ale-cci

Modelli Algoritmi per il Supporto alle Decisioni

April 27, 2020

Contents

Introduction **1**
Grafi bipartiti . . . 1

Minimum Spanning Tree **2**
Algoritmo di Kruskal (Greedy) . . . 2
Foresta di supporto . . . 3
Algoritmo di Prim (MST-1) . . . 4
Algoritmo Borůkova (MST-2) . . . 5
Note . . . 6

Shortest Path **7**
Algoritmo di Dijkstra . . . 7
Operazione di Triangolazione . . . 9
Algoritmo di Floyd-Warshall . . . 9
Note . . . 10

Flusso a costo minimo **11**
Grafo di flusso . . . 11
Coefficienti di costo ridotto . . . 12
Cambio di Base . . . 13
Algoritmo del simplesso su rete . . . 13
Algoritmo del simplesso con capacità limitate . . . 13

Flusso a costo massimo **16**
Procedura di soluzione . . . 16
Algoritmo di Ford-Fulkerson . . . 17

Matching **20**
Definizione di Matching . . . 20
Matching a peso massimo . . . 20
Assegnamento . . . 21
Algoritmo Ungherese . . . 21

Branch and bound **25**
Branching . . . 25
Algoritmo branch and bound . . . 26
Knapsack Problem . . . 26

Programmazione Dinamica **28**
Principio di ottimalità . . . 28
Problema dello zaino . . . 28

Classificazione di complessità **30**
Problemi di ottimizzazione . . . 30

Introduction

Gli algoritmi trattati rientrano in tre categorie: costruttivi, di raffinamento locale, e di enumerazione.

Con $G = (V, E)$ indichiamo un grafo generico, di nodi V (*Vertices*) e di archi E (*Edges*).

Due strutture dati frequentemente utilizzate per lavorare con i grafi sono la matrice di incidenza e la lista di adiacenza.

Si definisce circuito Hamiltoniano un ciclo elementare che tocca tutti i nodi del grafo.

Un grafo è detto completo se per ogni coppia di nodi (i, j) esiste un arco che li collega.

Grafi bipartiti

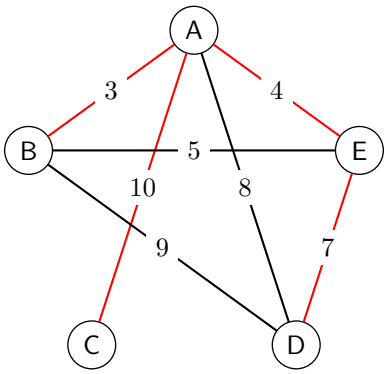
Un grafo si dice bipartito se l'insieme di nodi V può essere partizionato in due sottoinsiemi V_1 e V_2 tali che $V_1 \cap V_2 = \emptyset$.

Per determinare se un grafo è bipartito si utilizza una BFS, marcando inizialmente il nodo sorgente si marcano in alternanza i nodi che appartengono ai gruppi V_2 e V_1.
Nel caso in cui risulti un nodo appartenente a più di gruppo, il grafo non è bipartito.

```
import collections


def is_biparted(graph):
    queue = collections.deque([('a', 0)])
    visited = set()
    color_of = {}

    biparted = True
    while biparted and len(queue):
        parent, color = queue.pop()
        visited.add(parent)
        color_of[parent] = color

        for neightbour in graph[parent]:
            if neightbour not in visited:
                queue.append((neightbour, 1-color))
            elif color_of[neightbour] == color:
                biparted = False

    return biparted


if __name__ == '__main__':
    is_biparted({
        'a': ['b', 'c'],
        'b': ['a'],
        'c': ['a']
    })
```

Minimum Spanning Tree

Algoritmo di Kruskal (Greedy)




```python
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.

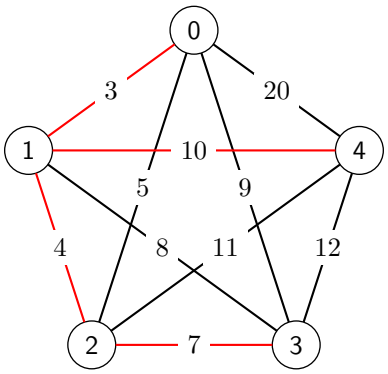
Teorema

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 di Prim (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

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 Borůkova (MST-2)

```
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.

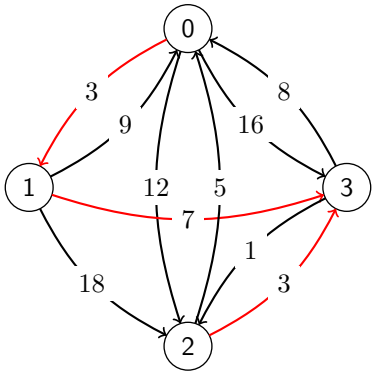
Note

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

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

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

Grafo di flusso

Dato un grafo orientato e connesso $G = (V, A)$, viene indicato con c_{ij} il costo di trasporto su un arco, d_{ij} la capacità massima di trasporto e con b_i la quantità di prodotto che ogni nodo è in grado di produrre.

I nodi si possono distinguere in tre gruppi in base al valore di b_i

1. Nodi sorgente $b_i > 0$, viene realizzato il prodotto che circola internamente alla rete
2. Nodi di transito: $b_i = 0$, dove il prodotto rimane invariato e si limita a transitare.
3. Nodi destinazione: $b_i < 0$, viene consumato il prodotto della rete

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 $+\infty$.

Soluzione del problema con archi di capacità illimitata

Il problema di flusso a costo minimo consiste nel far giungere il prodotto dai nodi sorgente ai nodi destinazione minimizzando il costo di trasporto.

Una base di un grafo di flusso è un suo albero di supporto. Ad ogni base è associata una soluzione di base, ottenuta ponendo a 0 il flusso [1] di ogni arco non contenuto in essa.

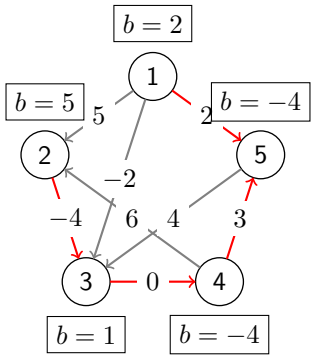


Figure 1: b_i sono indicati vicino al nodo

Prendiamo come esempio la base $B_0 = \{(1,5),(2,3),(3,4),(4,5)\}$ del grafo $G = (V, A)$ rappresentata in figura 1. Ponendo nullo il flusso degli archi che non appartenenti a B_0, otteniamo che i flussi della soluzione di base sono:

$$(1,\ 5) \to 2 \qquad (2,\ 3) \to 5$$
$$(3,\ 4) \to 6 \qquad (4,\ 5) \to 2$$

Se tutti i flussi degli archi in base è non negativo, allora la soluzione si dice ammissibile. Inoltre se i flussi sono tutti strettamente maggiori di 0, allora si parla di soluzione non degenere.

