G opt

Documentazione del progetto di ottimizzazione su reti

Michelangelo Fiorentino

Ricerca Operativa
Prof. Stefano Giordani
Università degli Studi di Roma Tor Vergata

Sommario

G-opt è un programma Python sviluppato durante il corso di Ricerca Operativa 2025/2026. Permette di caricare una rete, risolvere i problemi studiati e visualizzare i risultati. Alcuni algoritmi sono implementati direttamente, mentre altri problemi sono formulati e risolti con CVXPY.

1 Obiettivi e ambito

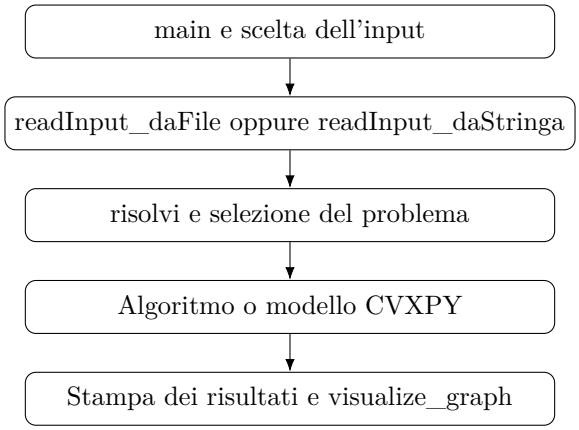
Il programma tratta grafi non orientati e digrafi. Nel seguito si assumono grafi semplici, senza cappi né archi duplicati. Un grafo non orientato è indicato con $G = (V, E)$, dove V è l'insieme dei vertici ed E quello degli spigoli. Un digrafo è indicato con $D = (N, A)$, dove N è l'insieme dei nodi ed A quello degli archi. Si pone $n = |V|$ e $m = |E|$ per un grafo, $n = |N|$ e $m = |A|$ per un digrafo.

Problema	Metodo adottato
Minimo insieme dominante	formulazione risolta con CVXPY
Minimo vertex cover	formulazione risolta con CVXPY
Massimo insieme Stabile	Riduzione a vertex cover
Massima Clique	Riduzione a insieme stabile
Massimo Matching	formulazione risolta con CVXPY
Albero ricoprente di costo minimo	Algoritmo di Prim–Dijkstra
Cammini minimi	Algoritmo su reti acicliche, Dijkstra, Bellman-Ford
Massimo flusso	Ford Fulkerson
Flusso a costo minimo	Programmazione lineare CVXPY
Trasporto	Caso particolare del flusso a costo minimo

2 Struttura del programma e interfaccia

2.1 Moduli e chiamate

File	Responsabilità
`main.py`	Interazione, selezione del problema e chiamate.
`Read_Input.py`	Lettura da file o stringa; restituzione dei dati.
`Matrici_Grafo.py`	Indicizzazione, adiacenza e incidenza.
`Algoritmi_NP_hard.py`	Modelli binari e riduzioni tra problemi.
`Algoritmi_Principali.py`	Albero, cammini e flussi.
`Algoritmi_supporto.py`	Ricerca, connessione, topologia e bipartizione.
`Visualizza.py`	Disegno e evidenziazione della soluzione.



Gli algoritmi richiamano funzioni di supporto o matrici secondo necessità. Per esempio, clique richiama stabile sul complemento; stabile richiama vertex cover; vertex cover costruisce la matrice di incidenza e invoca CVXPY. NetworkX è usato per la rappresentazione grafica, non come risolutore dei problemi principali.

2.2 Interfaccia

L'interazione avviene attraverso due menu testuali. Il primo permette di scegliere come inserire la rete:

```
1. Carica un file .txt esistente
2. Scrivi/Incolla input in una finestra
3. Stampa la legenda di inserimento dati
4. Chiudi il programma
```

Dopo il caricamento, il secondo menu consente di visualizzare la rete oppure scegliere il problema da risolvere:

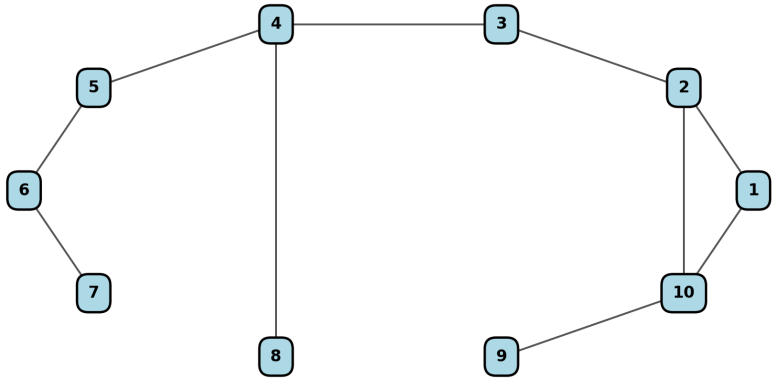
```
0. Visualizza rete inserita
1. Minimo Insieme Dominante
2. Minimo Vertex Cover (insieme ricoprente)
3. Massimo Insieme Stabile
4. Massima Clique
5. Massimo Matching (abbinamento)
6. Albero Ricoprente di Costo Minimo
7. Cammini Minimi: nodo-rete
8. Cammino Minimo: nodo-nodo
9. Massimo Flusso
10. Flusso di costo minimo
11. Cambia rete
12. Esci dal programma
```

I menu sono riportati in forma compatta. La scelta viene confermata con Invio; quando necessario vengono richiesti sorgente e destinazione. Tkinter gestisce la selezione del file e la finestra per incollare i dati; Matplotlib mostra la rete e gli elementi della soluzione.

