Università degli Studi di Palermo

Corso di Laurea in Informatica

Esame di “Laboratorio di Algoritmi"

Relazione della prova pratica

Vincenzo Barreca (matricola:0733568)

Manuele Mormorio (matricola:0732451)

Salvatore Giuseppe Amato (matricola:0733236)

Data di consegna: 28/05/2024

Titolo della prova pratica: Internetlandia

Soluzione proposta:

La soluzione proposta implementa un algoritmo per trovare percorsi più brevi tra coppie di nodi in un grafo utilizzando la ricerca in ampiezza (BFS).

Il grafo è rappresentato tramite liste di adiacenza, implementate come un array dinamico di insiemi ordinati (`set`). Tale set permette di mantenere le adiacenze dei nodi in ordine crescente, consentendo in caso di multi-path di ottenere tra tutti i path possibili quello con il primo 'ID' diverso minore.

L'algoritmo inizia leggendo i dati del grafo da un file di input, che include:

- il numero di nodi
- i nodi adiacenti a ciascun nodo, dunque gli archi
- le coppie di nodi per cui si deve calcolare lo shortest path.

Il grafo viene costruito dinamicamente, e ogni arco viene aggiunto utilizzando il metodo `addEdge`, che inserisce il nodo di destinazione nella lista di adiacenza del nodo di partenza.

Per ciascuna coppia di nodi, il percorso più breve, se esiste, viene trovato utilizzando la visita BFS del grafo.

Viene quindi inizialmente creato un array di booleani per tenere traccia dei nodi visitati e un array di interi per memorizzare i genitori di ciascun nodo, permettendo così di ricostruire il percorso una volta raggiunto il nodo di destinazione. La BFS inizia dal nodo di partenza, segnandolo come visitato e aggiungendolo a una coda. Successivamente, esplora tutti i nodi adiacenti non ancora visitati, li segna come visitati, aggiorna il loro genitore e li aggiunge alla coda.

Al termine della BFS si distinguono due casi:

- Se il nodo di destinazione viene visitato, l'algoritmo ricostruisce il percorso tracciando i genitori a ritroso dal nodo di destinazione al nodo di partenza e quindi invertendo il percorso per ottenere l'ordine corretto.
- Se il nodo di destinazione non viene visitato, viene segnalato che non esiste un tale path.

L'implementazione assicura che tutte le risorse di memoria allocate dinamicamente vengano liberate correttamente, prevenendo perdite di memoria. Il programma include controlli per garantire che il numero di nodi e il numero di adiacenze per ciascun nodo siano entro i limiti specificati, migliorando la robustezza del codice.

Correttezza dell'algoritmo

BFS calcola gli shortest path dal nodo sorgente 's' a tutti gli altri nodi del grafo ad esso connessi in tempo O(E + V).

DIMOSTRAZIONE

Per un dato intero k > 0, la coda contiene sempre zero o più vertici di distanza 'k' dalla sorgente, seguiti da zero o più vertici di distanza k + 1. I vertici entrano nella coda e lasciano la coda in ordine rispetto alla loro distanza dalla sorgente, e quindi l'algoritmo è corretto.
La marcatura assicura che ogni vertice connesso a s è visitato una volta, e il controllo di marcatura richiede tempo proporzionale al grado. Ovvero, in totale:
$O(V + \Sigma_{v \in V} deg(v)) = O(V + E)$

Complessità di tempo e spazio

Tempo:

1. Costruzione del Grafo:
    - ogni nodo viene visitato una volta per leggere i suoi vicini. Poiché ci sono V nodi e, nel peggiore dei casi, ogni nodo ha 25 vicini, la complessità temporale è |V| * 25;
    - la costruzione del grafo richiede O(V + E) operazioni.
2. BFS (Breadth-First Search):
    - Il BFS visita ogni nodo e ogni arco una volta, dunque la complessità temporale è O(V + E), dove V è il numero di nodi e E è il numero di archi nel grafo.
3. Lettura e Stampa dei Percorsi Minimi:
    - Poiché ognuno degli m percorsi richiede una visita BFS del grafo a partire dal nodo sorgente, la complessità temporale è O(m * tempo_BFS).

Complessità Totale di Tempo: O(m * (V + E)), dove V è il numero di nodi nella rete, E è il numero di archi e m è il numero di shortest path da calcolare.

Spazio:

1. Grafo:
    - La rappresentazione del grafo utilizza un array dinamico di set, con ogni set contenente al massimo 25 nodi (vicini). Quindi, lo spazio utilizzato, nel caso peggiore, per rappresentare il grafo è |V| * 25.

