Università degli Studi di Palermo

Corso di Laurea in Informatica

Esame di “Laboratorio di Algoritmi"

Relazione della prova pratica

Andrea Fazio
Giuseppe Rosa

27/05/2024

Prova pratica Gruppo 4 - Appello 05 giugno 2024

Soluzione proposta

Algoritmo(File input)

```
N ← il primo valore letto nel file input;

IF (N < 1 OR  N > 100 ) THEN
        return -1;

inizializza array square[NxN] con gli NxN valori presenti nel file input;
inizializza coda Q con gli indici delle posizioni delle celle con 3 nel file input;
visited_distance[NxN] ← 0;
max_of_min_distances ← - 1;

WHILE !Q->empty() DO
        c ← Q.pop();

        IF square[c] == 1 AND visited_distance[c] > max_of_min_distances THEN
                max_of_min_distances ← visited_distance[c];

        IF top_cell(c) è uno spostamento valido AND visited_distance[top_cell(c)] == 0  AND square[c] != 3  THEN
                Q.push(top_cell(c));
                visited_distance[top_cell(c)] ← visited_distance[c] + 1;

        IF right_cell(c) è uno spostamento valido AND visited_distance[right_cell(c)] == 0 AND square[c] != 3 THEN
                Q.push(right_cell(c));
                visited_distance[right_cell(c)] ← visited_distance[c] + 1;

        IF bottom_cell(c) è uno spostamento valido AND visited_distance[bottom_cell(c)] == 0 AND square[c] != 3 THEN
                Q.push(bottom_cell(c));
                visited_distance[bottom_cell(c)] ← visited_distance[c] + 1;

        IF left_cell(c) è uno spostamento valido AND visited_distance[left_cell(c)] == 0 AND square[c] != 3 THEN
                Q.push(left_cell(c));
                visited_distance[left_cell(c)] ← visited_distance[c] + 1;

return max_of_min_distances;
```

Per trovare il numero minimo di spostamenti da eseguire per giungere ad una cella con 3 pannelli solari da una cella con 1 pannello solare indipendentemente dalla cella di partenza, abbiamo scelto

di effettuare una BFS multi sorgente sull'intera matrice N x N fornita in input. La BFS infatti è efficace per trovare il percorso più corto tra le celle.

La matrice N x N è implementata con array monodimensionale, in modo da memorizzare un singolo indice nella coda per ogni cella. Per spostarsi nelle direzioni valide (alto, destra, basso, sinistra) basterà rispettivamente effettuare:

- ➔ top_cell(c) : c - N
- ➔ right_cell(c) : c + 1
- ➔ bottom_cell(c) : c + N
- ➔ left_cell(c) : c - 1

La coda, utilizzata per effettuare la BFS, verrà inizializzata con gli indici delle celle con 3 pannelli solari. Questo consente di eseguire una singola BFS su tutta la matrice in contemporanea per tutte le celle con 3 pannelli.

Per tenere conto delle celle già visitate e le distanze per ciascuna cella, utilizziamo una singola matrice N x N chiamata visited_distance, anch'essa implementata con array monodimensionale per il motivo prima citato. Inizializziamo questa matrice a 0 ed aggiorniamo ogni volta la distanza in ogni iterazione della BFS sulla cella corrispondente. In questo modo se il valore di una cella in questa matrice è 0, significa che non è stata ancora visitata, mentre se è maggiore di 0 corrisponde alla distanza minima da una cella con 3 pannelli solari.

Inoltre per ogni iterazione della BFS:

- Se la cella visitata è una cella con 1 pannello solare e la visited_distance corrispondente è maggiore del massimo delle minime distanze precedentemente trovato, allora memorizziamo la distanza delle cella appena visitata;
- Per ciascuna cella in cui ci si può spostare nelle direzioni valide (alto, destra, basso, sinistra):
    - se lo spostamento è valido, cioè se l'indice restituito non va oltre ai bordi della matrice;
    - se la cella non è stata ancora visita, cioè se la visited_distance è uguale a 0;
    - se la cella non ha 3 pannelli solari, perché non vogliamo rimettere in coda una cella con 3 pannelli solari;

  Allora aggiungi alla coda la cella e aggiorna la visited_distance.

OSSERVAZIONE: La visited_distance alla fine conterrà tutte le distanze minime da una cella con 3 pannelli solari.