3 Rappresentazione della rete

La rete viene descritta mediante un file di testo contenente il tipo di grafo, gli attributi disponibili e la lista degli archi. Il file `1_grafo_semplice.txt` offre un esempio di grafo non orientato con dieci nodi e dieci spigoli:

```
0 0 0 0 0
10 1
10 2
10 9
2 1
2 3
3 4
4 8
4 5
5 6
6 7
```



La prima riga contiene cinque indicatori binari: il primo distingue grafo non orientato (0) e digrafo (1); i successivi indicano la presenza di costi, capacità massime, capacità minime e divergenze dei nodi. In questo esempio sono tutti nulli: ciascuna riga successiva contiene soltanto i due estremi di uno spigolo. I nomi sono letti come stringhe. La figura è generata dalla funzione `visualize_graph` del progetto.

Se il quinto indicatore vale 1, dopo la lista degli archi si inserisce la riga `Nodi:`, seguita da una riga per ogni nodo con il nome e la relativa offerta o domanda. Un valore positivo indica un'offerta, uno negativo una domanda e zero un nodo di transito. Per esempio:

```
Nodi:
Fabbrica_A 20
Fabbrica_B 30
Negozio_1 -15
Negozio_2 -25
Fittizio -10
```

3.1 Dal testo alle liste

Il modulo `Read_Input.py` legge i dati da file con `readInput_daFile`, oppure dal testo inserito nell'interfaccia con `readInput_daStringa`. Dopo il controllo dell'intestazione e del numero di attributi, costruisce la lista dei nodi e quella degli archi. Nel ramo dedicato al grafo non orientato compare questo estratto:

```
vertice1 = lista_riga[0]
vertice2 = lista_riga[1]
if vertice1 not in nodi:
    nodi.append(vertice1)
if vertice2 not in nodi:
    nodi.append(vertice2)
archi.append(tuple(sorted((vertice1,vertice2))))
```

Ogni nodo è inserito alla prima occorrenza. Gli estremi di uno spigolo sono ordinati lessicograficamente, così da adottare una rappresentazione coerente delle coppie non orientate. Nel digrafo, invece, si conserva l'ordine sorgente–destinazione. La funzione restituisce tipo, nodi, archi, costi, capacità massime, capacità minime e divergenze; gli attributi degli archi seguono lo stesso ordine della lista degli archi.

3.2 Dalle liste alle matrici

Il modulo `Matrici_Grafo.py` associa a ogni nodo un indice numerico attraverso `Dizionari`. Nell'esempio l'ordine di prima comparsa è $[10, 1, 2, 9, 3, 4, 8, 5, 6, 7]$: il nodo `10` ha quindi indice 0, il nodo `1` indice 1, e così via. Gli indici permettono di costruire le matrici NumPy utilizzate nei modelli di ottimizzazione.

La matrice di adiacenza ha dimensione $n \times n$ e registra quali nodi sono collegati. In `Adiacenza_Grafo`, per ogni spigolo di estremi indicizzati con i, j, vengono aggiornate entrambe le posizioni:

```
i = Arco[0]
j = Arco[1]
A[i,j] += 1
A[j,i] += 1
```

La simmetria riflette l'assenza di orientamento. Per lo spigolo tra `10` e `1`, per esempio, si ottiene $A_{0,1} = A_{1,0} = 1$.

La matrice di incidenza ha invece dimensione $n \times m$: ciascuna colonna identifica uno spigolo e ciascuna riga un nodo. L'estratto di `Incidenza_Grafo` assegna un valore unitario ai due estremi:

```
i = Arco[0]
j = Arco[1]
A[i,k] = 1
A[j,k] = 1
```

Nel digrafo si adotta la convenzione $+1$ al nodo di partenza e -1 al nodo di arrivo.

4 Problemi su grafi non orientati

4.1 Insieme dominante minimo

Un insieme è dominante se ogni vertice del grafo appartiene all'insieme oppure è adiacente ad almeno un vertice dell'insieme. Si cerca quello di cardinalità minima. La variabile $x_v \in \{0, 1\}$ indica se il vertice v è selezionato; $A(v)$ indica la lista di adiacenza di v.

$$\min \sum_{v \in V} x_v \quad \text{s.t.} \quad x_v + \sum_{u \in A(v)} x_u \geq 1 \ (v \in V), \quad x_v \in \{0, 1\}.$$

```
x = cp.Variable(Matrice_Adiacenza.shape[0], boolean=True)
constraints = [a @ x >= b]
objective = cp.Minimize(c @ x)
problem = cp.Problem(objective, constraints)
problem.solve()
```

Qui `a` è la matrice di adiacenza più identità; `b` e `c` sono vettori unitari.

Esempio. Sul file `1_grafo_semplice.txt`, i vertici selezionati sono $\{10, 4, 6\}$.