2. Variabili Ausiliarie del BFS:

    - L'algoritmo utilizza O(V) spazio per memorizzare l'array visited e l'array parent.
    - Utilizza inoltre O(E) spazio per memorizzare le liste di adiacenza.
    - Oltre agli array visited e parent, il BFS utilizza una ***coda q*** per gestire l'esplorazione. Lo spazio occupato da queste variabili dipende dal numero di nodi e archi nel grafo.
    - La complessità di spazio per le variabili ausiliarie del BFS è O(V + E), dove V è il numero di nodi e E è il numero di archi nel grafo.

3. Allocazione e Deallocazione della Memoria:

    - Viene allocata memoria per i set, gli array e altre variabili locali. Questo spazio è trascurabile rispetto agli altri utilizzi.

Complessità Totale di Spazio: O(V + E), dove V è il numero di nodi nella rete e E è il numero di archi nel grafo.

Strutture dati utilizzate

Le strutture dati utilizzate in questo codice sono fondamentali per la rappresentazione del grafo e l'esecuzione dell'algoritmo BFS (Breadth-First Search). Esse sono:

1. ***set<int> *adjList***, per la lista di adiacenza:

    - *Ruolo*: La lista di adiacenza è utilizzata per memorizzare i nodi adiacenti a ciascun nodo del grafo.

    - *Descrizione*: Ogni nodo del grafo ha associato un set di interi che rappresentano i nodi adiacenti. Utilizzando un set, garantiamo che i nodi adiacenti siano ordinati in modo crescente, consentendo in caso di multi-path di ottenere tra tutti i path possibili quello con il primo ‘ID’ diverso minore.

2. ***bool *visited***, per tenere traccia dei nodi visitati:

    - *Ruolo*: Utilizzato per segnare i nodi visitati durante l'esecuzione dell'algoritmo BFS.

    - *Descrizione*: Un array di booleani viene utilizzato per segnare i nodi visitati durante l'esecuzione dell'algoritmo BFS. Ciò aiuta a evitare di visitare più volte lo stesso nodo, previene cicli infiniti nel caso di grafi con cicli e ci consente di stabilire se non esiste un path tra i due nodi richiesti.

3. ***int *parent***, per tenere traccia dei genitori dei nodi:

    - *Ruolo*: Utilizzato per tenere traccia dei genitori dei nodi durante l'esecuzione dell'algoritmo BFS.

    - *Descrizione*: Un array di interi viene utilizzato per memorizzare il nodo genitore di ciascun nodo durante l'esecuzione dell'algoritmo BFS. Questo viene utilizzato successivamente per ricostruire il percorso più breve tra due nodi.

4. ***queue<int>***, per la gestione dell'esplorazione BFS:

    - *Ruolo*: Utilizzata come struttura dati FIFO per gestire l'esplorazione BFS.

    - *Descrizione*: Una coda viene utilizzata per mantenere l'ordine di esplorazione dei nodi durante l'algoritmo BFS. I nodi vengono inseriti nella coda man mano che vengono visitati e vengono esplorati in ordine FIFO, garantendo che vengano prima esplorati i nodi più vicini al nodo di partenza.

5. ***stack<int>***, per ricavare lo shortest path al termine della BFS:

    - *Ruolo*: Lo stack è utilizzato per ottenere il percorso più breve tra due nodi del grafo.

    - *Descrizione*: Quando viene trovato un percorso più breve tra due nodi, viene memorizzato all'interno di uno stack. Il suo inserimento in cima consente la costruzione dello shortest path, se esiste, a partire dal vettore parent.

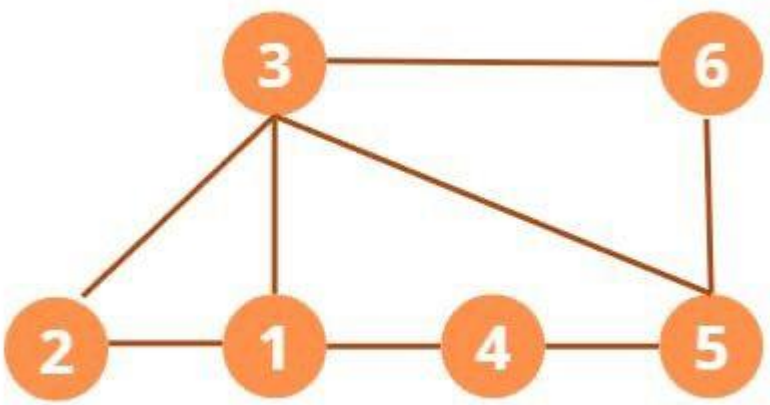
Ogni struttura dati svolge un ruolo specifico e cruciale nell'esecuzione del codice, garantendo una rappresentazione efficiente del grafo e facilitando l'implementazione e l'esecuzione dell'algoritmo BFS per trovare i percorsi minimi.