[1] quantità di prodotto inviata

Il costo totale del trasporto si calcola sommando per ogni arco della soluzione il prodotto tra quantità di flusso e costo di trasporto. Nel caso di esempio:

$$c_{15}x_{15} + c_{23}x_{23} + c_{34}x_{34} + c_{45}x_{45} = 2 \cdot 2 - 4 \cdot 5 + 0 \cdot 6 + 3 \cdot 2 = 10$$

Ovviamente questa rimane solo una soluzione ammissibile del problema, per avere conferma che sia anche ottimale, occorre confrontarla con altre soluzioni ammissibili.

Coefficienti di costo ridotto

I coefficienti a costo ridotto sono valori numerici associati agli archi che non fanno parte della base, misurano la variazione del costo di trasporto al crescere dell'unità del valore del flusso associato tale arco.

Condizione di ottimalità

Se i coefficienti di costo ridotto di tutti gli archi fuori base sono non negativi, questo indica che la crescita del flusso su qualsiasi arco fuori base comporta una crescita del costo di trasporto o nessuna variazione.
In tal caso possiamo concludere che la soluzione di base attuale è ottima. Inoltre se tutti i coefficienti a costo ridotto sono strettamente positivi, la soluzione è unica.

Calcolo dei coefficienti a costo ridotto

Per calcolare il coefficiente relativo ad un arco fuori base, si aggiunge tale arco alla base attuale, considerare l'unico ciclo che si forma con tale aggiunta. Percorrendo il ciclo che si forma nel verso indicato dall'arco, sommo tutti i costi relativi agli archi percorsi in senso concorde e sottraggo tutti i costi relativi agli archi percorsi in senso opposto.

Condizione di illimitatezza

Insieme alla condizione di ottimalità, esiste una seconda condizione d'arresto per l'algoritmo, che si verifica, quando, l'aggiunta di un arco (alla base) con coefficiente di costo ridotto negativo forma un ciclo orientato.
La motivazione è facilmente intuibile, siccome il costo di trasporto diminuisce indefinitivamente.

Dimostrazione

Dato un flusso ammissibile $\{\bar{x}_{ij}\}$ corrispondente ad una certa base e con valore dell'obbiettivo (costo di trasporto) C.
Sia $\bar{c}_{rs} < 0$ il coefficiente di costo ridotto dell'arco fuori base (r, s).
Sia $r \to s \to l_1 \to \dots l_t \to r$ il ciclo orientato creato aggiungendo alla base l'arco (r, s)

Per ogni $\Delta \geq 0$ il seguente aggiornamento del flusso lungo gli archi del ciclo orientato dà origine ad un flusso ancora ammissibile, con obbiettivo di corrispondenza:

$$C + \Delta c_{rs} \to -\infty \quad \text{per } \Delta \to +\infty$$

Il che mostra come l'obbiettivo diverga a $-\infty$ sulla regione ammissibile.

Cambio di Base

Nel caso in cui la base scelta non rispetti ne la condizione di ottimalità ne la condizione di illimitatezza esiste un algoritmo per cambiare la base attuale in una ammissibile.

Scelgo l'arco fuori base con coefficiente di costo ridotto negativo [2] attraverso un approccio greedy, prendendo quello con coefficiente di costo ridotto minore, e chiamo tale coefficiente $\bar{c}$. Aggiungendo tale arco alla base, formando un ciclo. Tra tutti gli archi percorsi in verso opposto, rimuovo quello con quantità di prodotto su arco minore, e chiamo tale quantità Δ. Successivamente, assegno all'arco appena aggiunto una quantità di prodotto inviata pari a Δ ed aggiorno il flusso degli archi rimanenti dell'ex-ciclo.

Il nuovo costo è il costo precedente, Chiamato T il costo precedente, avremo che il nuovo costo sarà dato dalla formula $T + \bar{c} \cdot \Delta$.

Algoritmo del simplesso su rete

Dopo aver trovato una base, ripete iterativamente

1. Condizione di illimitatezza
2. condizione di ottimalità
3. cambio di base

Si può notare che nel caso degenere può cambiare la base, tenendo costante il trasporto e la soluzione di base.

Problema di 1ª fase

Se il valore ottimo, risultato dalla 1ª fase, è maggiore di 0, allora il problema originario ha regione ammissibile vuota (non ha soluzione). Se il valore ottimo è uguale a 0, il problema originario ha regione ammissibile non vuota (ha soluzione). Questo è dovuto al fatto che tutti gli archi di collegamento a q, hanno costo 1, mentre quelli del grafo originario hanno costo nullo. Un valore ottimo nullo, indica che esiste una soluzione all'interno del grafo originario.

Inoltre, in tal caso esiste un albero di supporto ottimo che contiene solo uno dei nuovi archi (incidenti su q). Eliminando tale arco si ottiene una base ammissibile per il problema originario.

Algoritmo del simplesso con capacità limitate

```
def symplex(edges):
    base_solution = find_base_solution(edges)

    while not optimal_solution(base_solution):
        base_solution = change_base(base_solution)

    return base_solution
```

Gli archi sono suddivisi in tre tipi: gli archi in base B, archi fuori base a valore nullo N_0 ed archi fuori base saturi (con flusso pari al proprio limite superiore) N_1.

[2] Ne esiste almeno uno altrimenti sarebbe stata soddisfatta la condizione di ottimalità

Una soluzione di base è definita ammissibile se tutti gli archi in base hanno flusso compreso tra 0 e la propria capacità massima. Inoltre la soluzione non è degenere se tutti gli archi hanno flusso strettamente compreso tra 0 e la capacità dell'arco.
Partendo da una tripla (B, N_0, N_1), per trovare la soluzione di base associata seguo i passaggi:

1. Inizializzazione
   (a) Pongo a 0 il flusso lungo tutti gli archi in N_0
   (b) Pongo pongo saturi tutti gli archi $\in N_1$
2. Determinare la quantità di prodotto inviata
   (a) Partendo dai nodi foglia dell'albero di supporto, invio la massima quantità di prodotto inviabile
   (b) DFS per calcolare il prodotto inviato su ogni arco
3. Verifico che la tripla in input sia una soluzione ammissibile

La tripla ammissibile viene trovata attraverso il metodo a due fasi:
Introduco un nodo aggiuntivo q, lo collego ad ogni sorgente e destinazione come nei casi recedenti.

