

**Università degli Studi di Palermo**

**Corso di Laurea in Informatica**

**Esame di “Laboratorio di Algoritmi”**

**Relazione della prova pratica**

**Habasescu Alin Marian**

**Soluzione prova pratica**

Data di consegna: September 5, 2024

**Contents**

# 1 Soluzione proposta

Il progetto richiede di minimizzare la perdita d'acqua nel trasporto tra acquedotti, modellando la rete idrica come un grafo pesato dove i nodi rappresentano gli acquedotti e i lati bidirezionali rappresentano le tubazioni con un determinato livello di perdita d'acqua espressa in millilitri al secondo.

Il problema principale sta nel voler calcolare i millilitri al secondo che vengono persi lungo una sequenza di tubi, da un certo acquedoto S ad un altro acquedoto T, individuando la minor perdita possibile.

Per risolvere il problema, ho scelto di applicare l'algoritmo di Dijkstra, che è particolarmente adatto per trovare il percorso minimo in termini di peso, da un vertice S detto "sorgente" ad un vertice T detto "destinazione", purché i pesi degli archi siano non negativi.

## 1.1 Algoritmi e codici utilizzati

Durante lo svolgimento del corso, mi è stato molto utile creare una repository GitHub con diverse implementazioni di algoritmi e strutture dati. Per il completamento del progetto ho utilizzato alcuni algoritmi e strutture dati da me stesso implementate, tra cui l'algoritmo di Dijkstra, la classe `WeightedGraph`, la classe `Heap`.

Il codice sorgente completo può essere consultato al seguente link:

<https://github.com/H-Alin02/DSA-in-cpp>

All'interno del progetto, ho riutilizzato alcune classi precedentemente sviluppate per la gestione di grafi pesati e l'implementazione di algoritmi sui grafi. In particolare, ho utilizzato:

- La classe `WeightedGraph` gestisce la struttura dati del **grafo pesato**, che viene implementata tramite lista di adiacenza o matrice di adiacenza.
- L'algoritmo di **Dijkstra** per il calcolo del percorso minimo tra nodi in un grafo pesato, che è stato implementato nella classe `GraphAlgorithms`.

## 1.2 Algoritmo di Dijkstra

L'algoritmo di Dijkstra per il calcolo del percorso minimo è stato implementato nella classe `GraphAlgorithms`. Di seguito è riportata una versione dell'algoritmo:

```

1  // Dijkstra implementation to find the shortest path from start to
    destination
2  static int dijkstra(int start, int end, const GraphType& graph) {
3      int nodes = graph.getVertices();
4
5      if (start < 0 || start >= nodes || end < 0 || end >= nodes) {
6          throw out_of_range("Invalid start or end node");
7      }
8
9      // Priority queue to select the node with the smallest distance
10     Heap<pair<T, double>> pq(
11         [](const pair<T, double>& a, const pair<T, double>& b) {
12             return a.second < b.second;
13         }, true);
14
15     // Initialize distances to infinity
16     vector<double> dist(nodes, numeric_limits<double>::infinity());
17     dist[start] = 0;
18     pq.push({start, 0});
19
20     while (!pq.empty()) {
21         int u = pq.top().first;
22         int u_dist = pq.top().second;
23         pq.pop();
24
25         if (u == end)
26             return u_dist;
27
28         for (const auto &neighbor : graph.getEdges(u)) {
29             int v = neighbor.first.second;
30             double weight = neighbor.second;
31
32             if (dist[v] > u_dist + weight) {
33                 dist[v] = u_dist + weight;
34                 pq.push({v, dist[v]});
35             }
36         }
37     }
38
39     return -1;
40 }

```

L'algoritmo utilizza una **coda di priorità** (Min-Heap) per estrarre il nodo con la distanza minore e aggiornare le distanze per i nodi adiacenti. Il codice gestisce sia grafi con lista di adiacenza che grafi pesati.

Nel contesto del progetto attuale, queste classi sono state riutilizzate per implementare una serie di funzionalità, tra cui il calcolo dei percorsi minimi tra i nodi e la visualizzazione di grafi pesati. Il riuso del codice ha consentito una notevole riduzione della complessità, poiché le classi e gli algoritmi erano già testati e ottimizzati.

Le implementazioni sono state facilmente adattate alle specifiche del progetto, permettendo l'integrazione efficiente di funzionalità come la gestione di grafi con pesi variabili e l'esecuzione di algoritmi di ricerca del percorso minimo.

