Tutorato 29 Novembre

1 Esercizio

Testo

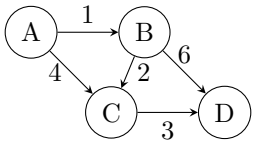
Considera un grafo diretto pesato con i seguenti archi e pesi:

- (A, B): 1
- (A, C): 4
- (B, C): 2
- (B, D): 6
- (C, D): 3

1. Rappresenta il grafo come una lista di adiacenza.
2. Usa l'algoritmo di Dijkstra per calcolare il cammino minimo da A a tutti gli altri nodi.

Soluzione

Disegno del grafo:



Lista di adiacenza:

$$A : \{(B, 1), (C, 4)\}$$
$$B : \{(C, 2), (D, 6)\}$$
$$C : \{(D, 3)\}$$
$$D : \{\}$$

Esecuzione dell'algoritmo di Dijkstra:

- Inizializziamo $\text{dist}[A] = 0$ e $\text{dist}[x] = \infty$ per ogni $x \neq A$.
- Passo 1: Nodo A. Aggiorniamo $\text{dist}[B] = 1$, $\text{dist}[C] = 4$.
- Passo 2: Nodo B. Aggiorniamo $\text{dist}[C] = \min(4, 1 + 2) = 3$, $\text{dist}[D] = 1 + 6 = 7$.
- Passo 3: Nodo C. Aggiorniamo $\text{dist}[D] = \min(7, 3 + 3) = 6$.
- Passo 4: Nodo D. Nessuna modifica.

Risultati finali:

$$\text{dist}[A] = 0, \quad \text{dist}[B] = 1, \quad \text{dist}[C] = 3, \quad \text{dist}[D] = 6$$

Esercizio: Breadth-First Search (BFS)

Problema: Dato un grafo $G = (V, E)$ rappresentato tramite lista di adiacenza, implementa l'algoritmo BFS per calcolare la distanza minima (numero di archi) di ogni nodo a partire da un nodo sorgente s. Usa BFS per determinare se il grafo è connesso.

Soluzione

Idea di base

BFS utilizza una struttura dati FIFO (coda) per visitare i nodi del grafo. Si parte dal nodo sorgente s e si visitano iterativamente tutti i suoi vicini, marcandoli come visitati e aggiornando la loro distanza dalla sorgente.

Pseudocodice

L'algoritmo è organizzato come segue:

Algorithm 1 Breadth-First Search (BFS)

Require: Grafo $G = (V, E)$ come lista di adiacenza, nodo sorgente s
Ensure: Distanze $d[v]$ per ogni nodo v da s, e verifica se il grafo è connesso

```
Inizializza d[v] ← ∞ per ogni v ∈ V
d[s] ← 0
Inizializza una coda Q e aggiungi s a Q
Inizializza visitati[v] ← falso per ogni v ∈ V
visitati[s] ← vero
while Q non è vuota do
    Estrai il nodo u da Q
    for ogni vicino v di u do
        if ¬visitati[v] then
            visitati[v] ← vero
            d[v] ← d[u] + 1
            Aggiungi v a Q
        end if
    end for
end while
Verifica se ∀v ∈ V, visitati[v] = vero. Se sì, il grafo è connesso.
```

Esempio di Input e Output

Input:

$$G = \{V = \{1, 2, 3, 4\}, E = \{(1, 2), (2, 3), (3, 4)\}\}, \quad s = 1$$

Output: Distanze: $d[1] = 0, d[2] = 1, d[3] = 2, d[4] = 3$ Il grafo è connesso.

Dimostrazione della Complessità

L'algoritmo visita ciascun nodo una sola volta e processa ogni arco una sola volta. Pertanto, la complessità temporale è $O(V + E)$, dove V è il numero di vertici e E il numero di archi.

Esercizio: Verifica Cicli in un Grafo Orientato (DFS)

Testo

Dato un grafo orientato $G = (V, E)$, implementa un algoritmo che utilizza DFS per verificare se il grafo contiene cicli.

Modellazione del problema

Possiamo rappresentare il problema in termini di stati durante la visita DFS:

- Ogni nodo può essere in uno dei seguenti stati:
  - **Non visitato** (bianco): il nodo non è ancora stato esplorato.
  - **In visita** (grigio): il nodo è in corso di esplorazione.
  - **Visitato** (nero): tutti i vicini del nodo sono stati visitati.
- Un ciclo esiste se, durante la visita di un nodo, troviamo un vicino che è già **in visita** (grigio).

L'obiettivo è determinare se esistono cicli nel grafo orientato.

Strategia

Utilizziamo DFS per:

- Esplorare il grafo e assegnare uno stato a ogni nodo.
- Verificare, durante l'esplorazione, se esiste un arco che punta a un nodo in stato "grigio".

Algoritmo

Passaggi principali:

1. Inizializza un array *color* per tracciare lo stato di ciascun nodo (*bianco*, *grigio*, *nero*).
2. Per ogni nodo non visitato, avvia DFS:
   - Durante la visita, marca il nodo come *grigio*.
   - Se trovi un vicino *grigio*, rileva un ciclo.
   - Dopo aver visitato tutti i vicini, marca il nodo come *nero*.
3. Se la DFS termina senza rilevare cicli, il grafo è aciclico.