Condizione di Ottimalità

Per gli archi in N_1 l'unica cosa possibile da fare è far decrescere la quantità di prodotto inviata, per questo voglio calcolare quanto varia il costo totale facendo **diminuire** la quantità di prodotto inviata sull'arco.

Per gli archi in N_0, il coefficiente di costo ridotto è non negativo

Per gli archi in N_1, il coefficiente di costo ridotto deve essere non positivo.

$$\bar{c}_{ij} \geq 0 \quad \forall (i,j) \in N_0 \qquad \bar{c}_{ij} \leq 0 \quad \forall (i,j) \in N_1$$

Se le disuguaglianze sono strette la soluzione ottima è unica.

Condizione di Illimitatezza

Se l'aggiunta alla base attuale di un arco in N_0, con coefficiente di costo ridotto negativo crea un ciclo orientato e **tutti** gli archi del ciclo hanno capacità pari a $+\infty$, allora il problema del flusso a costo minimo ha obbiettivo illimitato.

Cambio di base

Attraverso sempre un approccio greedy, l'arco entrante in base è

$$(i,j) \in \arg\max \Big\{ \max_{(i,j)\in N_0} -\bar{c}_{ij}, \max_{(i,j)\in N_1} \bar{c}_{ij} \Big\}$$

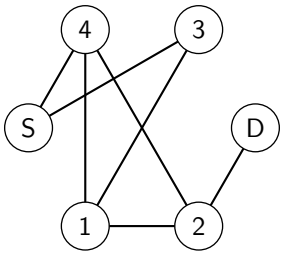
Per trovare l'arco uscente, aumento la quantità di prodotto Δ se l'arco aggiunto era in N_0, altrimenti diminuiamo Δ inviata lungo il ciclo in caso di arco in N_1. Al crescere di Δ:

- Arco si azzera, esce dalla base ed entra in N_0.
- Arco si satura, esce dalla base ed entra in N_1.

- L'arco fuori base si azzera/satura, la base non cambia, ma l'arco cambia di collocazione (da N_0 ad N_1 e viceversa)

Una volta effettuato il cambio di base, si applica il controllo delle condizioni sulla nuova tripla ammissibile. Viene ripetuto questo procedimento fino a quando le condizioni sono soddisfatte.

Flusso a costo massimo



Quando in un grafo di flusso voglio calcolare la massima trasmissione possibile tra un nodo sorgente S ed un nodo destinazione D. Il resto dei nodi della rete prende il nome di intermedio, ed hanno una capacità limitata c_{ij} che rappresenta la quantità massima di flusso inviabile attraverso l'arco.

Taglio a costo minimo

Si consideri $U \subset V$ tale che $S \in U$ e $D \notin U$. L'insieme di archi $S_U = \{(i,j) \in A : i \in U, j \notin U\}$ ovvero gli archi con il un solo estremo in U, è detto taglio della rete.

Ad un taglio è associabile anche un costo, pari alla somma delle capacità del taglio:

$$C(S_U) = \sum_{(i,j) \in S_U} c_{ij}$$

NOTA: Dalla sorgente alla destinazione non è possibile inviare una quantità di prodotto superiore alla capacità di taglio. Quindi il taglio a costo minimo indica il bottleneck della rete, ovvero la quantità massima di prodotto da sorgente a destinazione. L'algoritmo di risoluzione del flusso a costo massimo è anche la soluzione al problema del taglio a costo minimo.

Procedura di soluzione

Partire da un flusso ammissibile, $\bar{X} = (\bar{x}_{ij})_{(i,j) \in A}$ ed un cammino orientato $S = q_0 \to q_1 \to \cdots \to q_r \to q_{r+1} = D$, privo di archi saturi.
In questo caso $\bar{X}$ non è ottimo, dato che posso ancora inviare una quantità $\Delta = \min_{i=0\ldots r}[c_{q_i,q_{i+1}} - \bar{x}_{q_i,q_{i+1}}]$ di prodotto senza violare i vincoli di capacità degli archi. Sommando Δ otteniamo un cammino P da S a D contenente almeno un arco saturo.

Definiamo un nuovo grafo orientato $G(\bar{X}) = (V, A(\bar{X}))$, detto grafo associato al flusso $\bar{X}$. Il nuovo grafo ha gli stessi nodi della rete originaria ed ha il seguente insieme di archi A_f (archi forward) archi in P non saturi, A_b (archi backward) sono tutti gli archi della rete con $x_{ij} > 0$ (stanno inviando prodotto), rivolti in orientamento opposto.

Supponiamo che esista un cammino orientato P da S a D in $G(\bar{X})$. Per ogni arco (i,j) del cammino, calcoliamo il seguente valore:

$$\alpha_{ij} = \begin{cases} c_{ij} - \bar{x}_{ij} & \text{se} \quad (i,j) \in A_f(\bar{X}) \cap P \\ x_{ji} & \text{se} \quad (i,j) \in A_b(\bar{X}) \cap P \end{cases}$$

Note: α_{ij} indica la quantità di prodotto che si può rispedire indietro lungo l'arco, i.e. di quanto è possibile ridurre il flusso lungo l'arco (j, i).
Una volta calcolati i coefficienti α_{ij}, definisco

$$\Delta = \min_{(i,j) \in P} \alpha_{ij}$$

Note: Sottraendo questa quantità Δ non è mai possibile avere valori di flusso negativi.
Ed aggiorno i flussi $\bar{x}_{ij}$ nel modo seguente:

$$\bar{x}_{ij} = \begin{cases} \bar{x}_{ij} + \Delta & \text{se} \quad (i,j) \in A_f(\bar{X}) \cap P \\ \bar{x}_{ij} - \Delta & \text{se} \quad (j,i) \in A_b(\bar{X}) \cap P \\ \bar{x}_{ij} & \text{altrimenti} \end{cases}$$

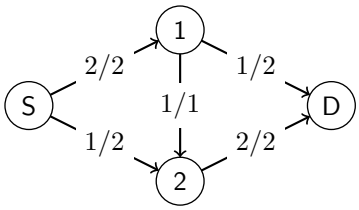
Note: segue che attraverso tale aggiornamento, sto inviando anche una quantità Δ in più da sorgente a destinazione.

