

Esercizi svolti di Progettazione di algoritmi
Facoltà di Ingegneria dell'informazione, informatica e statistica
Dipartimento di informatica
Anno Accademico 2022-2023

Esercitazione 6



SAPIENZA
UNIVERSITÀ DI ROMA



Es 1. Un albero minimo di copertura non contiene mai l'arco pesante di un qualche ciclo

Testo

Sia e un arco di peso massimo su un ciclo del grafo connesso $G = (V, E)$. Dimostrare che esiste un minimo albero di copertura di $G' = (V, E - \{e\})$ che è anche un minimo albero di copertura di G . Ovvero, esiste un minimo albero di copertura di G che non include e .



Es 1. Un albero minimo di copertura non contiene mai l'arco pesante di un qualche ciclo

Soluzione

Sia A un qualsiasi taglio che fa sì che alcuni nodi del ciclo si trovino su un lato del taglio e alcuni nodi del ciclo sull'altro. Per uno qualsiasi di questi tagli, sappiamo che l'arco e non è un arco leggero per questo taglio. Dal momento che tutti gli altri tagli non avranno l'arco e e che lo attraversa, non avremo nemmeno che l'arco e è leggero per nessuno di questi tagli.



Es. 2. Cammini minimi su grafo con archi gialli (di costo 0) e rossi (di costo 1)

Testo

Scrivere una funzione che calcola il cammino minimo in un grafo con archi gialli (costo 0) e rossi (costo 1). La funzione prende in input il grafo e il nodo di partenza per il cammino minimo, e restituisce il costo minimo del cammino dal nodo di partenza a ogni altro nodo.



Es. 2. Cammini minimi su grafo con archi gialli (di costo 0) e rossi (di costo 1)

Idea

Poiché gli archi gialli hanno costo zero, è sempre conveniente scegliere un arco giallo piuttosto che un arco rosso. Pertanto, un cammino minimo in questo tipo di grafo sarà sempre costituito esclusivamente da archi gialli.

Se non esiste un cammino solo con archi gialli, allora il cammino minimo dovrà includere anche alcuni archi rossi. In questo caso, l'obiettivo sarà quello di minimizzare il numero di archi rossi necessari per raggiungere la destinazione.



Es. 2. Cammini minimi su grafo con archi gialli (di costo 0) e rossi (di costo 1)

Soluzione

```
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    heap = [(0, start)]
    while heap:
        (distance, current_node) = heapq.heappop(heap)
        if distance > distances[current_node]:
            continue
        for neighbor, cost in graph[current_node].items():
            if cost == 0:
                new_distance = distance
            else: # Se l'arco è rosso (costo 1)
                new_distance = distance + cost
            if new_distance < distances[neighbor]:
                distances[neighbor] = new_distance
                heapq.heappush(heap, (new_distance, neighbor))
    return distances
```



Es. 2. Cammini minimi su grafo con archi gialli (di costo 0) e rossi (di costo 1)

Esecuzione

```
graph = {  
  'A': {'B': 1, 'C': 1, 'D': 0},  
  'B': {'A': 1, 'C': 0},  
  'C': {'A': 1, 'B': 0},  
  'D': {'A': 0}  
}  
dijkstra(graph, 'C') -> {'A': 1, 'B': 0, 'C': 0, 'D': 1}
```



Es. 3. Algoritmo per determinare se un grafo diretto è semi-connesso

Testo

Un grafo diretto $G=(V,E)$ è semi-connesso se $u \rightsquigarrow v$ implica che G contiene al massimo un percorso semplice da u a v per tutti i vertici $u,v \in V$. Fornite un algoritmo efficiente per determinare se un grafo diretto è semi-connesso o meno.



