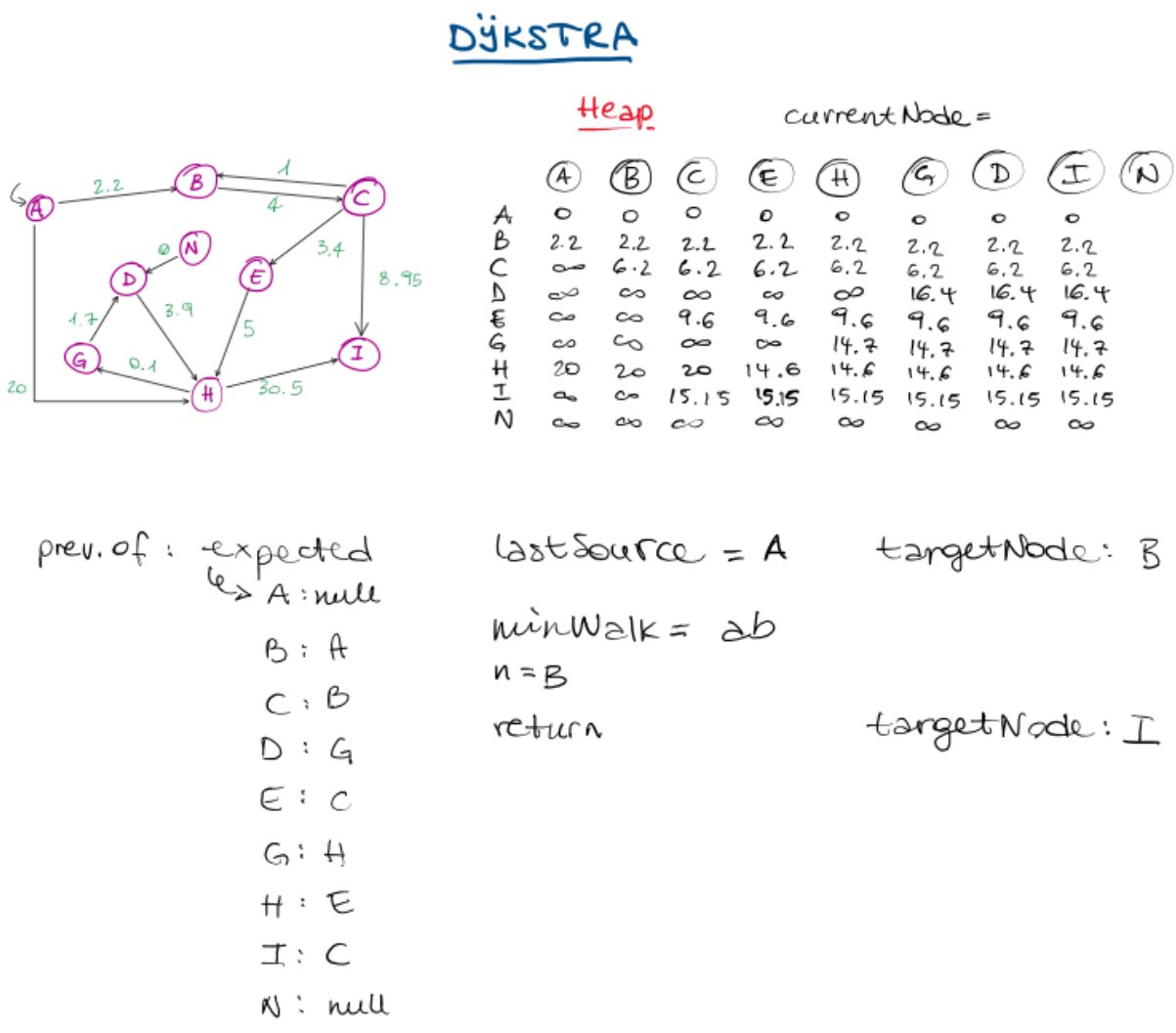
TABLE OF CONTENTS

La preparazione alla stesura del codice si è incentrata sulla ricerca e studio del funzionamento dei vari algoritmi e strutture dati. Lo studio della teoria è avvenuto tramite il libro, nonché attraverso siti web. Di particolare aiuto sono stati YouTube.com e le immagini in formato .gif su Wikipedia.com, che hanno permesso di vedere una rappresentazione più dinamica e intuitiva degli algoritmi.

In seguito sulla teoria, si è passati alla scrittura, prima delle implementazioni dei grafi, e in seguito degli algoritmi. Le parti centrali degli algoritmi sono risultate, come previsto, le più laboriose.