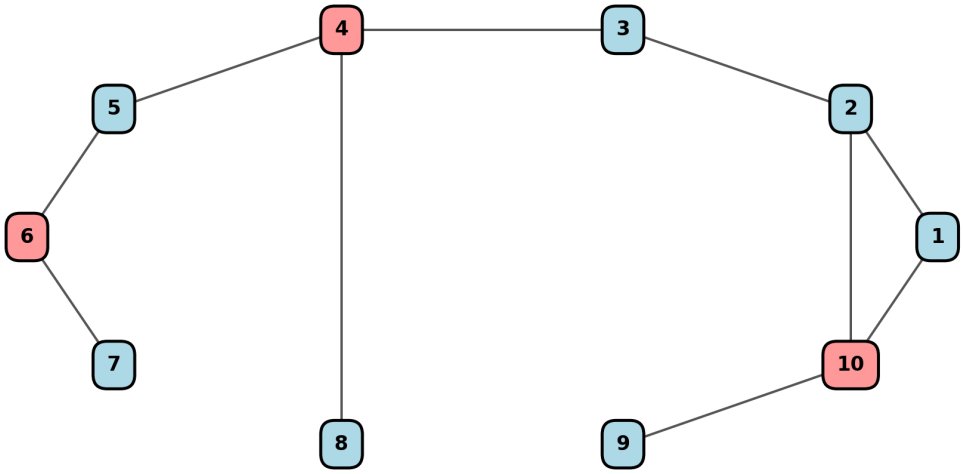


Figura 1: Dominante minimo sul file `1_grafo_semplice.txt`. I tre nodi selezionati dominano tutti i vertici.

4.2 Vertex cover minimo

Un vertex cover contiene almeno un estremo di ogni spigolo. Si cerca quello di cardinalità minima. La variabile $x_v \in \{0,1\}$ indica se il vertice v è selezionato.

$$\min \sum_{v \in V} x_v \quad \text{s.t.} \quad x_u + x_v \geq 1 \ (\{u,v\} \in E), \quad x_v \in \{0,1\}.$$

```
x = cp.Variable(len(Nodi), boolean=True)
funzione_obiettivo = cp.Minimize(c @ x)
vincolo= [a @ x >= b]
problem = cp.Problem(funzione_obiettivo,vincolo)
problem.solve()
```

Qui `a` è la matrice di incidenza trasposta; `b` e `c` sono vettori unitari.

Esempio. Sullo stesso file, il cover restituito è $\{10, 2, 4, 6\}$, di cardinalità 4.

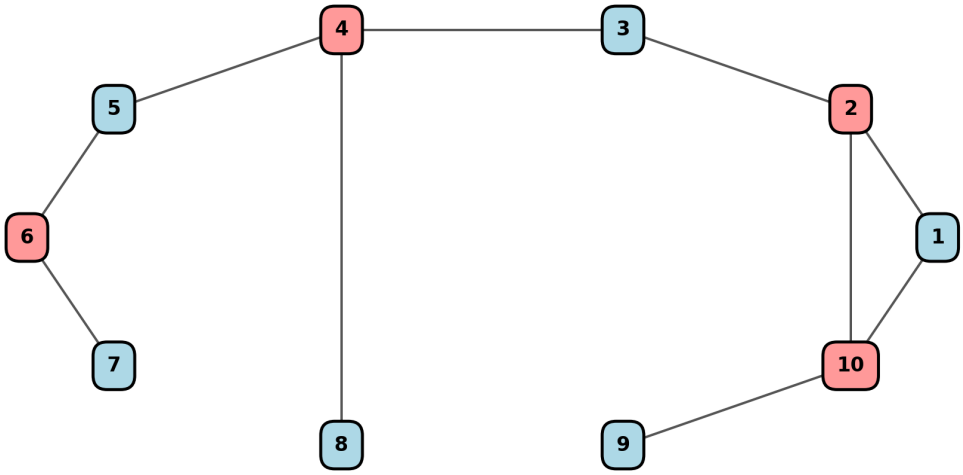


Figura 2: Vertex cover minimo sul file `1_grafo_semplice.txt`. Ogni spigolo incontra almeno uno dei quattro nodi rosa.

4.3 Insieme stabile massimo

Un insieme stabile contiene vertici a due a due non adiacenti. Si cerca quello di cardinalità massima. La variabile $x_v \in \{0,1\}$ indica se il vertice v è selezionato.

$$\max \sum_{v \in V} x_v \quad \text{s.t.} \quad x_u + x_v \leq 1 \ (\{u,v\} \in E), \quad x_v \in \{0,1\}.$$

```
Nodi_Evidenziati_Vertex_Cover,Archi_Evidenziati = Minimo_Vertex_Cover(Nodi,Archi)
Nodi_Evidenziati = []
Archi_Evidenziati = []
for Nodo in Nodi:
    if Nodo not in Nodi_Evidenziati_Vertex_Cover:
        Nodi_Evidenziati.append(Nodo)
```

Il codice ottiene lo stabile I come complemento del vertex cover minimo W risolto con CVXPY: $I = V \setminus W$.

Esempio. Il risultato è $\{1, 9, 3, 8, 5, 7\}$, di cardinalità 6.

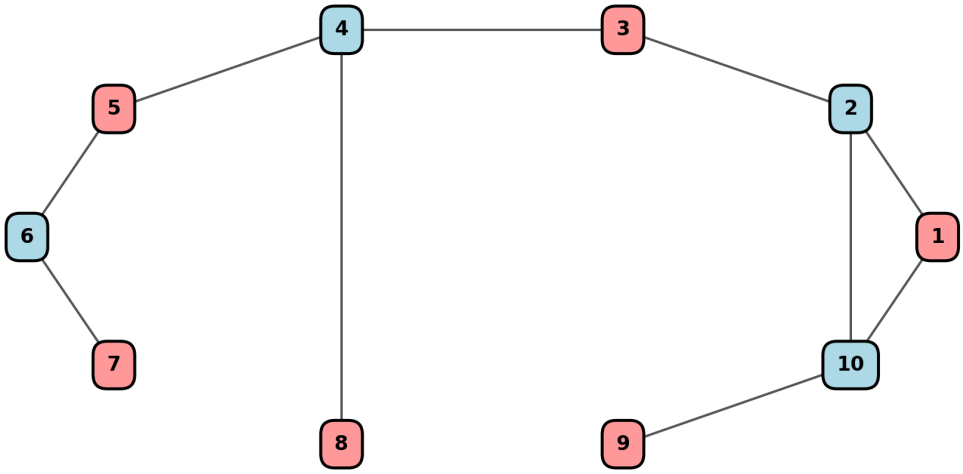


Figura 3: Insieme stabile massimo sullo stesso file. Nessuno spigolo unisce due dei sei nodi selezionati.