Es. 3. Algoritmo per determinare se un grafo diretto è semi-connesso

Idea

Questo può essere fatto in tempo $O(|V| |E|)$. Per farlo, si esegue prima un ordinamento topologico dei vertici. Poi, per ogni vertice, si avrà un elenco dei suoi antenati in-degree 0. Si calcolano questi elenchi per ogni nodo nell'ordine che parte da quelli precedenti dal punto di vista topologico.

Se un vertice ha lo stesso grado 0 nelle liste di due dei suoi genitori immediati, sappiamo che il grafo non è singolarmente connesso. Se invece, a ogni passo, tutti i genitori hanno come antenati insiemi disgiunti di vertici di grado 0, il grafo è singolarmente connesso.

Poiché, per ogni vertice, la quantità di tempo richiesta è limitata dal numero di vertici moltiplicato per l'in-degree di quel particolare vertice, il tempo di esecuzione totale è limitato da $O(|V| |E|)$.



Es. 3. Algoritmo per determinare se un grafo diretto è semi-connesso

Soluzione

```
from collections import defaultdict

def is_singly_connected(graph):
    # 1. Ordinamento topologico
    topo_order = topological_sort(graph)
    # 2. Dizionario degli antenati in-degree 0
    ancestors = defaultdict(set)
    # 3. Calcolo degli antenati in-degree 0 per ogni vertice
    visited = set()
    for v in topo_order:
        ancestors_v = set()
        for p in graph[v]["parents"]:
            if p not in visited:
                visited |= set(find_zero_indegree_ancestors(p, graph, visited, ancestors))
            ancestors_v |= ancestors[p]
        if any(ancestors_v & ancestors[p] for p in graph[v]["parents"]):
            return False # Grafo non singolarmente connesso
        visited.add(v)
        ancestors[v] = ancestors_v
    # 4. Controllo se tutti i vertici sono stati visitati
    return len(visited) == len(graph)
```

```
def topological_sort(graph):
    in_degree = {v: 0 for v in graph}
    for v in graph:
        for p in graph[v]["parents"]:
            in_degree[p] += 1
    queue = [v for v in graph if in_degree[v] == 0]
    topo_order = []
    while queue:
        v = queue.pop(0)
        topo_order.append(v)
        for c in graph[v]["children"]:
            in_degree[c] -= 1
            if in_degree[c] == 0:
                queue.append(c)
    return topo_order
```

```
def find_zero_indegree_ancestors(v, graph, visited, ancestors):
    if v in visited:
        return []
    visited.add(v)
    ancestors_v = set()
    for p in graph[v]["parents"]:
        visited |= set(find_zero_indegree_ancestors(p, graph, visited, ancestors))
        ancestors_v |= ancestors[p]
    if graph[v]["in_degree"] == 0:
        ancestors_v.add(v)
    ancestors[v] = ancestors_v
    return ancestors_v
```



Es. 3. Algoritmo per determinare se un grafo diretto è semi-connesso

Esecuzione

```
graph = {  
  "A": {"parents": [], "children": ["B", "C"], "in_degree": 0},  
  "B": {"parents": ["A"], "children": ["D"], "in_degree": 1},  
  "C": {"parents": ["A"], "children": ["D"], "in_degree": 1},  
  "D": {"parents": ["B", "C"], "children": [], "in_degree": 2}  
}  
is_singly_connected(graph) -> False
```



Es. 4. Selezionare il k -esimo elemento nell'unione di due vettori ordinati

Testo

Date due matrici ordinate di dimensione m e n rispettivamente, il compito è quello di trovare l'elemento che si trova nella posizione k della matrice ordinata finale.



Es. 4. Selezionare il k -esimo elemento nell'unione di due vettori ordinati

Idea $O(m+n)$

Dato che abbiamo due array ordinati, possiamo usare la tecnica dell'unione per ottenere l'array finale unito. A partire da questo, si passa semplicemente all'indice k .



