Soluzioni ad esercizi su grafi

Esercizio 1

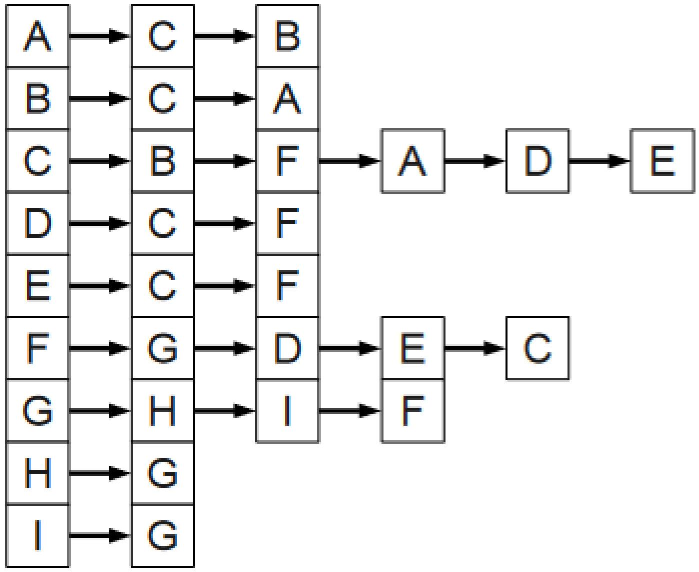
Supponiamo per assurdo che il cammino monotono $< v_1, v_2, \ldots, v_k >$ contenga un ciclo, cioè che esistano interi $1 \leq i < j \leq k$ tali che il cammino possa essere scritto come $< v_1, \ldots, v_i, \ldots, v_j, \ldots, v_k >$, e $< v_i, \ldots, v_j >$ sia un ciclo (ossia, $v_i = v_j$). Poiché il cammino iniziale è monotono, lo sarà anche il sottocammino $< v_i, \ldots, v_j >$. Quindi deve essere: $w(v_i) < w(v_{i+1}) < \ldots < w(v_j) = w(v_i)$ da cui si conclude $w(v_i) < w(v_i)$ il che è assurdo. Per quanto riguarda l'algoritmo che identifica se esiste un cammino aciclico che parte da s e termina in d, è possibile utilizzare una banale variazione dell'algoritmo di visita in profondità come segue.

```python
def cammino_monotono(G, s, d):
    # Inizializza tutti i nodi come non visitati
    for v in G.nodes():
        G.nodes[v]['mark'] = False
    # Avvia la ricerca ricorsiva da 's'
    return cammino_monotono_ric(G, s, d)

def cammino_monotono_ric(G, s, d):
    # Marca il nodo corrente come visitato
    G.nodes[s]['mark'] = True
    # Se il nodo corrente è la destinazione, restituisce True
    if s == d:
        print(d)
        return True
    else:
        # Itera su tutti i nodi adiacenti non visitati
        for x in G.adj[s]:
                if not G.nodes[x]['mark'] and G.nodes[s]['weight'] < G.nodes[x]['weight']:
                    if cammino_monotono_ric(G, x, d):
                        print(s)
                        return True
        # Se non esistono altri nodi adiacenti non visitati, restituisce False
        return False
```

Esercizio 2

La risposta è affermativa in entrambi i casi. Per la visita in profondità si può ad esempio partire da A oppure B. Per la visita in ampiezza si può partire ad esempio da F. In entrambi i casi (al di là del nodo di partenza, che è differente per la visita DFS e BFS) si può ottenere l'albero di mostrato nel testo utilizzando, ad esempio, la seguente rappresentazione con liste di adiacenza:



Esercizio 3

Se l'algoritmo di Dijkstra viene eseguito su un grafo in cui le lunghezze degli archi possono essere anche negative, possono verificarsi due situazioni:

- Nel caso di un grafo non orientato, l'algoritmo di Dijkstra non termina.
- Nel caso di un grafo orientato, l'algoritmo di Dijkstra termina, ma la complessità computazionale è maggiore, poiché un nodo può essere inserito più volte nella coda di priorità.

Esercizio 4

Soluzione: si modifica l'algoritmo di Dijkstra in modo tale che, quando si incontra un nodo v che è già stato visitato, si controlla se il suo peso è uguale al peso del nodo corrente u più il peso dell'arco (u, v). Se è così, si incrementa il contatore dei cammini minimi.

```python
def dijkstra_count_paths(G, s, r):
    '''
    G: grafo orientato con pesi positivi
    s: nodo sorgente
    r: nodo destinazione
    '''
    # Inizializziamo le distanze e il numero di cammini
    num_paths = [0] * len(G.nodes())
    distances = [float('inf')] * len(G.nodes())
```

```
distances[s] = 0
num_paths[s] = 1

queue.push(G.nodes())

while queue:
   u = queue.pop()
   for v in G.adj[u]:
      # modifica all'operazione di rilassamento
      if distances[v] > distances[u] + G.edges[u,v]['weight']:
         distances[v] = distances[u] + G.edges[u,v]['weight']
         # aggiorniamo usando il contatore del nodo u
         num_paths[v] = num_paths[u]
      elif distances[v] == distances[u] + G.edges[u,v]['weight']:
         num_paths[v] += num_paths[u]

return num_paths[r]
```