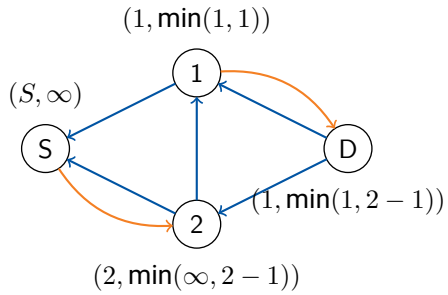
Diversamente se il grafo $G(\bar{X})$ non contiene cammini orientati da S a D, possiamo immediatamente concludere che il flusso $\bar{X}$ è soluzione ottima del problema di flusso massimo. Inoltre tutti i nodi raggiungibili dal nodo sorgente con un camino orientato formano l'insieme di nodi che andranno a formare il taglio minimo.

Algoritmo di Ford-Fulkerson

```python
#TODO
from collections import deque

infinity = 10**20

def ford_fulkerson(adj_list, S, D):
    while True:
        # Passo 1
        label = {
            S: (S, infinity)
        }

        visited = {S}
        frontier = deque([S])
        marked = set()

        while len(visited - marked) > 0:
            # Passo 2
            i = visited.popleft()
            marked.add(i)

            _, delta = label[i]
            for j, product_sended, capacity in adj_list[i]:
                if j in visited:
                    continue

                if product_sended < capacity:
                    # Forward edges
                    label[j] = (i, min(delta, capacity - product_sended))
                else:
                    # Backward edges
                    label[j] = (i, product_sended)

                if j not in visited:
                    visited.add(j)
```

```
                    frontier.append(j)

            # Passo 3
            if D in visited:
                _, delta = label[D]

                node = D
                while node != S:
                    update_arc(adj_list, (label[node], node), delta)
                    node, _ = label[node]
                break
        else:
            break
```





Teorema

Se si pone $U = E$, dove E è l'insieme dei nodi etichettati al momento della terminazione dell'algoritmo, si ha che il taglio S_U, indotto da U è soluzione ottima del problema di taglio minimo e il flusso attuale è soluzione ottima del problema di flusso massimo.

Dimostrazione

Al momento della terminazione dell'algoritmo si ha $S \in E$ (etichettato al passo 1) e $D \notin E$ (Perché rimosso al passo 3). Quindi l'insieme E induce effettivamente un taglio.

Se il valore di tale taglio coincide con il valore del flusso uscente da S, avendo già osservato che il costo di ogni taglio non è inferiore al valore del flusso massimo, possiamo concludere che esso è il taglio a costo minimo ed il flusso attualmente uscente da S è massimo.

Misurando il flusso uscente da S, esso coincide con il flusso uscente dai nodi in E, che viene spostato verso i nodi nel complemento $\bar{E} = V - E$, meno il flusso che in senso opposto: da $\bar{E}$ a E.

Per dimostrare che la quantità di prodotto complessivamente inviata da S a D è pari a:

$$\sum_{(i,j):(i,j)\in A, i\in E, j\in\bar{E}} \bar{x}_{ij} - \sum_{(j,i):(j,i)\in A, i\in E, j\in\bar{E}} \bar{x}_{ji}$$

osserviamo che si deve avere $\forall (i,j) \in A, i \in E, j \in \bar{E} \bar{x}_{ij} = c_{ij}$. Infatti, per assurdo si supponga che $\exists (i_1, j_1) \in A, i_1 \in E, j_1 \in \bar{E} : \bar{x}_{i_1 j_1} < c_{i_1 j_1}$. In tal caso $(i_1, j_1) \in A_f(\bar{X})$, ma al passo 2, il nodo j_1 dovrebbe essere stato aggiunto ad E. Il che contraddice $j_1 \notin E$.

Si deve avere inoltre che $\forall (j,i) \in A, i \in E, j \in \bar{E} \quad \bar{x}_{ji} = 0$, infatti, per assurdo si supponga che esista un $(j_1, i_1) \in A : i_1 \in E, j_1 \in \bar{E}, \bar{x}_{j_1 i_1} > 0$. In tal caso $(i_1, j_1) \in A_b(\bar{X})$, e quindi al passo 2, sarebbe anch'esso stato aggiunto ad E, contraddicendo l'ipotesi.

Sostituendo i valori di queste due osservazioni nella formula precedente, si ottiene che il valore de flusso è pari al costo del taglio introdotto da E:

$$\sum_{(i,j):(i,j)\in A, i\in E, j\in \bar{E}} c_{ij} = C(S_E)$$

Complessità dell'algoritmo

$O(|A||V|^2)$ (se i è scelto tramite un metodo FIFO). Esistono raffinamenti di questo algoritmo di complessità $O(|V|^3)$.

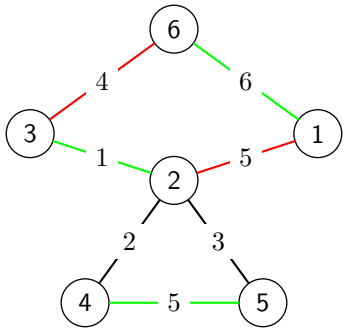
Matching

Definizione di Matching

Dato un grafo non orientato $G = (v, A)$, un matching è un sottoinsieme $M \subseteq A$ dell'insieme di archi A tale che in M non ci siano coppie di archi adiacenti[3].

Matching a peso massimo

Di tutti i matching possibili, vogliamo trovare quello il cui peso $w(M) = \sum_{e \in M} w_e$ sia massimo.



L'insieme di archi $M_1 = \{(1,2)\,(3,6)\}$ forma un matching di peso 9, mentre $M_2 = \{(2,3)\,(4,5)\,(1,6)\}$ ha un peso di 12. Algoritmi di matching sono utilizzati per determinare accoppiamenti ottimali tra i nodi.