Es. 4. Selezionare il k-esimo elemento nell'unione di due vettori ordinati

Soluzione $O(m+n)$

```
def kth(arr1, arr2, len_arr1, len_arr2, k):  
    sorted_arr = [0] * (len_arr1 + len_arr2)  
    i, j, d = 0, 0, 0  
    while (i < len_arr1 and j < len_arr2):  
        if (arr1[i] < arr2[j]):  
            sorted_arr[d] = arr1[i]  
            i += 1  
        else:  
            sorted_arr[d] = arr2[j]  
            j += 1  
        d += 1  
    ...
```

```
...  
while (i < len_arr1):  
    sorted_arr[d] = arr1[i]  
    d += 1  
    i += 1  
while (j < len_arr2):  
    sorted_arr[d] = arr2[j]  
    d += 1  
    j += 1  
return sorted_arr[k - 1]
```



Es. 4. Selezionare il k-esimo elemento nell'unione di due vettori ordinati

Esecuzione $O(m+n)$

arr1 = [2, 3, 6, 7, 9]

arr2 = [1, 4, 8, 10]

len_arr1 = 5

len_arr2 = 4

k = 5

kth(arr1, arr2, len_arr1, len_arr2, k) -> 6



Es. 4. Selezionare il k-esimo elemento nell'unione di due vettori ordinati

Idea $O(\log m + \log n)$

Se il metodo precedente funziona, possiamo rendere il nostro algoritmo più efficiente? La risposta è sì. Utilizzando un approccio divide et impera, simile a quello usato nella ricerca binaria, possiamo cercare di trovare il k-esimo elemento in modo più efficiente.

Confrontiamo gli elementi centrali degli array `arr1` e `arr2`, chiamiamo questi indici rispettivamente `mid1` e `mid2`. Supponiamo che `arr1[mid1] > arr2[mid2]`, quindi chiaramente gli elementi dopo `mid2` non possono essere l'elemento richiesto. Impostiamo che l'ultimo elemento di `arr2` sia `arr2[mid2]`.

In questo modo, si definisce un nuovo sottoproblema con la metà delle dimensioni di uno degli array.



Es. 4. Selezionare il k-esimo elemento nell'unione di due vettori ordinati

Soluzione $O(\log m + \log n)$

```
def kth(arr1, arr2, len_arr1, len_arr2, k):  
    if len_arr1 == 1 or len_arr2 == 1:  
        if len_arr2 == 1:  
            arr2, arr1 = arr1, arr2  
            len_arr2 = len_arr1  
        if k == 1:  
            return min(arr1[0], arr2[0])  
        elif k == len_arr2 + 1:  
            return max(arr1[0], arr2[0])  
        else:  
            if arr2[k - 1] < arr1[0]:  
                return arr2[k - 1]  
            else:  
                return max(arr1[0], arr2[k - 2])  
    ...
```

```
    ...  
    mid1 = (len_arr1 - 1)//2  
    mid2 = (len_arr2 - 1)//2  
    if mid1+mid2+1 < k:  
        if arr1[mid1] < arr2[mid2]:  
            return kth(arr1[mid1 + 1:], arr2, len_arr1 - mid1 - 1, len_arr2, k - mid1 - 1)  
        else:  
            return kth(arr1, arr2[mid2 + 1:], len_arr1, len_arr2 - mid2 - 1, k - mid2 - 1)  
    else:  
        if arr1[mid1] < arr2[mid2]:  
            return kth(arr1, arr2[:mid2 + 1], len_arr1, mid2 + 1, k)  
        else:  
            return kth(arr1[:mid1 + 1], arr2, mid1 + 1, len_arr2, k)
```



Es. 4. Selezionare il k-esimo elemento nell'unione di due vettori ordinati

Esecuzione $O(\log m + \log n)$

arr1 = [2, 3, 6, 7, 9]

arr2 = [1, 4, 8, 10]

len_arr1 = 5

len_arr2 = 4

k = 5

kth(arr1, arr2, len_arr1, len_arr2, k) -> 6