ale-cci

Modelli Algoritmi per il Supporto alle Decisioni

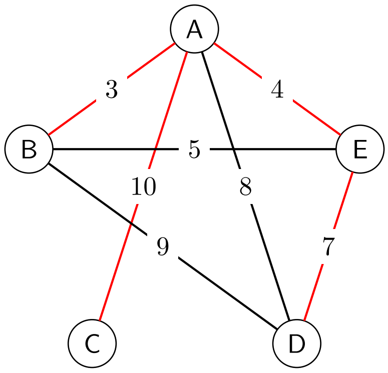
April 1, 2020

Introduction

TODO

Minimum Spanning Tree

Algoritmo di Kruskal (Greedy)




```
from utils import num_vertices

def kruskal(edges: list, N: int) -> list:
    connected = set()
    mst = []

    edges = sorted(graph)
    for edge in edges:
        weight, lhs, rhs = edge

        # Two nodes already connected
        if lhs in connected and rhs in connected:
            continue

        mst.append(edge)
        connected.update({lhs, rhs})

        if len(mst) == N:
            break

    return mst


if __name__ == '__main__':
    graph = [(10, 'A', 'C'), (8, 'A', 'D'),
             (7, 'D', 'E'), (4, 'A', 'E'),
             (3, 'B', 'A'), (9, 'B', 'D'), (5, 'B', 'E')]
    N = num_vertices(edges=graph)

    print(kruskal(graph, N))
```

Correttezza algoritmo di Kruskal

Supponiamo per assurdo che esista un' diverso MST $T' = (V, E_{T'})$ di peso inferiore a $T = (V, E_T)$, quello restituito dall'algoritmo greedy.

Siccome i due alberi hanno costo diverso, differiscono di almeno un' arco. Indichiamo con e_h l'arco a peso minore appartenente a $\{E_T - E_{T'}\}$. Dato che T' è un MST, esiste un ciclo C in $\{e_h\} \cup E_{T'}$ contenente l'arco e_h. Siccome anche T è un albero, quindi non ha cicli, allora $C \cap E_T \neq \emptyset$. Chiamiamo e_r l'arco a peso minore appartenente a $C \cap \{E_T - E_{T'}\}$. Necessariamente $w_{e_r} \leq w_{e_h}$, altrimenti l'algoritmo greedy applicato a T avrebbe selezionato prima e_h al posto di e_r. Sostituendo in T' l'arco e_r con e_h ottengo un nuovo albero di peso inferiore.

Questo va contro l'ipotesi T' è l'albero di supporto a peso minore.

Analisi complessità

$O(E \cdot \log(E))$, dovuta all'ordinamento degli archi in ordine di peso. Il controllo dell'esistenza di cicli è effettuato in $O(1)$.

Foresta di supporto

Viene chiamata foresta di supporto di un grafo G un grafo parziale $F = (V, E_F)$ privo di cicli. In particolare, un albero di supporto è una foresta con una sola componente connessa.

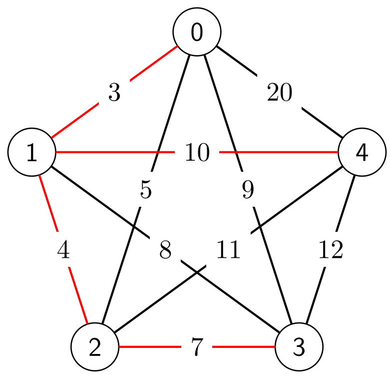
Teorema

Indichiamo con $(V_1, E_1), \ldots, (V_k, E_k)$ le componenti connesse di una foresta di supporto $F = (V, E_F)$ del grafo G. Sia inoltre (u, v) un arco a peso minimo tra quelli con un unico estremo in V_1. Allora esiste almeno un albero di supporto a peso minimo appartenente a $\bigcup_{i=1}^{k} E_i$, che contiene (v, v).

Dimostrazione

Per assurdo, l'albero a peso minimo non contiene (u, v). Ma aggiungendo (u, v) a tale albero si forma un ciclo contenente un altro arco (u', v') con un solo estremo in V_1. Se si toglie questo arco e si lascia (u, v) si ottiene un albero di peso minore, contraddicendo l'ipotesi.

Algoritmo MST-1




```python
def mst_1(w: list) -> list:
    V = set(range(len(w)))  # {0, 1, 2, 3, 4}
    c = [0] * len(V)        # [0, 0, 0, 0, 0]
    U = {0}
    mst = []

    while U != V:
        weight, u = min((w[v][c[v]], v) for v in V - U)
        U.add(u)
        mst.append((u, c[u]))

        for v in V - U:
            if w[v][u] < w[v][c[v]]:
                c[v] = u

    return mst
```

```python
if __name__ == '__main__':
    w = [[ 0,  3,  5,  9, 20],
         [ 3,  0,  4,  8, 10],
         [ 5,  4,  0,  7, 11],
         [ 9,  8,  7,  0, 12],
         [20, 10, 11, 12,  0]]

    print(mst_1(w))
```

Correttezza MST-1

Inizialmente abbiamo la foresta con $V_1 \equiv U = \{v_1\}, V_i = \{v_i\} \quad i = 2 \dots n$, con tutti gli $E_i = \emptyset$.