Viene detto matching di **cardinalità massima**, l'insieme di problemi dove il peso di ogni arco è unitario. In caso di grafi bipartiti, si cerca una soluzione che colleghi i due set di nodi.

```
def matching_max_cardinality(adj_list):
    V1, V2 = biparted_sets(adj_list)

    M = initial_match()
    L = {}

    while unlabeled := (V1 - L.keys() - nodes_in(M)):
        R = set()
        L = {}

        start = unlabeled.pop()
        L[start] = ('E', '-')

        while available_nodes := L.keys() - R:
            current = available_nodes.pop()
            R.add(current)
            group, parent = L[current]

            if group == 'E':
                valid_neightbours = filter(lambda n: n not in L, adj_list[
                    current])
                for adj_node in valid_neightbours:
                    L[adj_node] = ('O', current)
```

[3] Con un nodo in comune

```
            else:
                adj_nodes = set(adj_list[current]) - L.keys()
                nodes = {node for node in adj_nodes
                               if frozenset((current, node)) in M}

                if nodes:
                    node = nodes.pop()
                    L[node] = ('E', current)
                else:
                    while L[current][1] != '-':
                        group, next_node = L[current]
                        if group == 'O':
                            M.add(frozenset((current, next_node)))
                        else:
                            M.remove(frozenset((current, next_node)))

                        current = next_node
                    break

    return M
```

Complessità

$O(\min(|V_1|, |V_2|)|A|)$, quindi polinomiale. Ad ogni iterazione la cardinalità del matching incrementa di 1 unità, appena raggiunta la cardinalità di V_1 e V_2 termina. Esiste un algoritmo più efficiente di complessità $O(|V|½|A|)$ ma non è trattato nel corso.

Note

Collegando un nodo sorgente alla prima classe di partizione, ed un nodo destinazione alla seconda classe, il problema è riconducibile ad un problema di flusso massimo, risolvibile con Ford-Fulkerson.

Assegnamento

Dato un grafo bipartito completo $G = (V_1 \cup V_2, E)$ con $V_1 = \{a_1 \dots a_n\}$ e $V_2 = \{b_1 \dots b_n\}$ ($|V_1| = |V_2|$), e $\forall (i,j) \in E \quad d_{ij} \geq 0$ ed interi, il problema d'assegnamento è trovare il matching M di cardinalità n tale per cui $\sum_{(i,j)\in M} d_{ij}$ è minimo.

Il numero di soluzioni ammissibili è ovviamente $n!$, siccome ad ogni nodo nella prima partizione deve corrispondere un nodo nella seconda.
Nei casi in cui $|V_1| \neq |V_2|$ vengono aggiunti elementi fittizi con collegamenti di costo 0 nell'insieme con meno elementi.

Algoritmo Ungherese

Presa in ingresso una tabella dei costi $T_0 = d_{ij}$ di ordine n come nel caso in esempio 1, dove ad ogni lavoratore $a_1 \dots a_n$ è associato costo per lavoro $b_1 \dots b_n$, il procedimento dell'algoritmo è il seguente:

- Si calcola il minimo valore presente in ogni colonna: $d_j^0 = \min_i d_{ij}$

	b_1	b_2	b_3	b_4
a_1	2	3	4	5
a_2	6	2	2	2
a_3	7	2	3	3
a_4	2	3	4	5

Table 1: Tabella dei costi d'esempio

- Per ogni colonna j, sottrarre ad ogni suo elemento il valore d_j^0 rispettivo
- Ripetere la stessa operazione per le righe della matrice, sottraendo il valore $d_i^1 = \min_j d_{ij}$ ad ogni elemento della riga

Seguendo il caso di esempio si ottiene una nuova tabella T_1:

	b_1	b_2	b_3	b_4
a_1	0	1	2	3
a_2	4	0	0	0
a_3	5	0	1	1
a_4	0	1	2	3

Table 2: Tabella T_1 dei costi d'esempio

È importante notare che tutti gli elementi della tabella $T_1 = d_{ij}^2$ dopo queste due operazioni sono strettamente positivi.

Definita la matrice $x_{ij} = 1$ se (i, j) è un assegnamento, 0 altrimenti; la richiesta di assegnamento di uno ed un solo lavoro per lavoratore è descritta da:

$$\sum_{j \in V_2} x_{ij} = 1 \quad \forall i \in V_1 \qquad \bigwedge \qquad \sum_{i \in V_1} x_{ij} = 1 \quad \forall j \in V_1$$

Possiamo osservare quindi che

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2 x_{ij} + \sum_{ij}^{n} d_j^0 + \sum_{j=1}^{n} d_i^1 = \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^2 x_{ij} + D_0 + D_1$$

Dato che la quantità $d_{ij}^2 > 0$ e $x_{ij} > 0$ per ogni coppia (i, j) si ha che:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} \geq D_0 + D_1$$

Nel caso in cui esista una soluzione di peso pari a D_0 e D_1 essa sarà ottima. Il problema diventa quindi determinare un sottoinsieme Δ di cardinalità massima degli 0 nella matrice T_2, tale che presi due elementi qualsiasi, essi siano indipendenti [4].

Se il problema ammette soluzione Δ tale che $|\Delta| = n$, allora a tale è un assegnamento. Consideriamo la matrice $x_{ij} = 1$ se $(i, j) \in \Delta$, nel caso in cui Δ non sia un assegnamento

$$\exists j : \sum_{i=1}^{n} x_{ij} \neq 1$$

Nel caso in cui $\sum_{i=1}^{n} x_{ij} = 0$, nessun elemento di Δ è presente nella colonna j, quindi dato che $|\Delta| = n$ dovranno esserci n elementi nelle rimanenti $n - 1$ colonne. Per il principio della piccionaia

[4] appartengano a righe e colonne diverse

almeno una colonna contiene due elementi. Quindi segue l'assurdo dato che gli elementi in Δ devono essere indipendenti, quindi devono appartenere a colonne diverse.

Nel caso in cui $\sum_{i=1}^{n} x_{ij} \geq 2$, in tal caso nella stessa colonna j ci sono 2 o più elementi di Δ segue l'assurdo identico al caso precedente.

Calcolo dell'assegnamento Δ

Dato un grafo $G = (A, B; E)$, con $a_1 \dots a_n \in A$ e $b_1 \dots b_n \in B$; tra il vertice a_i ed il vertice b_j si traccia un arco se e solo se $d_{ij}^2 = 0$.

Cercare il massimo insieme di 0 indipendenti equivale a risolvere il problema di matching di massima cardinalità.

Nel nostro caso di esempio quindi otterremo la soluzione

$$\Delta = \{(a_1, b_1)(a_2, b_3)(a_3, b_2)\}$$

tale per cui $|\Delta| = 4$ quindi $|\Delta| < n$.