4.4 Clique massima

Una clique contiene vertici a due a due adiacenti. Si cerca quella di cardinalità massima. La variabile $x_v \in \{0,1\}$ indica se il vertice v è selezionato. Una clique è un insieme stabile sul grafo complemento. Indicando con $\overline{E} = \{\{u,v\} : u,v \in V,\ u \neq v,\ \{u,v\} \notin E\}$ l'insieme degli spigoli del complemento, il problema si formula come:

$$\max \sum_{v \in V} x_v \quad \text{s.t.} \quad x_u + x_v \leq 1 \ (\{u,v\} \in \overline{E}), \quad x_v \in \{0,1\}.$$

```
Nodi_Complemento,Archi_Complemento= Liste_Grafo_Complemento(Nodi,Archi)
Nodi_Evidenziati,Archi_Evidenziati = Massimo_Insieme_Stabile(Nodi_Complemento,Archi_Complemento)
return Nodi_Evidenziati,Archi_Evidenziati
```

Il codice risolve l'insieme stabile sul grafo complemento.

Esempio. La clique restituita è $\{10, 1, 2\}$, di cardinalità 3.

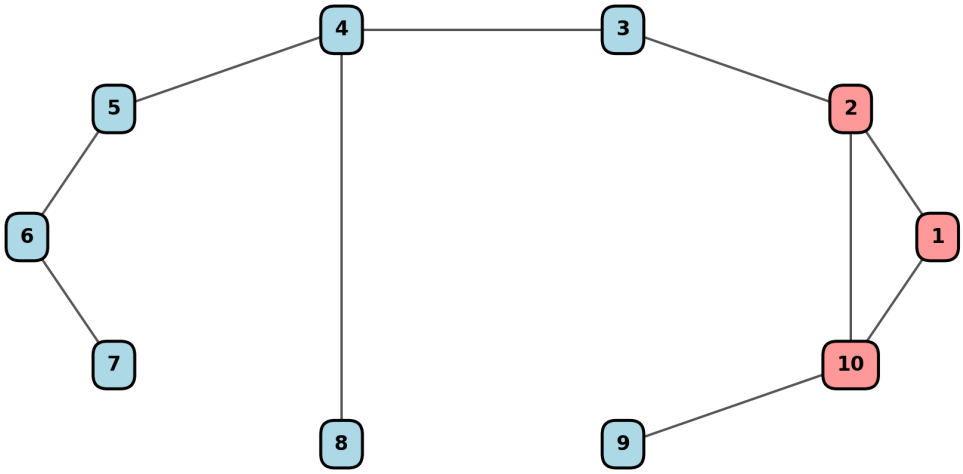


Figura 4: Clique massima sul file `1_grafo_semplice.txt`. Sono evidenziati i nodi 10, 1 e 2.

4.5 Matching massimo

Un matching è un insieme di spigoli senza estremi comuni. Si cerca quello di cardinalità massima. La variabile $x_e \in \{0,1\}$ indica se lo spigolo e è selezionato; $\Gamma(v)$ indica la stella del vertice v.

$$\max \sum_{e \in E} x_e \quad \text{s.t.} \quad \sum_{e \in \Gamma(v)} x_e \leq 1 \ (v \in V), \quad x_e \in \{0,1\}.$$

```
x = cp.Variable(len(Archi),boolean = True)
funzione_obbiettivo = cp.Maximize(c @ x)
a = Incidenza_Grafo(Nodi,Archi)
vincoli = [a @ x <= 1]
problema = cp.Problem(funzione_obbiettivo, vincoli)
problema.solve()
```

Qui `c` è un vettore unitario.

Esempio. Gli spigoli selezionati sono $\{(10,9),(1,2),(4,8),(6,7)\}$, per un totale di 4.

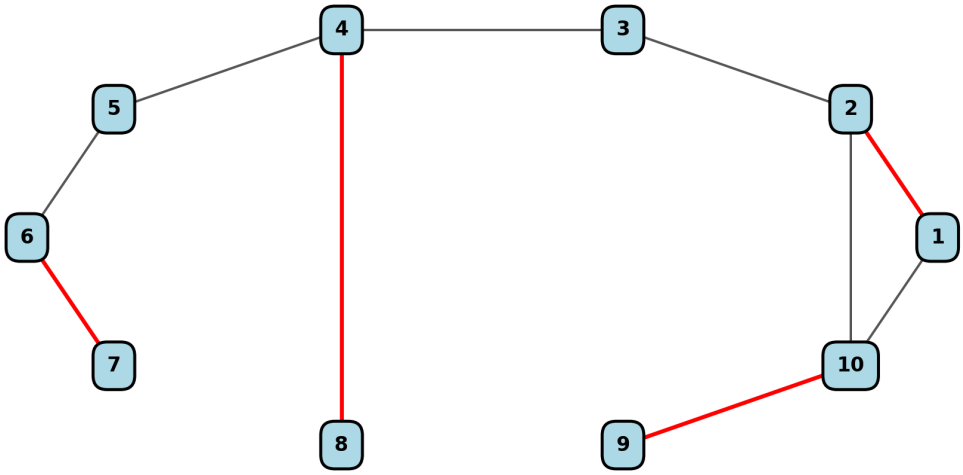


Figura 5: Matching massimo sul file `1_grafo_semplice.txt`. I quattro spigoli rossi non condividono vertici.