Alla prima iterazione si inserisce l'arco (V_i, v_{j_1}), $j_1 \neq 1$, a peso minimo tra quelli con un solo estremo in $U \equiv V_1$ e quindi, per il teorema visto al paragrafo della foresta di supporto, tale arco farà parte dell'albero di supporto a peso minimo tra tutti i possibili alberi di supporto.

Con l'aggiunta di questo arco, le due componenti connesse (V_1, E_1) e (V_{j_1}, E_{j_1}) si fondono in un'unica componente connessa con nodi $U = \{v_i, v_{j_1}\}$ e l'insieme di archi $E_T = \{(v_1, v_{j_1})\}$, mentre le altre componenti connesse non cambiano. Abbiamo cioè che le componenti connesse

$$(U, E_T), \quad (V_i, \emptyset) i \in \{2, \dots n\} - \{j_1\}$$

Alla seconda iterazione andiamo a selezionare il nodo v_{j_2} e il relativo arco $(v_{j_2}, c(v_{j_2}))$ con il peso minimo tra tutti quelli con un solo estremo in U. In base al teorema, l'arco $(v_{j_2}, c(v_{j_2}))$ farà parte di un albero di supporto a peso minimo tra tutti quelli che contengono l'unione di tutti gli archi delle componenti connesse, che si riduce ad E_T.

Effettuiamo lo stesso ragionamento per tutti gli n ottenendo l'albero di supporto a peso minimo.

Complessità dell'algoritmo

Il numero di operazioni richiesto è pari a $O(V^2)$ dovuta al ciclo eseguito V volte $(O(V))$ e la ricerca del minimo in tempo lineare.

Confronto con algoritmo di Kruskal

Anche se risulta essere peggiore rispetto all'algoritmo di greedy Kruskal, è possibile dimostrare che, in caso di grafi densi ha una complessità ottima. Infatti per tali grafi non possiamo aspettarci di fare meglio di $O(V^2)$: la sola operazione di lettura dei pesi degli archi richiede $O(V^2)$.

Algoritmo MST-2

```python
import utils

def mst_2(edges):
    N = utils.num_vertices(edges=edges)
    mst = set()
    component = list(range(N))

    while len(set(component)) > 1:
        minimum = {set_name: None for set_name in set(component)}
        shortest = {set_name: None for set_name in set(component)}

        for weight, u, v in edges:
            s_u = component[u]
            s_v = component[v]

            if s_u == s_v:
                continue
```

```
            if minimum[s_u] is None or weight < minimum[s_u]:
                shortest[s_u] = (u, v)
                minimum[s_u] = weight

            if minimum[s_v] is None or weight < minimum[s_v]:
                shortest[s_v] = (u, v)
                minimum[s_v] = weight

        mst.update(shortest.values())

        # Find connected components with union-disjoint set
        for u, v in shortest.values():
            utils.union_set(component, u, v)
        component = [utils.get_set(component, i) for i in component]

    return mst


if __name__ == '__main__':
    edges = [(1, 0, 1), (2, 0, 2), (4, 0, 3), (3, 1, 2), (4, 1, 3), (1, 2, 3)]
    N = utils.num_vertices(edges=edges)

    print(mst_2(edges))
```

Dimostrazione correttezza

Lasciata per esercizio, si basa sul teorema della foresta. *"Tutti gli archi shortest aggiunti ad una certa iterazione, sono tutti archi che fanno parte ad un albero di supporto ottimo, tra tutti i possibili alberi di supporto."*

Complessità algoritmo

$O(E \cdot \log_2(V))$ derivato dal costo dell'iterazione su tutti gli archi $O(E)$, eseguita un numero massimo di $\log(|V|)$ volte.

Inizialmente il numero di componenti connesse è pari al numero di nodi. Sicuramente ad ogni iterazione, il numero di componenti connesse viene almeno dimezzato. Per cui, il primo ciclo viene eseguito al più $\log(|V|)$ volte.

Per grafi densi con $|E| = O(|V|^2)$ questa complessità è peggiore di quella di MST-1, ma se il numero di archi scende sotto l'ordine $O(|V|^2/log(|V|))$ l'algoritmo MST-2 ha prestazioni migliori.