Indicando genericamente con linee le righe o le colonne; il problema si trasforma ulteriormente quindi nel trovare un'insieme minimo di linee in grado di coprire tutti gli zeri della matrice T_2. È dimostrabile che il ricoprimento ottimo è formato esattamente da $|\Delta|$ linee, ed è costituito dalle righe a_i corrispondenti ai nodi non etichettati, ed alle colonne b_i corrispondenti alle colonne etichettate.

	b_1	b_2	b_3	b_4
a_1	0	1	2	3
a_2	4	0	0	0
a_3	5	0	1	1
a_4	0	1	2	3

Attraverso questo ricoprimento troviamo il minimo valore λ tra gli elementi non ricoperti, è strettamente positivo dato che tutti gli zeri sono coperti.

Si definisce a questo punto una nuova matrice $T_3 = d_{ij}^3$ tale per cui $d_{ij}^2 + d_i^3 + d_j^3$, dove $d_i^3 = -\lambda$ se la riga a_i non fa parte del ricoprimento, e $d_j^3 = \lambda$ se la colonna b_j è parte del ricoprimento.

In altre parole tutti gli elementi ricoperti da due linee in T_2 sono incrementati di λ, e tutti gli elementi non ricoperti sono decrementati di λ.

	b_1	b_2	b_3	b_4
a_1	0	0	1	2
a_2	5	0	0	0
a_3	6	0	1	1
a_4	0	0	1	2

Tutti gli elementi rimangono strettamente positivi, tutti gli elementi che sono decrementati, sono decrementati di una quantità pari al minimo tra essi.

Per ogni assegnamento sarà quindi vero che

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^3 x_{ij} - \sum_{j=1}^{n} d_j^3 - \sum_{i=1}^{n} d_i^3 + D_0 + D_1$$

Indicato con h_1 il numero di righe nel ricoprimento, ed h_2 il numero di colonne nel ricoprimento ($h_1 + h_2 = |\Delta|$), la formula sopra riportata diventa

$$\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}^3 x_{ij} + \lambda(n - |\Delta|) + D_0 + D_1$$

Siccome analogamente al caso precedente d_{ij}^3 ed x_{ij} sono positivi per ogni valore di (i, j), l'espressione $\lambda(n - |\Delta|) + D_0 + D_1$ forma il nuovo limite inferiore per il problema di assegnamento; risolvibile sempre cercando un sottoinsieme indipendente di cardinalità massima. Se la cardinalità della soluzione è ancora inferiore di n, si riapplica lo stesso procedimento.

Note su piccola ottimizzazione

Come matching iniziale su grafo bipartito associato agli zeri di T_3 conviene utilizzare il matching ottimo associato agli zeri di T_2. È dimostrabile che anche dopo l'aggiornamento di T_2 in T_3 l'insieme di zeri indipendenti soluzione di T_2 è contenuto nella soluzione di T_3.

Finitezza algoritmo

L'algoritmo termina quando si trova una matrice T_h contenente un sottoinsieme di zeri di cardinalità n. Nel caso in cui tutti i coefficienti siano interi l'algoritmo termina sicuramente, dato che il lower bound continua a crescere di un valore strettamente positivo ad ogni iterazione (come upper bound può essere considerata la somma degli elementi della matrice).

Complessità

$O(n^3)$ rientrando nella classe di algoritmi con complessità polinomiale.

Note

Rientra nella categoria di algoritmi costruttivi, con la possibilità di rivedere decisioni passate: ad ogni iterazione si ha un insieme Δ definisce un assegnamento incompleto, che può essere alterato durante ogni iterazione.

Branch and bound

Il generico problema associato a questo algoritmo è determinare il massimo valore che può assumere una funzione $f(x)$.

Upper Bound

Considerato un insieme $T \subseteq S$, un' upper bound per un insieme T è un valore $U(T)$ tale che $U(T) \geq f(x) \forall x \in T$.

Il valore di $U(T)$ è calcolato tramite una procedura che deve avere tempo di esecuzione minore possibile, e precisione più elevata possibile.

Indicato con $\alpha(f,T) = \max_{x \in T} f(x)$ il valore ottimo della funzione f su T, è definito rilassamento:

$$\alpha(f',T') = \max_{x \in T'} f'(x)$$

Dove $T \subseteq T'$ e $f'(x) \geq f(x) \quad \forall x \in T$.

Il rilassamento è un upper bound del sottoinsieme T, dato che $\alpha(f',T') \geq \alpha(f,T)$: chiamata $\bar{x} \in T$ una soluzione ottima del problema su T , e $x' \in T'$ una soluzione ottima del rilassamento; Siccome $T \subseteq T'$, allora $\bar{x} \in T'$ e $f'(\bar{x}) \geq f(\bar{x})$, da cui $\alpha(f',T') \geq \alpha(f,T)$.

Per trovare un' upper bound di un insieme, basta risolvere quindi il problema di rilassamento.

Lower Bound

Indicato con L_B, rappresenta la limitazione inferiore della funzione f sull'intero insieme S:

$$L_B \leq f(x) \quad \forall x \in S$$

Per ottenere una precisione più elevata possibile, dato un insieme di punti $y_1 \dots y_h \in S$

$$L_B = \max \left\{ f(y_i) : i = 1 \dots h \right\} \leq f(x)$$

I punti $y_1 \dots y_h$ sono spesso individuati attraverso un euristica [5].

Branching

L'operazione consiste nel sostituire l'insieme $T \subseteq S$ con una sua partizione $T_1 \dots T_m$. $T_1 \dots T_m$ formano una partizione se e solo se:

$$T = \bigcup_{i=1}^{m} T_i \qquad \bigwedge \qquad T_i \cap T_j = \emptyset \quad \forall i \neq j$$

La partizione è rappresentata con un albero.

[5] Rapido algoritmo che restituisce delle buone soluzioni ammissibili ma non necessariamente ottime

Eliminazione di sottoinsiemi

Un insieme T_i viene eliminato se $U(T_i) \leq L_B$. In tale insieme non può essere contenuto il massimo di f.