## 2 Correttezza dell'algoritmo

Analizziamo ora la correttezza dell'implementazione dell'algoritmo di Dijkstra confrontandolo con il suo Pseudocodice originale:

### ALGORITMO DI DIJKSTRA: PSEUDOCODICE

```
begin
1.    $S \leftarrow \{v_0\};$ 
2.    $D[v_0] \leftarrow 0;$ 
3.   for each  $v$  in  $V - \{v_0\}$  do  $D[v] \leftarrow l(v_0, v);$ 
4.   while  $S \neq V$  do
       begin
5.         choose a vertex  $w$  in  $V - S$  such that  $D[w]$  is a minimum;
6.         add  $w$  to  $S;$ 
7.         for each  $v$  in  $V - S$  do
8.            $D[v] \leftarrow \text{MIN}(D[v], D[w] + l(w, v))$ 
       end
end
```

[A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974]

Figure 1: Pseudocodice dell'algoritmo di Dijkstra.

1. **Inizializzazione del set  $S$  e del vettore  $\text{dist}$ :** Nell'implementazione, il set  $S$  viene implicitamente gestito attraverso l'uso di una coda a priorità (implementata con un Min-Heap). Se un nodo è estratto dalla coda significa che il percorso minimo per quel nodo è già stato trovato. Il vettore "dist" viene inizializzato con infinito per tutti i nodi, tranne il nodo di partenza, il cui costo è impostato a 0.
2. **Aggiornamento delle Distanze:** Durante l'iterazione, se viene trovato un percorso più breve verso un nodo, la distanza viene aggiornata e il nodo viene reinserito nella coda di priorità con la nuova distanza. Questo passaggio viene implementato con la condizione  $\text{dist}[v] > \text{u\_dist} + \text{weight}$ , che aggiorna le distanze e aggiunge il nodo  $v$  alla coda di priorità con la nuova distanza.
3. **Condizione di Terminazione:** Nello pseudocodice l'algoritmo continua fino a quando tutti i nodi non sono stati elaborati (cioè  $S = V$ ), nell'implementazione l'algoritmo si interrompe non appena viene trovato il nodo di destinazione

### 3 Alcuni esempi di esecuzione

Di seguito vengono riportati alcuni esempi di esecuzione dell'algoritmo su piccoli grafi.

#### Esempio 1

- **Input:** 5 acquedotti, 7 tubazioni, partenza da  $S=0$ , arrivo a  $T=4$ .
- **Output:** La minima perdita da 0 a 4 è  $9ml/s$ .

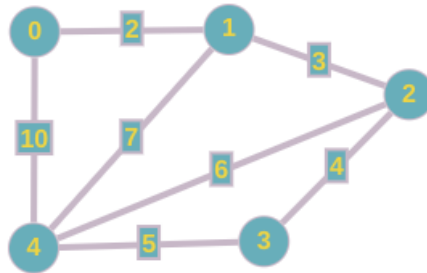


Figure 2: Grafo 1.

Iteration	S	w	D[w]	D[1]	D[2]	D[3]	D[4]
Initial	0	-	-	2	$+\infty$	$+\infty$	10
1	0,1	1	2	2	5	$+\infty$	9
2	0,1,2	2	5	2	5	9	9
3	0,1,2,3	3	9	2	5	9	9
4	All	4	9	2	5	9	9

Table 1: Esecuzione del algoritmo di Dijkstra per il nodo 0 sul grafo 1

## Esempio 2

- **Input:** 7 acquedotti, 8 tubazioni, partenza da S=0, arrivo a T=6.
- **Output:** La minima perdita da 0 a 6 è  $19ml/s$ .

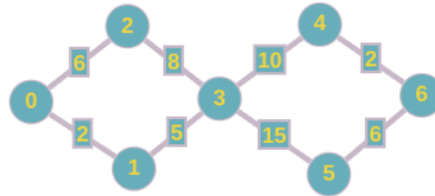


Figure 3: Grafo 2.