5 Albero ricoprente di costo minimo

Dato un grafo connesso $G = (V, E)$, con costo c_e associato a ogni spigolo, si cerca un albero ricoprente di costo minimo. La variabile $x_e \in \{0, 1\}$ indica se lo spigolo e è selezionato; $E(S)$ è l'insieme degli spigoli con entrambi gli estremi in $S \subseteq V$.

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ \text{s.t.} \quad & \sum_{e \in E} x_e = n - 1, \\ & \sum_{e \in E(S)} x_e \leq |S| - 1 \quad (\varnothing \neq S \subsetneq V), \\ & x_e \in \{0, 1\} \qquad\qquad (e \in E). \end{aligned}$$

Il primo vincolo seleziona $n-1$ spigoli; il secondo impedisce la presenza di cicli. Insieme garantiscono che gli spigoli selezionati formino un albero ricoprente.

```
while len(attributi) != 0:
    i = min(attributi, key=lambda k: attributi[k][0])
    if i != nodo_iniziale:
        spigoli_albero.append(tuple(sorted((attributi[i][1], i))))
        costo_totale += attributi[i][0]
    for j in vertici2:
        if (i, j) in Spigoli or (j, i) in Spigoli:
            if attributi[j][0] > dizionario_costi[tuple(sorted((i, j)))] :
                attributi[j][0] = dizionario_costi[tuple(sorted((i, j)))]
                attributi[j][1] = i

    attributi.pop(i)
    vertici2.remove(i)
```

Esempio. Sul file `3_grafo_costi_positivi.txt`, l'albero restituito contiene otto spigoli e ha costo totale 114.

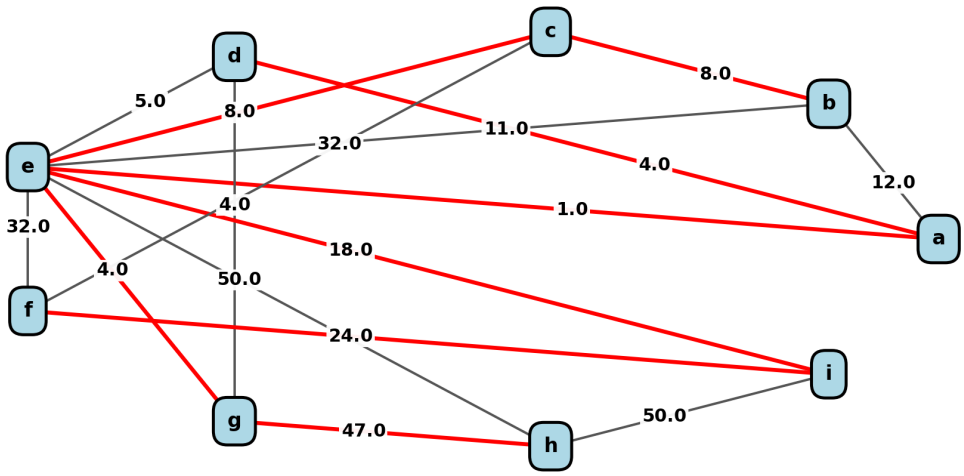


Figura 6: Albero ricoprente minimo sul file `3_grafo_costi_positivi.txt`. Le etichette sono i costi; gli spigoli rossi costano in totale 114.

6 Cammini minimi

Dato un digrafo $D = (N, A)$, con costi c_{ij} e sorgente s, si cercano i cammini di costo minimo da s agli altri nodi oppure a una destinazione t.

La funzione `Cammini_minimimi` seleziona l'algoritmo in base alla struttura della rete e ai costi. Il seguente estratto riporta le condizioni e le chiamate, omettendo le istruzioni intermedie di stampa e ricostruzione del cammino:

```
aciclico , ordinamento = Ordinamento_Topologico(nodi, archi, nodo_iniziale = nodo_iniziale)
if aciclico == True:
    albero_cammini_minimi, costi_totali = Camminimi_Minimi_acicilica(archi, costi, ordinamento)
elif all(costo >= 0 for costo in costi):
    albero_cammini_minimi, costi_totali = Camminimi_Minimi_Dijkstra(nodi, archi, costi,
    nodo_iniziale)
else:
    albero_cammini_minimi, costi_totali = Bellman_Ford(nodi, archi, costi, nodo_iniziale)
```

Digrafo aciclico Su un digrafo aciclico, i nodi vengono elaborati secondo un ordinamento topologico. Per ciascun nodo si determina il collegamento entrante che minimizza il costo complessivo dalla sorgente:

```
for k in range (1, len(ordinamento)):
    nodo_da_aggiungere = ordinamento[k]
    dizionario_costi_interni = {}
    for nodo in nodi:
        if (nodo,nodo_da_aggiungere) in archi:
            dizionario_costi_interni[(nodo,nodo_da_aggiungere)] = costi[archi.index((nodo,
    nodo_da_aggiungere))] + attributi[nodo][0]
    (pred,nodo_da_aggiungere) = min(dizionario_costi_interni, key=dizionario_costi_interni.get)
    albero_cammini_minimi.append((pred,nodo_da_aggiungere))
```

Esempio. Nel file `4_digrafo_semplice(aciclico).txt` i costi sono assenti e vengono assunti unitari. Dalla sorgente 1, il solver restituisce l'albero dei cammini minimi verso tutti gli altri nodi. Le distanze sono $d_2 = d_3 = d_4 = 1$, $d_5 = d_6 = 2$, $d_7 = d_8 = 3$.