Algoritmo di Dijkstra

L'algoritmo di Dijkstra permette di trovare cammini minimi con sorgente singola in grafi pesati con pesi non negativi. Dopo una stesura approssimativa dell'algoritmo di Dijkstra si è passati ai test, grazie ai quali si sono potute osservare le mancanze nel codice. Un aiuto decisivo è stato fornito dalla realizzazione per iscritto di un grafo (raffigurato sotto), che è poi stato riutilizzato con piccole variazioni per gli altri algoritmi, e la scrittura della relativa struttura dati seguendo l'algoritmo precedentemente scritto.



Seguendo passo passo il codice siamo riusciti ad individuare quali sezioni del codice fosse necessario correggere. Un esempio di codice di codice di Dijkstra che ha richiesto lavoro è il seguente (righe 78-87, DijkstraShortestPathComputer):

```
GraphNode<L> node;
for (int i = 0; i < this.graph.nodeCount(); i++) {
    node = (GraphNode<L>) this.heap.extractMinimum();
    if (node.getPriority() == Double.POSITIVE_INFINITY) {
        node.setPrevious(null);
        continue;
    }
    node.setColor(GraphNode.COLOR_BLACK);
    checkNewPriority(node);
}
```

Questo è il fulcro dell'algoritmo di Dijkstra. Il ciclo for itera tutti i nodi nel grafo al fine di trovare il nodo con priorità inferiore ed estrarlo dalla struttura dati (heap, in questa implementazione). In seguito, si controlla che non vi siano nodi che non possono essere raggiunti dal nodo di partenza. Se ve ne dovessero essere, questi saranno privi di previous node, e dunque questo verrà posto a `null`. Successivamente, si pone nero il colore del nodo visitato. Infine, per semplificare e rendere modulare il metodo, si è scelto di creare un metodo separato per il controllo e assegnazione della priorità ai nodi (`checkNewPriority(node)`).

ALGORITMO DI BELLMAN-FORD

L'algoritmo di Bellman-Ford è molto simile a quello di Dijkstra, infatti anch'esso permette di trovare il percorso minimo da sorgente singola, ma si discosta perché permette anche di avere archi negativi (ma non cicli negativi). Il numero massimo di iterazioni del grafo è pari al numero di nodi nel grafo. Con lo scopo di calcolare i cicli negativi è stato aggiunta un'iterazione aggiuntiva del grafo di modo che, se questa dovesse continuare a modificare le priorità in calando, significa che ci sono cicli negativi che tendono a -∞. Come si vede qui di seguito, il ciclo for include un'iterazione aggiuntiva (riga 119, BellmanFordShortestPathComputer):

```
for (i = 0; i < this.graph.nodeCount(); i++)
```

L'algoritmo originale infatti vorrebbe `nodeCount() - 1` come implementazione standard.

Il controllo del ciclo negativo avviene anche tramite la variabile `boolean hasImproved`, che per default è false, ma in caso di cambiamenti deve essere posta true. Vediamo nel frammento di codice qui sotto (righe 136-142, ibid.) che infatti se `hasImproved` è true alla conclusione dell'ultima iterazione si ha un ciclo negativo, e viene lanciata un'eccezione:

```
if (hasImproved) {
       if (i == (this.graph.nodeCount() - 1))
              throw new IllegalStateException();
       […]
```

Di nuovo, di grande assistenza è stato l'utilizzo di carta e penna per l'osservazione passo per passo dell'algoritmo implementato, che ci ha permesso di osservare più nel dettaglio il funzionamento del codice.

ALGORITMO DI FLOYD-WARSHALL

L'algoritmo di Floyd-Warshall calcola il cammino minimo tra coppie di nodi, con grafo orientato e la possibilità di avere archi con peso negativo (ma non cicli negativi). L'implementazione è stata effettuata tramite una matrice che contiene il costo minimo da ogni nodo ad ogni altro nodo nel grafo.