ESEMPI DI INPUT / OUTPUT

Esempio 1 - esempio di input presente nella traccia

Input

```
6
1-2,3,4
2-1,3
3-1,2,5,6
4-1,5
5-3,4,6
6-3,5
6
1 6
1 5
2 4
2 5
3 6
2 1
```



Output

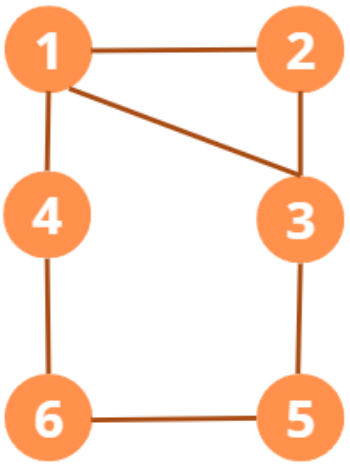
```
C:\Users\user\Desktop\Progetto Laboratorio di algoritmi>g++ -o Project Project.cpp

C:\Users\user\Desktop\Progetto Laboratorio di algoritmi>Project
1 3 6
1 3 5
2 1 4
2 3 5
3 6
2 1
```

Esempio 2 - Grafo connesso

Input

```
6
1-2,3,4
2-3,1
3-2,5,1
4-6,1
5-3,6
6-4,5
2
6 1
4 3
```



Output

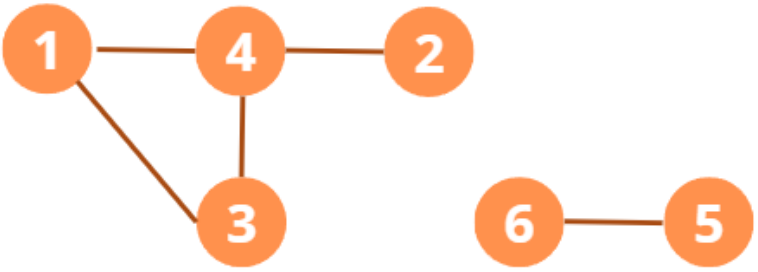
```
C:\Users\user\Desktop\Progetto Laboratorio di algoritmi>g++ -o Project Project.cpp

C:\Users\user\Desktop\Progetto Laboratorio di algoritmi>Project
6 4 1
4 1 3
```

Esempio 3 - Grafo NON connesso / NO PATH

Input

```
6
1-3,4
2-4
3-1,4
4-2,1,3
5-6
6-5
5
1 2
2 3
1 5
4 6
6 5
```



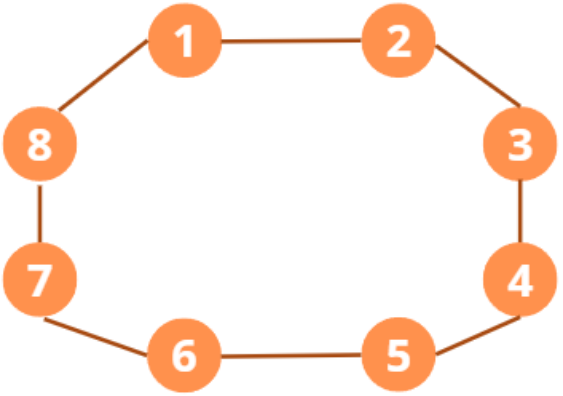
Output

```
1 4 2
2 4 3
Path non esiste
Path non esiste
6 5
```

Esempio 4 - Grafo Connesso Multi-Path

Input

```
8
1-8,2
2-3,1
3-2,4
4-3,5
5-6,4
6-5,7
7-6,8
8-1,7
3
1 3
2 6
8 4
```



Output

```
C:\Users\user\Desktop\Progetto Laboratorio di algoritmi>g++ -o Project Project.cpp

C:\Users\user\Desktop\Progetto Laboratorio di algoritmi>Project
1 2 3
2 1 8 7 6
8 1 2 3 4
```