Algoritmo branch and bound

1. Posto $C = \{S\}$ il set di sottoinsiemi analizzabili, e $Q = \emptyset$ il set di insiemi eliminati, si calcoli $U(S)$ e L_B in caso si disponga di un euristica, altrimenti $L_B = -\infty$.
2. Selezionato $T \in C$, ad esempio il sottoinsieme con upper bound maggiore: $U(T) = \max_{Q \in C} U(C)$
3. Si sostituisca T in C con la sua partizione in k sottoinsiemi: $C = C \cup \{T_1 \dots T_k\} - \{T\}$
4. Calcolare per ciascuno dei sottoinsiemi il valore $U(T_i) \quad \forall i \in 1 \dots k$
5. Aggiornare il valore L_B con il massimo valore di f osservato durante l'esecuzione dell'algoritmo.
6. Spostare da C a Q tutti i sottoinsiemi per cui $U(T') \leq L_B$: $C = C - \{T' : U(T') \leq L_B\}$, $Q = Q \cup \{T' : U(T') \leq L_B\}$
7. Se $C = \emptyset$, allora L_B coincide con il valore ottimo. Altrimenti ripartire dal passo 2

Esistono problemi la cui regione ammissibile è vuota, per questo il loro lower bound risulta $-\infty$.

Dimostrazione che L_B è soluzione ottima

Durante tutto l'algoritmo $C + Q$ forma una partizione di S, quindi al termine dell'algoritmo esiste sicuramente $T \in Q$ tale che $\bar{x} \in T$, ma dato che T è stato cancellato, allora

$$f(\bar{x}) \leq U(T) \leq L_B \leq f(\bar{x})$$

da cui $L_B = f(\bar{x})$.

Per trovare il minimo di una funzione, o consideriamo $-f(x)$, o invertiamo il ruolo di L_B ed U_B

Knapsack Problem

È dato uno zaino con capacità b (intero positivo) ed un numero di oggetti n. A ciascun oggetto i è associato un peso p_i ed un valore v_i.
Il problema consiste trovare un gruppo di oggetti il cui peso sia inferiore alla capacità b, e valore cumulativo sia massimo.

Applichiamo un modello matematico per studiare questo modello. Indichiamo con $x_i = 1$ gli oggetti che sono presi nello zaino, e con $x_i = 0$ gli oggetti non presi. In tal caso $\sum_i^n p_i x_i \leq b$ e $\sum_i^n v_i x_i$ è massima.

Calcolo dell'upper bound

Aggiungendo l'ipotesi che gli oggetti possano essere inseriti parzialmente: $x \in [0, 1]$

1. Riordiniamo gli oggetti in ordine non crescente rispetto ai rapporti peso/valore.

2. Calcoliamo i valori $b-p_1$, $b-p_1-p_2$, ...Calcoliamo il massimo valore di r per cui $b-\sum_{i=1}^{r+1} p_i > 0$ risulta vera.
   Nel caso in cui non si arriva ad un valore negativo, allora la soluzione ottima è mettere tutti gli oggetti nello zaino.

3. La soluzione ottima del rilassamento lineare è

$$x_1 \dots x_r = 1 \quad , \quad x_{r+2} \dots x_n = 0$$
$$x_{r+1} = \frac{b - \sum_{j=1}^{r} p_j}{p_r + 1}$$

   con valore ottimo $\sum^r v_j + v_{r+1} \frac{b-\sum^r p_j}{p_{r+1}}$.

4. $x_1 \dots x_r$ è una soluzione ammissibile al problema originario, quindi utilizziamo quella per calcolare il lower bound. Per upper bound, prendiamo il valore ottimo calcolato al punto 3 arrotondato per difetto.

Operazione di branching

Indichiamo con $S(I_0, I_1)$ tutti i sottoinsiemi che non contengono elementi in I_0 ma contengono tutti gli elementi appartenenti ad I_1. Supponendo di non essere nel caso banale in cui non tutto gli oggetti sono contenuti nello zaino, utilizziamo la regola di branching $S(\{r+1\}, \emptyset)$ tutti i sottoinsiemi in cui non è presente l'oggetto $r+1$ e $S(\emptyset, \{r+1\})$ tutti i sottoinsiemi in cui è presente $r+1$.

Per calcolare l'upper bound di questi insiemi al punto 3 basta forzare nella formula tutti gli elementi in I_1 e non prendere in considerazione gli elementi in I_0.

Procedura di calcolo

1. Caso base: se $b - \sum_{i \in I_1} p_i < 0$ allora il nodo non ha soluzioni ammissibili. In tal caso poniamo $U(S(I_0, I_1)) = -\infty$.

2. Altrimenti trovo il massimo valore di r per cui $b - \sum_{i \in I_1} p_i - \sum_{i \notin I_1}^{r} p_i \geq 0$.

3. Se esiste tale valore di r allora l'upper bound di tale insieme è

$$\sum_{i \in I_1} v_i + \sum_{h=1}^{r} v_{i_h} + \left\lfloor v_{i_{r+1}} \frac{b - \sum_{i \in I_1} p_i - \sum_{h=1}^{r} p_{i_h}}{p_{i_{r+1}}} \right\rfloor$$

   Con soluzione ammissibile $N = I_1 \cup \{i_1 \dots i_r\}$

4. Se si sottraggono i pesi di tutti gli oggetti senza mai arrivare ad un valore negativo, l'upper bound sarà semplicemente:

$$\sum_{i \in I_1} v_i + \sum_{h=1}^{k} v_{i_h}$$

   Con soluzione ammissibile $N = I_1 \cup I_f \in S$

Complessità

Il caso peggiore si ha quando non si riesce a cancellare alcun nodo. In tal caso il numero di nodi generati è dell'ordine 2^n. Quindi non è migliore rispetto ad un algoritmo di enumerazione, ma mediamente il tempo di esecuzione è minore. Questo è valido per tutti gli algoritmi di branch and bound.

Programmazione Dinamica

Questa metodologia è applicabile solamente per problemi suddivisibili in n blocchi. In ogni blocco k $(k = 1 \dots n)$ ci si trova in uno degli stati s_k appartenenti all'insieme di stati S_k.

In uno stato s_k si deve prendere una decisione d_k appartenente all'insieme di possibili decisioni $D_k(s_k)$.

Si definisce la funzione di transizione di stato $t(d_k, s_k)$, come lo spostamento da uno stato s_k ad s_{k+1} attraverso la decisione d_k.

Se in un blocco k ci si trova in uno stato s_k e si prende la decisione Il risultato della funzione obbiettivo f è dato dalla somma (in rari casi prodotto) di tutti i contributi $u(d_k, s_k)$ dipendenti dalle decisioni prese in ogni stato.

Principio di ottimalità

Allo stato s_k la sequenza di decisioni ottime da prendere nei blocchi $k + 1 \dots n$ è indipendente dalle decisioni prese per arrivare allo stato s_k.