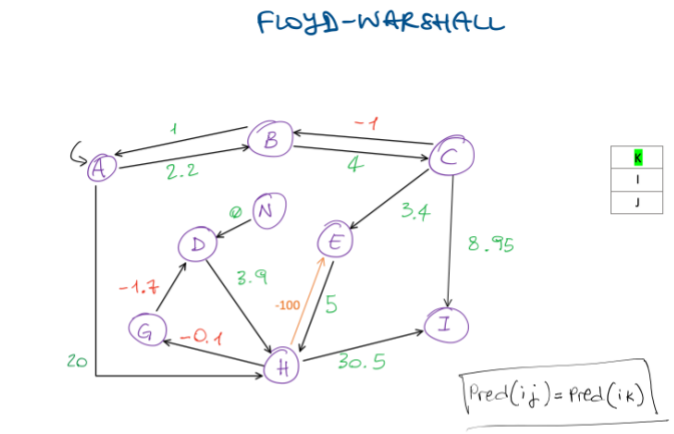
La compilazione della matrice avviene attraverso il metodo `void computeShortestPaths()`. Inizialmente il controllo per il ciclo negativo era stato implementato con un loop aggiuntivo a quello che compila la matrice, come qui di seguito:

```
boolean hasImproved = false;
for (int k = 0; k < this.graph.nodeCount(); k++) {
    for (int i = 0; i < this.graph.nodeCount(); i++) {
        for (int j = 0; j < this.graph.nodeCount(); j++) {
            if (i == j && costMatrix[i][j] < 0.0)
                throw new IllegalStateException();
            if (costMatrix[i][j] <= (costMatrix[i][k] + costMatrix[k][j]))
                continue;
            else costMatrix[i][j] = (costMatrix[i][k] + costMatrix[k][j]);
            predecessorMatrix[i][j] = k;
            if (!hasImproved)
                hasImproved = true;
        }
    }
}
for (int k = 0; k < this.graph.nodeCount(); k++)
    for (int i = 0; i < this.graph.nodeCount(); i++)
        for (int j = 0; j < this.graph.nodeCount(); j++)
            if (i == j && costMatrix[i][j] < 0)
                throw new IllegalStateException();
this.isComputed = true;
```

Tuttavia, per assicurare coerenza tra i vari algoritmi implementati si è scelto di inserire il controllo per il ciclo negativo all'interno del primo loop, come già fatto in Bellman-Ford (righe 93-113, FloydWarshallAllPairsShortestPathsComputer):

```
boolean hasImproved = false;
for (int k = 0; k < this.graph.nodeCount() + 1; k++) {
    for (int i = 0; i < this.graph.nodeCount(); i++) {
        for (int j = 0; j < this.graph.nodeCount(); j++) {
            if (k == this.graph.nodeCount()) {
                if (i == j && costMatrix[i][j] < 0.0)
                    throw new IllegalStateException();
                continue;
            }
            if (i == j && costMatrix[i][j] < 0.0)
                throw new IllegalStateException();
            if (costMatrix[i][j] <= (costMatrix[i][k] + costMatrix[k][j]))
                continue;
            else costMatrix[i][j] = (costMatrix[i][k] + costMatrix[k][j]);
            predecessorMatrix[i][j] = k;
            if (!hasImproved)
                hasImproved = true;
        }
    }
}
this.isComputed = true;
```

Per la stesura e controllo dei test si è nuovamente fatto uso di carta e penna, e sono state calcolate le matrici di costo e predecessore, così da poter scrivere i test più agevolmente:



cost matrix

	A	B	C	D	E	G	H	I	N
A	0.0	2.2	Inf	Inf	Inf	Inf	20	Inf	Inf
B	1.0	0.0	4.0	10.6	7.4	12.3	12.4	12.95	Inf
C	0.0	-1.0	0.0	6.6	3.4	8.3	8.4	8.95	Inf
D	Inf	Inf	Inf	0.0	Inf	3.8	3.9	34.4	Inf
E	Inf	Inf	Inf	3.2	0.0	4.9	5	35.5	Inf
G	Inf	Inf	Inf	-1.7	Inf	0.0	2.2	32.7	Inf
H	Inf	Inf	Inf	-1.8	Inf	-0.1	0.0	30.5	Inf
I	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0.0	Inf
N	Inf	Inf	Inf	0.0	Inf	3.8	3.9	34.4	0.0

predecessor matrix

	A	B	C	D	E	G	H	I	N
A (0)	0	0	1	5	2	6	4	2	-1
B (1)	1	1	1	5	2	6	4	2	-1
C (2)	1	2	2	5	2	6	4	2	-1
D (3)	-1	-1	-1	3	-1	6	3	6	-1
E (4)	-1	-1	-1	5	4	6	4	6	-1
G (5)	-1	-1	-1	5	-1	5	3	6	-1
H (6)	-1	-1	-1	5	-1	6	6	6	-1
I (7)	-1	-1	-1	-1	-1	-1	-1	7	-1
N (8)	-1	-1	-1	8	-1	6	3	6	8

ALGORITMO DI KRUSKAL

L'algoritmo di Kruskal permette di trovare il minimum spanning tree di un grafo non orientato con archi con peso non negativo.