Pseudocodice

```
def has_cycle(graph, n):
    color = [0] * n  # 0 = bianco, 1 = grigio, 2 = nero

    def dfs(node):
        color[node] = 1  # Grigio: in visita
        for neighbor in graph[node]:
            if color[neighbor] == 1:  # Nodo in stato grigio
                return True  # Rilevato un ciclo
            if color[neighbor] == 0:  # Nodo bianco
                if dfs(neighbor):
                    return True
        color[node] = 2  # Nero: visitato
        return False

    for i in range(n):
        if color[i] == 0:  # Nodo bianco
            if dfs(i):
                return True
    return False
```

Complessità

- **Tempo:** $O(V + E)$, dove V è il numero di nodi e E il numero di archi, poiché visitiamo ogni nodo e ogni arco una sola volta.
- **Spazio:** $O(V)$, per memorizzare lo stato di ciascun nodo e la ricorsione DFS.

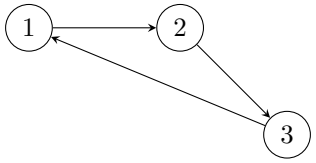
Esempio

Input:

$$G = \{V = \{1, 2, 3\}, E = \{(1, 2), (2, 3), (3, 1)\}\}$$

Output: Il grafo contiene un ciclo.

Rappresentazione del grafo



Spiegazione dell'esempio

- Durante la visita DFS: - Si parte dal nodo 1 e si esplora il nodo 2, marcandolo come *grigio*. - Dal nodo 2, si esplora il nodo 3, marcandolo come *grigio*. - Dal nodo 3, si tenta di esplorare il nodo 1, già in stato *grigio*. Questo indica la presenza di un ciclo.

Conclusione: il grafo contiene un ciclo.

Esercizio: Percorso minimo per un robot in una griglia

Testo

Un robot deve muoversi in una griglia 2D di dimensioni $n \times m$, rappresentata da una matrice C, dove $C[i][j]$ indica il costo di attraversamento della cella (i, j). Il robot parte da una cella iniziale (x_s, y_s) e deve raggiungere una cella finale (x_t, y_t).

Progetta un algoritmo per trovare il costo minimo per raggiungere la destinazione usando l'algoritmo di Dijkstra. L'algoritmo deve restituire anche il percorso minimo.

Modellazione del problema

Possiamo rappresentare la griglia come un **grafo orientato**:

- Ogni cella è un nodo.
- Ogni spostamento verso una cella adiacente (su, giù, sinistra, destra) è un arco con peso pari al costo della cella di destinazione.

L'obiettivo è trovare il costo minimo del cammino da (x_s, y_s) a (x_t, y_t).

Strategia

Utilizziamo l'algoritmo di **Dijkstra**:

- Manteniamo una coda con priorità per selezionare il nodo con il costo minimo non ancora elaborato.
- Aggiorniamo iterativamente le distanze minime delle celle adiacenti, tenendo traccia del percorso.
- Quando raggiungiamo la cella di destinazione, terminiamo.

Algoritmo

Passaggi principali:

1. Inizializza una matrice $dist$ con valori ∞, tranne $dist[x_s][y_s] = C[x_s][y_s]$.
2. Usa una coda con priorità per mantenere i nodi da esplorare, partendo da (x_s, y_s).
3. Per ogni cella estratta dalla coda:
   - Per ogni cella vicina, calcola il nuovo costo.
   - Se il nuovo costo è inferiore alla distanza attuale, aggiorna la distanza e il predecessore della cella, e aggiungila alla coda.
4. Ricostruisci il percorso minimo usando i predecessori.

Pseudocodice

```
def dijkstra(grid, start, target):
    n, m = len(grid), len(grid[0])
    dist = [[float('inf')] * m for _ in range(n)]
    prev = [[None] * m for _ in range(n)]
    dist[start[0]][start[1]] = grid[start[0]][start[1]]

    import heapq
    pq = [(dist[start[0]][start[1]], start)]  # (distanza, cella)

    directions = [(0,1), (1, 0), (0, -1), (-1, 0)] # dx, giu, sx, su

    while pq:
        d, (x, y) = heapq.heappop(pq)
        if (x, y) == target:
            break
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < n and 0 <= ny < m:  # Controlla limiti griglia
                new_cost = d + grid[nx][ny]
                if new_cost < dist[nx][ny]:
                    dist[nx][ny] = new_cost
                    prev[nx][ny] = (x, y)
                    heapq.heappush(pq, (new_cost, (nx, ny)))

    # Ricostruzione del percorso
    path = []
    x, y = target
    while (x, y) is not None:
        path.append((x, y))
        x, y = prev[x][y]
    return dist[target[0]][target[1]], path[::-1]
```

Complessità

- Tempo: $O(n \cdot m \cdot \log(n \cdot m))$, poiché ogni nodo viene estratto dalla coda con priorità al massimo una volta.
- Spazio: $O(n \cdot m)$, per memorizzare le distanze, i predecessori e la coda con priorità.