In seguito l'implementazione in C++.

```cpp
int algoritmo(ifstream *in)
{
    /* LETTURA DEL FILE */
    int N;   // valore N in input
    *in >> N; // lettura del valore N dal file

    if (N < 1 || N > 100) // requisito del problema
    {
        return -1;
    }

    char value_reader;                // variabile per la lettura del file
    int square[N * N];                // inizializzo l'array che conterrà i valori del square
    queue<int> Q;                     // inizializzo la coda per la multiple source BFS
    int visited_distance[N * N] = {0}; // inizializzo l'array che conterrà i minimi delle distanze da un 3
    int max_of_min_distances = -1;    // inizializzo la variabile che conterrà il massimo delle distanze minime da un 3.
                                      // Inizializzo a -1 sia per essere garantire che venga sostituito al primo valore e sia perchè in caso di errore ritornerà - 1

    for (int i = 0; i < N * N; i++)
    {
        *in >> value_reader;          // leggo il carattere dal file
        square[i] = value_reader - '0'; // leggo carattere per carattere traducendolo in int
        if (square[i] == 3)           // memorizzo le posizioni dei 3 nella matrice
        {
            Q.push(i); // inserisco i 3 nella coda
        }
    }

    /* multiple souse BFS*/
    while (!Q.empty()) // continua fino a quanto la coda è vuota cioè fino a quando tutte le celle saranno state visitate
    {

        int current_index = Q.front(); // prendo l'indice in prima posizione della coda
        Q.pop();                       // rimuovo l'elemente in prima posizione

        if (square[current_index] == 3 && visited_distance[current_index] > max_of_min_distances) // se la cella corrente è un 1 e la sua distanza dal 3 più vicino è maggiore del massimo delle distanze minime da un 3 allora
        {
            max_of_min_distances = visited_distance[current_index]; // assegnane il valore
        }

        int index_of_adjacent_cells[4] = {top_cell(current_index, N),
                                          right_cell(current_index, N),
                                          bottom_cell(current_index, N),
                                          left_cell(current_index, N)}; // array che memorizza gli indici delle celle adiacenti

        for (int i = 0; i < 4; i++) // per tutte le celle adiacenti
        {
            if (index_of_adjacent_cells[i] != -1 && visited_distance[index_of_adjacent_cells[i]] == 0 && square[index_of_adjacent_cells[i]] != 3) // se ha un indice valido (cioè se non siamo sforati dalla matrice) e se la cella non è stata ancora visitata e se la cella non è un 3
            {
                Q.push(index_of_adjacent_cells[i]);                                                 // inserisci l'indice nella coda
                visited_distance[index_of_adjacent_cells[i]] = visited_distance[current_index] + 1; // aggiorna la distanza della cella a quella attuale + 1
            }
        }
    }
    return max_of_min_distances; // ritorno il massimo delle distanze minime
}
```

Nel codice C++, usiamo l'implementazione della coda fornita dalla libreria Standard Template Library (STL).

Correttezza dell'algoritmo

Dimostriamo la correttezza per induzione:

1. **Base dell'induzione:** All'inizio, prima di qualsiasi iterazione del ciclo while, l'array visited_distance è inizializzato a 0 e la coda Q contiene tutte le posizioni delle celle con valore 3 con distanza iniziale 0. Quindi, la proprietà è vera all'inizio.
2. **Passo induttivo:** Supponiamo che la proprietà sia vera all'inizio di un'iterazione del ciclo while. Durante l'iterazione, si estrae una cella c dalla coda Q. Si esaminano tutte le celle adiacenti. Per ogni cella adiacente valida che non è stata visitata (distanza 0) e non contiene il valore 3, si aggiunge alla coda Q con una distanza incrementata di 1. Questo assicura che ogni cella venga visitata la prima volta con la distanza minima possibile dalla cella contenente il valore 3. Alla fine dell'iterazione, l'invariante rimane vera poiché tutte le celle visitate hanno registrato la distanza minima da una cella con valore 3.

Il ciclo termina quando tutte le celle raggiungibili sono state visitate. A questo punto, visited_distance contiene le distanze minime da qualsiasi cella con valore 3 a ogni altra cella. Il valore massimo tra queste distanze, per le celle contenenti 1, viene aggiornato nel risultato.

Complessità di tempo e spazio

Complessità di tempo = O(N^2)

- Inizializzare il quadrato richiede O(N^2) operazioni

- Inizializzare la coda richiede O(N^2) operazioni
- Nella BFS, ogni cella può essere inserita ed estratta nella coda al più una volta, quindi si ha O(N^2)

Dunque, la complessità totale è O(N^2).

Complessità spaziale = O(N^2)

- Il quadrato richiede spazio O(N^2)
- La coda richiede spazio O(N^2) nel caso pessimo (quando tutte le celle sono 3)
- L'array visited_distance richiede spazio O(N^2)

Dunque, la complessità totale è O(N^2).

Strutture dati utilizzate

- Array monodimensionale, per la rappresentazione della matrice in input e la matrice delle distanze;
- Coda, per la corretta esecuzione dell'algoritmo BFS.