In questo algoritmo si è scelto di suddividere le casistiche su base di colorazione dei vari nodi nel grafo (righe 72-84, KruskalMSP):

```
    if (minEdge.getNode1().getColor() == GraphNode.COLOR_WHITE &&
            minEdge.getNode2().getColor() == GraphNode.COLOR_WHITE) {
        bothWhite(minEdge);
    } else if (minEdge.getNode1().getColor() == GraphNode.COLOR_BLACK &&
            minEdge.getNode2().getColor() == GraphNode.COLOR_BLACK) {
        bothBlack(minEdge);
    } else
        blackNWhite(minEdge);

    edgeSet.remove(minEdge);
    minWeight = Double.MAX_VALUE;
}
return this.msp;
```

Se i nodi sono entrambi bianchi, è da interpretarli come entrambi ancora assenti da un Set disgiunto, pertanto formeranno un nuovo Set disgiunto composto solo dall'arco che li connette (bothWhite, riga 126, ibid.).

Se i nodi sono uno bianco e l'altro nero, significa che solo uno dei due appartiene ad un Set, pertanto l'arco che li connette verrà aggiunto al Set preesistente.

Nel caso in cui siano entrambi neri, significa che bisogna fare dei controlli aggiuntivi.

Prima ci si assicura che i due nodi appartengano a due Set diversi (righe 94-105, ibid.):

```
ArrayList<HashSet<GraphNode<L>>> support = this.disjointSets;
HashSet<GraphNode<L>> secondSet = new HashSet<>();
int i = 0;
for (HashSet<GraphNode<L>> currentSet : support) {
    if (!currentSet.contains(edge.getNode1())) //uno è già nero
        continue;
    if (currentSet.contains(edge.getNode2())) {
        i = -1;
        break;
    }
    i++;
}
```

In seguito, se dovessero appartenere allo stesso Set, ci troveremmo di fronte ad un potenziale loop, e pertanto l'arco che li collega non verrebbe aggiunto al Set (righe 106-109, ibid.):

```
if (i == -1) {
    this.disjointSets.remove(edge);
    return;
}
```

Infine, rimane il caso in cui i due nodi siano appartenenti a due Set diversi. Se così fosse, i due Set dovrebbero essere uniti (righe 110-118, ibid.):

```
for (HashSet<GraphNode<L>> otherSet : support)
    if (otherSet.contains(edge.getNode2()))  //entrambi neri
        secondSet = otherSet;
if (secondSet.isEmpty())
    return;

this.disjointSets.get(i).addAll(secondSet);
this.disjointSets.remove(secondSet);
this.msp.add(edge);
```

ALGORITMO DI PRIM

L'algoritmo di Prim, come anche quello di Kruskal, trova il minimum spanning tree di un grafo non orientato e con archi non negativi. La differenza sta nel fatto che, dove Kruskal si muove di arco in arco sulla base del peso (dal peso minore al peso maggiore del grafo), nel caso dell'algoritmo di Prim ci si sposta di nodo in nodo trovando gli adiacenti dello stesso.

Partendo da un nodo `s` (per start) si osservano i suoi adiacenti per trovare quello con il costo minore, dove poi ci si sposta per cercare a sua volta l'arco con costo minore, e così via (righe 75-85, PrimMSP):

```
Iterator<GraphNode<L>> iter = g.getAdjacentNodesOf(node).iterator();
while (iter.hasNext()) {
    adjNode = iter.next();
    if (adjNode.getColor() == GraphNode.COLOR_BLACK)
        continue;
    if (adjNode.getPriority() > g.getEdge(node, adjNode).getWeight()) {
        this.queue.decreasePriority(adjNode, g.getEdge(node, adjNode).getWeight());
```

```
            adjNode.setColor(GraphNode.COLOR_GREY);
            adjNode.setPrevious(node);
        }
    }
```

L'iteratore scorre i nodi adiacenti (`adjNode`) al nodo in esame (`node`). Per prima cosa osserva il colore di `adjNode`, se questo dovesse essere nero, starebbe a significare che è già parte del minimum spanning tree, e dunque verrà ignorato. Se invece non lo fosse, bisognerebbe controllare la priorità. Gli `adjNode` possono avere priorità $+\infty$ se bianchi (ovvero mai visitati), oppure una priorità già assegnatagli nel caso l'`adjNode` fosse grigio (e dunque fosse stato già visitato). Dopo essersi assicurati che la priorità preesistente di `adjNode` sia maggiore di quella trovata da `node`, si aggiorna la priorità, si imposta il colore grigio e si assegna il precedente a `adjNode`.