Problema dello zaino

Chiamando b il numero di oggetti a disposizione, ogni stato $s_k \in S_{0\dots b}$ rappresenta lo spazio disponibile nello zaino.
Chiamata p_k il peso dell'oggetto k, indicando con $d_k = 0$ o $d_k = 1$ se l'oggetto k è preso nello zaino, la funzione D_k è esprimibile come:

$$D_k(s_k) = \begin{cases} \{0, 1\} & \text{se} \quad s_k \geq p_k \\ \{0\} & \text{se} \quad s_k < p_k \end{cases}$$

Il contributo è quindi $u(d_k, s_k) = d_k \cdot v_k$, e la funzione di transizione: $t(d_k, s_k) = s_k - p_k \cdot d_k$.

Il principio di ottimalità è rispettato: ad ogni blocco k, le decisioni ottime da k ad n, si ottengono risolvendo il problema dello zaino con capacità s_k ed oggetti $k \dots n$, indipendentemente dalle decisioni prese per arrivare al blocco k.

$$f'_n(s_n) = \max_{d_n \in D_n(s_n)} u(d_n, s_n) = d'_n u(d'_n, s_n)$$

In altre parole, in ogni stato finale l'ultimo oggetto nello zaino è inserito solo se c'è posto.

Chiamiamo $f'_k(s_k)$ la funzione che calcola il valore ottimo delle somme dei contributi dei blocchi $k \dots n$ partendo dallo stato s_k.

Se mi trovo in uno stato s_k e prendo la decisione d_k il contributo ottimo complessivo dei blocchi $k \dots n$ sarà dato, per il principio di ottimalità da $u(d_k, s_k) + f'_{k+1}(t(d_k, s_k))$.
Da cui prendendo la decisione d'_k che ne massimizza il valore, avremo:

$$f'_k(s_k) = \max_{d_k \in D_k(s_k)} \left\{ u(d_k, s_k) + f'_{k+1}(t(d_k, s_k)) \right\}$$

Il blocco 1 ha un unico stato s_1, quindi il valore di $f'_1(s_1)$ coincide con il valore ottimo del problema, caratterizzato dalle decisioni ottime d'_k.

Algoritmo - valore ottimo

1. Per ogni $s_n \in S_n$ si calcoli $f'_n(s_n)$ e la corrispondente decisione ottima $d'_n(s_n)$.
   Si ponga $k = n - 1$
2. Per ogni $s_k \in S_k$ si calcoli
$$f'_k(s_k) = \max_{d_k \in D_k(s_k)} \left\{ u(d_k, s_k) + f'_{k+1}(t(d_k, s_k)) \right\}$$
3. Se $k = 1$ si ha che $f'_1(s_1)$ è il valore ottimo del problema, altrimenti ripetere il passo 2 con $k = k - 1$

Algoritmo - soluzione ottima

1. Si ponga $s'_1 = s_1$ e $k = 1$
2. Per il blocco k, la decisione ottima è $d'_k(s'_k)$, si ponga
$$s'_{k+1} = t(d'_k(s'_k), s'_k)$$
3. Se $k = n$ l'algoritmo termina, e la soluzione ottima è rappresentata dalle decisioni $d'_1(s'_1) \dots d'_n(s'_n)$.
   Altrimenti si ripete il passo 2 con $k = k + 1$

Note

La programmazione dinamica rientra tra gli algoritmi di enumerazione implicita.

Complessità

Dato che per ogni stato è necessario calcolare $u(d_k, s_k)$, $t(d_k, s_k)$ e $u(d_k, s_k) + f'_{k+1}(t(d_k, s_k))$ ed il minimo dei vari d_k, il numero di operazioni richieste per ogni stato è $\sum_{s_k \in S_k} 4|D_k(s_k)|$,. L'algoritmo, siccome eseguito su n blocchi, richiede un numero di operazioni pari a:

$$\sum_{k=1}^{n} \sum_{s_k \in S_k} 4|D_k(s_k)|$$

Nel caso particolare del problema dello zaino, $|D_k(s_k)| = 2$ e $|S_k| = b + 1$, in definitiva $O(nb)$.

Confrontando questa complessità con l'algoritmo branch and bound 2^n, se b è molto inferiore di 2^n, allora questo algoritmo è migliore.

La complessità di questo problema però non è lineare come appare, ma esponenziale, infatti, come vedremo più avanti, la complessità del problema è esponenziale e legata al numero di bit richiesti per rappresentare il problema.

??? b cresce esponenzialmente in base al numero di bit richiesti per immagazzinare b stesso

Classificazione di complessità

Problemi di ottimizzazione

Un problema di ottimizzazione è formato da un'insieme di istanze, ognuna rappresentata dalla coppia (f, S): funzione obbiettivo e regione ammissibile. La regione ammissibile S contiene tutti i possibili elementi che, presi singolarmente, costituiscono una soluzione.

Nel caso in cui il problema di ottimizzazione sia di massimo, allora la coppia (f, S) è rappresentata come

$$\max_{x \in S} f(x)$$

Risolvere vuol dire trovare $x' \in S$ tale che $f(x') \geq f(x) \quad \forall x \in S$. x' viene quindi detta soluzione ottima dell'istanza, mentre $f(x')$ viene detto valore ottimo dell'istanza.

Data un'istanza S con un numero arbitrario di punti, il problema di ottimizzazione viene detto ottimizzazione combinatoria se i punti sono numerabili, ed ottimizzazione continua se i punti non sono numerabili.

Clique (ricerca di sottografi con almeno un arco) TSP (*Traveling Salesman Problem*): dato un grafo completo $G = (V, A)$ con distanze degli archi d_{ij} interi non negativi. Individuare nel grafo il circuito hemiltoniano di cammino minimo.

Risolvere il problema di ottimizzazione, attraverso un algoritmo $\mathcal{A}$ in grado di fornire una soluzione ottima ed il valore ottimo.

Se ci limitiamo a problemi di ottimizzazione combinatoria, è semplice identificare un algoritmo di risoluzione, infatti se la regione ammissibile S contiene un numero di finiti di elementi, basta un'enumerazione completa per elencare tutte possibili soluzioni.

Molto spesso i problemi di risoluzione completa richiedono tempi di esecuzione esagerati, per questo è utile sapere se è sempre possibile trovare soluzioni eseguibili in tempi ragionevoli.