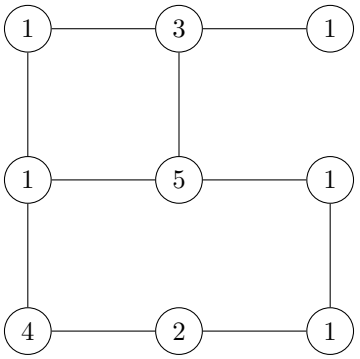
Esempio

Input:

$$C = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 5 & 1 \\ 4 & 2 & 1 \end{bmatrix}, \quad (x_s, y_s) = (0, 0), \quad (x_t, y_t) = (2, 2)$$

Output: Costo minimo: 7 Percorso: $(0,0) \to (1,0) \to (1,1) \to (1,2) \to (2,2)$.

Rappresentazione del grafo



Conclusione: Il robot percorre il cammino minimo con costo 7.

Esercizio: Rivalità tra wrestlers.

Testo

Ci sono due tipi di wrestler professionisti: i “babyfaces” (“buoni”) e gli “heels” (“cattivi”). Tra ogni coppia di wrestler, può esistere o meno una rivalità. Supponiamo di avere n wrestler e una lista di r coppie di wrestler per cui esistono rivalità.

1. Progetta un algoritmo che, in tempo $O(n + r)$, determini se è possibile designare alcuni wrestler come babyfaces e altri come heels, in modo tale che ogni rivalità sia tra un babyface e un heel. 2. Se tale soluzione è possibile, il tuo algoritmo deve produrre una soluzione.

Modellazione del problema

Possiamo rappresentare i wrestler e le loro rivalità come un **grafo non orientato**:

- Ogni nodo rappresenta un wrestler.
- Ogni arco rappresenta una rivalità tra due wrestler.

L’obiettivo è verificare se il grafo è **bipartito**:

- Un grafo è bipartito se possiamo dividere i nodi in due insiemi disgiunti (S_1 e S_2) in modo tale che ogni arco colleghi un nodo in S_1 con un nodo in S_2.
- In questo caso, S_1 rappresenta i wrestler ”babyfaces” e S_2 rappresenta gli ”heels”.

Strategia

Utilizziamo l’algoritmo di **Breadth-First Search (BFS)** per:

- Assegnare un colore a ciascun nodo (0 per i babyfaces, 1 per gli heels).
- Verificare se il grafo è bipartito controllando che nessun arco colleghi due nodi con lo stesso colore.

Algoritmo

Passaggi principali:

1. Creiamo un array *color* inizializzato a -1, che rappresenta i nodi non ancora colorati.
2. Per ogni nodo non visitato, inizializziamo la BFS:
    - Coloriamo il nodo iniziale con 0.

- Durante la BFS, assegniamo ai vicini il colore opposto rispetto al nodo corrente.
- Se troviamo un vicino già colorato con lo stesso colore del nodo corrente, il grafo non è bipartito.

3. Se la BFS termina senza conflitti, restituiamo i due insiemi.

Pseudocodice

```
def is_bipartite(graph, n):
    color = [-1] * n  # Inizializza colori
    for start in range(n):
        if color[start] == -1:  # Nodo non visitato
            color[start] = 0
            queue = [start]
            while queue:
                node = queue.pop(0)
                for neighbor in graph[node]:
                    if color[neighbor] == -1:  # Non colorato
                        color[neighbor] = 1 - color[node]
                        queue.append(neighbor)
                    elif color[neighbor] == color[node]:
# Conflitto
                        return False, None, None
    # Costruzione dei due insiemi
    babyfaces = [i for i in range(n) if color[i] == 0]
    heels = [i for i in range(n) if color[i] == 1]
    return True, babyfaces, heels
```

Complessità

- Tempo: $O(n + r)$, poiché visitiamo ogni nodo e ogni arco al massimo una volta.
- Spazio: $O(n)$, per memorizzare i colori e la coda.

Esempio

Supponiamo di avere $n = 4$ wrestler e $r = 4$ rivalità:

$$(1, 2), (2, 3), (3, 4), (4, 1)$$

1. Costruiamo il grafo:

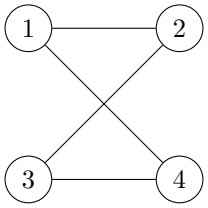
   Lista di adiacenza: $1 : [2, 4], \quad 2 : [1, 3], \quad 3 : [2, 4], \quad 4 : [1, 3]$

2. Applichiamo la BFS: - Iniziamo da 1: colore 0. - Colora 2 e 4 come 1. - Colora 3 come 0. - Nessun conflitto. Il grafo è bipartito.

3. Risultati:

$$\text{Babyfaces: } \{1, 3\}, \quad \text{Heels: } \{2, 4\}$$

Rappresentazione del grafo



Conclusione: È possibile designare i wrestler in modo che ogni rivalità sia tra un babyface e un heel.