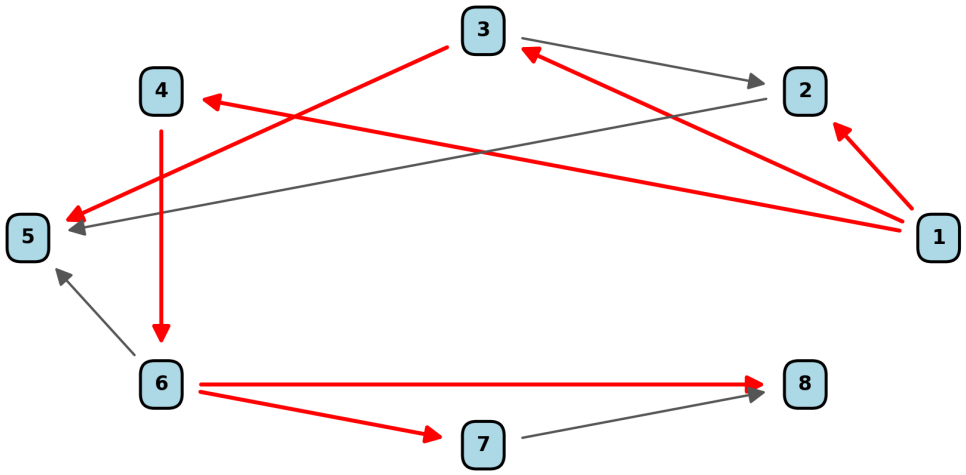


Figura 7: Albero dei cammini minimi dalla sorgente 1 nel file aciclico originale. Ogni arco ha costo unitario; gli archi rossi appartengono all'albero.

Dijkstra Se il digrafo è ciclico e i costi sono non negativi, viene utilizzato Dijkstra. A ogni iterazione si seleziona il nodo con etichetta minima e si aggiornano le etichette dei nodi raggiungibili

attraverso i suoi archi uscenti.

```
while len(nodi_copia) != 0:
    i = min(attributi, key=lambda k: attributi[k][0])
    if i != nodo_iniziale:
        albero_cammini_minimi.append((attributi[i][1], i))
        costo_totale += attributi[i][0]
    for j in nodi_copia:
        if (i, j) in archi:
            if attributi[j][0] > dizionario_costi[(i, j)] + attributi[i][0]:
                attributi[j][0] = dizionario_costi[(i, j)] + attributi[i][0]
                attributi[j][1] = i

    attributi.pop(i)
    nodi_copia.remove(i)
```

Esempio. Sul file `5_digrafo_costi_positivi.txt`, dalla sorgente 0, si ottengono le distanze $d_1 = 3$, $d_2 = 2$, $d_3 = 5$, $d_4 = 6$. Il cammino restituito da 0 a 4 è $0 \to 2 \to 1 \to 4$, di costo 6.

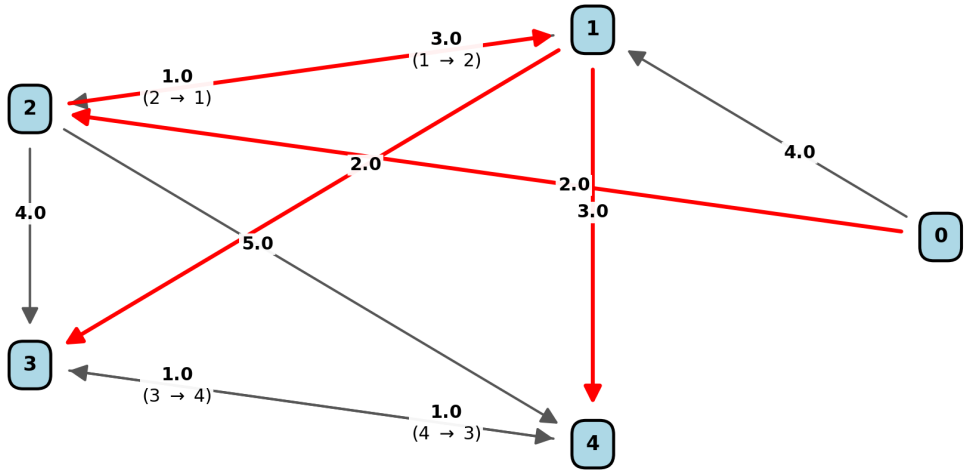


Figura 8: Albero dei cammini minimi da 0 sul file `5_digrafo_costi_positivi.txt`. Le etichette indicano i costi.

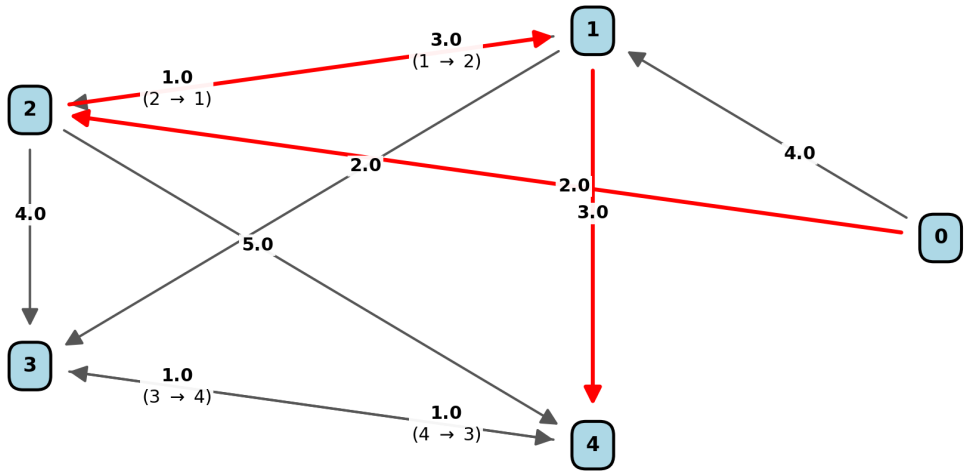


Figura 9: Cammino minimo da 0 a 4 sul medesimo file, di costo 6. Per gli archi opposti letichetta specifica anche il verso.