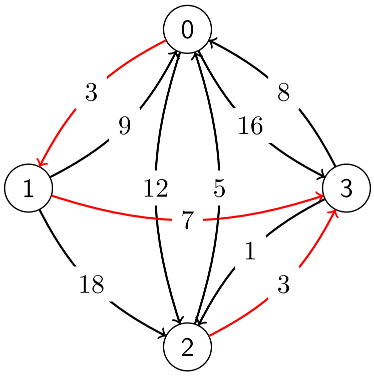
Note

Questi tre algoritmi appena visti sono tutti e tre algoritmi costruttivi, senza revisione delle decisioni passate.

Shortest Path

Algoritmo di Dijkstra

Applicabile soltanto nel caso in cui i pesi degli archi siano sempre non negativi.




```python
def dijkstra(adj_matrix: list, source: int):
    N = len(adj_matrix)
    W = {source}
    V = set(range(N))

    dist = [0 if i == source else adj_matrix[source][i]
            for i in range(N)]
    parent = [source] * N
    parent[source] = None

    while W != V:
        _, x = min((dist[i], i) for i in V - W if dist[i] is not None)
        W.add(x)

        for y in V - W:
            if adj_matrix[x][y] is None:
                continue

            if dist[y] is None or dist[y] > dist[x] + adj_matrix[x][y]:
                parent[y] = x
                dist[y] = dist[x] + adj_matrix[x][y]

    return parent


if __name__ == '__main__':
    w = [[None,    3,   12,   16],
         [   9, None,   18,    7],
         [   5, None, None,    3],
         [   8, None,    1, None]]

    print(dijkstra(w, 0))
```

Dimostrazione Correttezza

La dimostrazione viene effettuata per ragionamento induttivo sui due seguenti punti:

1. $\forall y \in V$, il valore `dist[y]` rappresenta ad ogni iterazione la lunghezza del cammino minimo da `source` a `y`, passando solo attraverso i nodi contenuti in W. `dist[y] == None` indica

che il nodo non è raggiungibile passando solamente attraverso i nodi di W. In `parent[y]` è memorizzato il nodo che precede immediatamente y in tale cammino. `parent[source] == None` indica che `source` non è preceduto da altri nodi.

2. quando il nodo x viene aggiunto a W, il valore `dist[x]` rappresenta la distanza minima tra `source` e x. Il cammino minimo è ricostruibile procedendo a ritroso partendo da `parent[x]`.

Per $n = i = 0$ sono ovviamente vere entrambe.

Dimostrazione punto 1

Per il passo $n = i + 1$, quando analizziamo y, abbiamo due casi possibili: x non è contenuto all'interno del cammino minimo, per cui, per ipotesi induttiva, la distanza minima è `dist[y]`; oppure x precede immediatamente y nel cammino minimo, quindi in tal caso deve avere distanza `dist[x] + weight[x][y]`.

Consideriamo per assurdo che x non preceda immediatamente y, allora $\exists t \in W$ nel cammino minimo tra x ed y, il cammino da s a t, per ipotesi induttiva, ha almeno una lunghezza che è $\geq$ `dist[t]`, con relativo cammino minimo che non comprende x. Il cammino da `source` a y, passante per x è quindi sostituibile da un' altro cammino di lunghezza inferiore, non passante per x, ma questo ricadrebbe nel primo caso, dove x non è contenuto all'interno del cammino minimo.

Dimostrazione punto 2

per assurdo, ipotizziamo che esista un cammino da s a x di lunghezza inferiore a $\rho(x)$ che passi per nodi $\notin W$, chiamato z il primo di tali nodi. Chiamato $L(s,x)$, la lunghezza del cammino passante per z, abbiamo che per ipotesi per assurdo: $L(s,z) + L(z,x) < \rho(x)$. Osserviamo che $L(s,z) \geq \rho(z)$ perché tra s e z tutti i nodi sono contenuti in W, e $\rho(z)$ è la lunghezza minima dei cammini da s a z non passanti per nodi al di fuori di W. $L(z,x) \geq 0$ perché per ipotesi tutte le distanze sono non negative, quindi abbiamo:

$$\rho(z) \leq L(s,z) + L(z,x) = L(s,x) < \rho(x)$$

Siamo giunti ad un assurdo perché, da come è definito l'algoritmo $x = \arg\min_{y \notin W} \{\rho(y)\}$, e di conseguenza $\rho(x) < \rho(z)$.

Complessità

Il numero di operazioni richiesto è $O(V^2)$, dovuta ad il ciclo principale di complessità $O(|V|)$ ed il calcolo del minimo ed il ciclo sui nodi adiacenti, entrambi di complessità $O(|V|)$.