Iteration	S	w	D[w]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
Initial	0	-	-	2	6	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1	0,1	1	2	2	6	7	$+\infty$	$+\infty$	$+\infty$
2	0,1,2	2	6	2	6	7	$+\infty$	$+\infty$	$+\infty$
3	0,1,2,3	3	7	2	6	7	17	22	$+\infty$
4	0,1,2,3,4	4	17	2	6	7	17	22	19
5	All	6	19	2	6	7	17	22	19

Table 2: Esecuzione del algoritmo di Dijkstra per il nodo 0 sul grafo 2

### Esempio 3

- **Input:** 9 acquedotti, 14 tubazioni, partenza da S=0, arrivo a T=8.
- **Output:** La minima perdita da 0 a 8 è  $14ml/s$ .

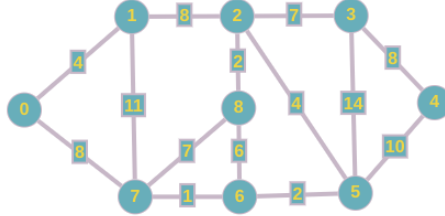


Figure 4: Grafo 3.

Iteration	S	w	D[w]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]	D[8]
Initial	0	-	-	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	8	$+\infty$
1	0,1	1	4	4	12	$+\infty$	$+\infty$	$+\infty$	$+\infty$	8	$+\infty$
2	0,1,7	7	8	4	12	$+\infty$	$+\infty$	$+\infty$	9	8	15
3	0,1,7,6	6	9	4	12	$+\infty$	$+\infty$	11	9	8	15
4	0,1,7,6,5	5	11	4	12	25	21	11	9	8	15
5	0,1,7,6,5,2	2	12	4	12	19	21	11	9	8	14
6	0,1,7,6,5,2,8	8	14	4	12	19	21	11	9	8	14
7	0,1,7,6,5,2,8,3	3	19	4	12	19	21	11	9	8	14
8	All	4	21	4	12	19	21	11	9	8	14

Table 3: Esecuzione dell'algoritmo di Dijkstra per il nodo 0 sul grafo 3.

## 4 Complessità di tempo e spazio

La complessità temporale dell'algoritmo di Dijkstra dipende dall'implementazione della coda a priorità, che in questo caso è fatta tramite **Heap binario**, e dal numero di archi e di vertici del grafo.

### 1. Operazioni sulla coda di priorità:

- **testbfInserimento**: l'inserimento di nuovi elementi nell'heap binario ha un costo di  $\mathcal{O}(\log V)$  necessario per mantenere l'ordinamento.
- **Estrazione del minimo**: l'estrazione del nodo con la distanza minima ha anch'essa complessità  $\mathcal{O}(\log V)$ , poichè la struttura deve essere aggiornata.

### 2. Esplorazione del grafo:

- Dijkstra visita ogni nodo una volta, quindi esegue al massimo  $V$  operazioni di pop dalla coda. Inoltre esamina ogni arco una volta per aggiornare le distanze. Ogni arco comporta un'operazione di push o un aggiornamento di un nodo nella coda. Poiché l'inserimento e l'aggiornamento nella coda di priorità hanno complessità  $\mathcal{O}(\log V)$ , e ci sono  $E$  archi, in totale eseguiamo  $E$  operazioni di push.

Riassumendo, si ha in totale una complessità temporale di  $\mathcal{O}(E + V \log V)$ .

In termini di complessità spaziale, il grafo è rappresentato con una lista di adiacenza che occupa  $\mathcal{O}(V + E)$ . La coda con priorità memorizza fino a  $V$  nodi, il che porta la complessità spaziale complessiva a  $\mathcal{O}(V + E)$ . Questa complessità è ottimale per il problema, dato il limite sugli acquedotti ( $n \leq 200$ ) e sui tubi ( $m \leq 5000$ ).

## 5 Strutture dati utilizzate

Per rappresentare il grafo, ho utilizzato due possibili rappresentazioni:

- **Lista di adiacenza**: Questa struttura è stata utilizzata per ridurre lo spazio necessario per grafi sparsi. Permette di memorizzare efficientemente i vicini di ogni nodo e le loro perdite associate.
- **Matrice di adiacenza**: Utilizzata per rappresentare grafi più densi, anche se meno efficiente in termini di spazio.

Per la gestione della selezione del nodo successivo con la minima distanza, è stata utilizzata una **coda con priorità** implementata come min-heap, che permette l'accesso al nodo con il peso minore in tempo  $\mathcal{O}(\log V)$ . Questo garantisce che l'algoritmo funzioni in modo efficiente anche con un numero elevato di nodi.