Bellman–Ford Se il digrafo è ciclico e presenta costi negativi, viene utilizzato l'algoritmo di Bellman–Ford. Quando un'etichetta migliora, il nodo viene inserito nella coda per propagarne l'aggiornamento ai successori.

```
while len(codaQ) != 0:
    i = codaQ[0]
    codaQ.remove(i)

    nodo_etichetta_min = min(attributi, key=lambda k: attributi[k][0])
    etichetta_min = attributi[nodo_etichetta_min][0]
    if etichetta_min < (len(nodi)-1)*  Costomin:
        return [], "-inf"

    for j in nodi:
        if (i,j) in archi and j!=nodo_iniziale:
            if attributi[j][0] > dizionario_costi[(i, j)] + attributi[i][0]:
                attributi[j][0] = dizionario_costi[(i, j)] + attributi[i][0]
                attributi[j][1] = i
                codaQ.append(j)


for nodo in attributi:
    if attributi[nodo][1] != nodo:
        costo_totale += attributi[nodo][0]
        pred = attributi[nodo][1]
        albero_cammini_minimi.append((pred, nodo))
```

Esempio. Il file `6_digrafo_costi_negativi.txt` contiene costi negativi e cicli, ma nessun ciclo di costo negativo. Dalla sorgente s si ottengono le distanze $d_a = 1$, $d_b = 3$, $d_c = 2$, $d_d = 6$, $d_e = 1$, $d_f = 4$. L'albero restituito è

$$T = \{(s, a), (a, d), (a, c), (c, b), (c, e), (e, f)\}.$$

Il valore restituito dalla funzione è 17, pari alla somma delle distanze dalla sorgente, mentre il costo totale degli archi dell'albero è 9.

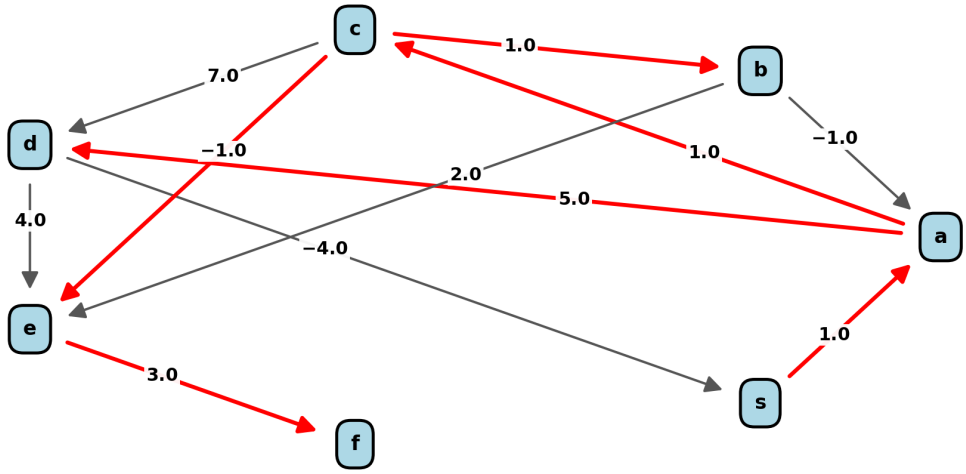


Figura 10: Albero dei cammini minimi dalla sorgente s sul file `6_digrafo_costi_negativi.txt`. Gli archi rossi appartengono all'albero; le etichette indicano i costi.

7 Problemi di flusso

7.1 Massimo flusso

Data una rete orientata con sorgente s, pozzo t e capacità massime u_{ij}, si cerca il massimo flusso ammissibile da s a t. La variabile x_{ij} indica il flusso sull'arco (i,j); $\Gamma^+(i)$ e $\Gamma^-(i)$ indicano rispettivamente la stella uscente ed entrante del nodo i. Indicando con v il valore del flusso:

$$\max v$$

$$\sum_{(i,j)\in\Gamma^+(i)} x_{ij} - \sum_{(h,i)\in\Gamma^-(i)} x_{hi} = \begin{cases} v & i = s, \\ -v & i = t, \\ 0 & i \in N \setminus \{s,t\}, \end{cases} \qquad 0 \le x_{ij} \le u_{ij} \quad ((i,j) \in A).$$

La funzione `Ford_Fulkerson` parte dal flusso nullo e cerca cammini aumentanti nella rete residua. Ogni cammino permette di incrementare il flusso della minima capacità residua incontrata.

```
while len(L) != 0 and nodo_pozzo not in S:
    i = L[0]
    L.remove(i)
    for j in nodi:
        if (i,j) in archi:
            if j not in S and dizionario_capacità[(i,j)] > dizionario_flussi[(i,j)]:
                S.append(j)
                L.append(j)
                etichette[j] = min(etichette[i],dizionario_capacità[(i,j)] - dizionario_flussi[(i,j
    )])
                dizionario_predecessori[j]=i
                versi[(i,j)] = "+"
    for j in nodi:
        if (j,i) in archi:
            if j not in S and dizionario_flussi[(j,i)]>0:
                S.append(j)
                L.append(j)
                etichette[j] = dizionario_flussi[(j,i)]
                dizionario_predecessori[j] = i
                versi[(i, j)] = "-"

if nodo_pozzo in S:
    archi_cammino = Traccia_Cammino(nodo_pozzo,dizionario_predecessori)
    dizionario_flussi = Aumenta_Flusso(archi_cammino,versi,etichette[nodo_pozzo],dizionario_flussi)
    flusso_totale += etichette[nodo_pozzo]
```

Esempio. Sul file `7_rete_max_flusso.txt`, dalla sorgente o al pozzo d, si ottiene un flusso di valore 7. Il taglio uscente dalla sorgente ha capacità $1 + 2 + 4 = 7$, che ne certifica l'ottimalità.