Operazione di Triangolazione

Data una matrice nxn di distanze R, per un dato $j \in \{1 \dots n\}$, chiamiamo "operazione di triangolazione" il seguente aggiornamento:

$$R_{jk} = \min\{R_{ik}, R_{ij} + R_{jk}\} \quad \forall i,k \in \{1 \dots n\} - \{j\}$$

Algoritmo di Floyd-Warshall

Applicabile solo se nel grafo non sono presenti cicli di lunghezza negativa. Restituisce la lunghezza dei cammini minimi tra ogni coppia di nodi.

```
infty = 10**20

def floyd_warshall(w):
    N = len(w)
    R = [weights[:] for weights in w]
    E = [[None]*N for _ in range(N)]
```

```
    for i in range(N):
        for j in range(N):
            if R[i][j] is None:
                R[i][j] = +infty

    for j in range(N):
        for i in set(range(N)) - {j}:
            for k in set(range(N)) - {j}:
                if R[i][k] > R[i][j] + R[j][k]:
                    E[i][k] = j
                    R[i][k] = R[i][j] + R[j][k]

        if any(R[i][i] < 0 for i in range(N)):
            break

    return R, E


if __name__ == '__main__':
    w = [[None,    3,   12,   16],
         [   9, None,   18,    7],
         [   5, None, None,    3],
         [   8, None,    1, None]]

    R, E = floyd_warshall(w)
```

Note

In caso di distanze $d_{ij} > 0$, la condizione di arresto $R_{ij} < 0$ non si potrà mai verificare, e si dimostra che i valori di R_{ij} danno la lunghezza del cammino minimo da i a j per ogni $i \neq j$, mentre le etichette E_{ij} consentono di ricostruire tali cammini minimi.

In caso di distanze negative, se non interviene la condizione di arresto $R_{ii} < 0$, allora anche qui gli R_{ij} danno la lunghezza del cammino minimo da i a j.

Se invece ad una certa iterazione si verifica la condizione $R_{ii} < 0$, indica la presenza di un ciclo a costo negativo nel grafo. In tal caso, anche ignorando la condizione di arresto $R_{ii} < 0$, non possiamo garantire che al memento della terminazione con $j = n$ gli R_{ij} diano la lunghezza del cammino minimo da i a j.

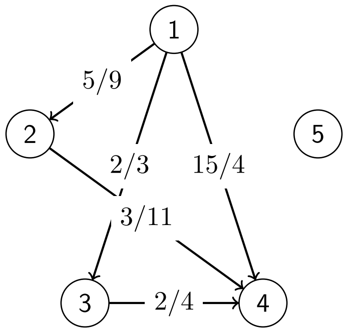
Complessità

È facile osservare che la complessità per questo algoritmo è $O(|V|^3)$, dovuta ai tre cicli nidificati, ognuno di complessità $O(|V|)$.

Note

Entrambi quest algoritmi, rientrano nella categoria di raffinamento locale. Infatti in entrambi i casi, data una coppia di nodi, si parte da una soluzione ammissibile, e ad ogni iterazione, tale cammino viene aggiornato nel caso se ne trovi uno di lunghezza inferiore.

Flusso a costo minimo



Classificazione dei nodi

Nei problemi di flusso a costo minimo, i nodi sono divisi in tre categorie, in base a b_i:

1. Nodi sorgente: $b_i > 0$, in essi viene realizzato il prodotto.
2. Nodi di transito: $b_i = 0$, il prodotto transita, senza variazioni
3. Nodi destinazione: $b_i < 0$, dove il prodotto viene consumato.

La proprietà $\sum_i^n b_i = 0$, quando non risulta valida, è forzabile aggiungendo un fittizio con $b_{n+1} = -\sum_i^n b_i$, collegato a tutti i nodi sorgente, attraverso archi di costo 0 e capacità $+\infty$.

Nel caso di archi con capacità illimitata

Prendiamo come esempio la rete $G = (V, A)$ in figura 1.

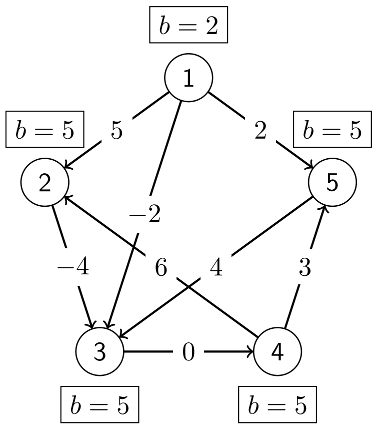


Figure 1: b_i sono indicati vicino al nodo

Contents

Introduction **1**

Minimum Spanning Tree **2**
Algoritmo di Kruskal (Greedy) 2
Foresta di supporto 3
Algoritmo MST-1 3
Algoritmo MST-2 4
Note 5

Shortest Path **6**
Algoritmo di Dijkstra 6
Operazione di Triangolazione 7
Algoritmo di Floyd-Warshall 7
Note 8

Flusso a costo minimo **9**
Classificazione dei nodi 9