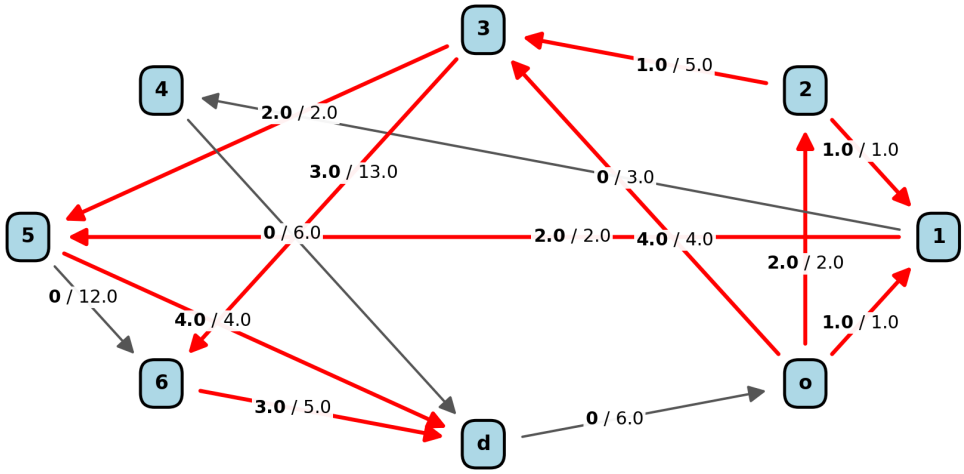


Figura 11: Massimo flusso sul file `7_rete_max_flusso.txt`. Etichette: flusso / capacità massima. Sono rossi gli archi con flusso positivo.

7.2 Flusso a costo minimo

Data una rete orientata, si cerca una distribuzione di flusso che soddisfi le offerte e le domande dei nodi, rispetti le capacità e minimizzi il costo totale. La variabile x_{ij} indica il flusso sull'arco; c_{ij} è il costo unitario, ℓ_{ij} e u_{ij} sono le capacità minima e massima. Il parametro b_i indica l'offerta se positivo e la domanda, cambiata di segno, se negativo.

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij}$$

$$\sum_{(i,j)\in\Gamma^+(i)} x_{ij} - \sum_{(h,i)\in\Gamma^-(i)} x_{hi} = b_i \quad (i \in N), \qquad \ell_{ij} \le x_{ij} \le u_{ij} \quad ((i,j) \in A).$$

```
x = cp.Variable(len(archi))
funzione_obiettivo = cp.Minimize(c @ x)
if cap_max != []:
    vincolo = [A @ x == b , cap_min <= x, x <= cap_max]
else:
    vincolo = [A @ x == b , cap_min <= x]
problem = cp.Problem(funzione_obiettivo, vincolo)
problem.solve()
```

Nel codice, `A` è la matrice di incidenza del digrafo: il vincolo `A@x==b` esprime i bilanci dei nodi.

Esempio. Sul file `8_flusso_costo_minimo.txt`, il solver restituisce una distribuzione di costo 214. Alcuni flussi sono frazionari: il modello è continuo e questa istanza ammette più soluzioni ottime.

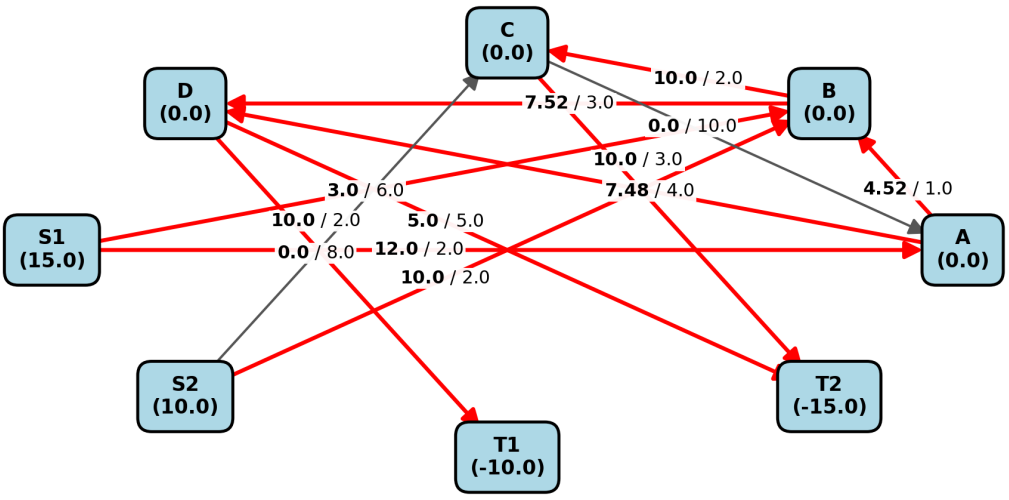


Figura 12: Flusso a costo minimo sul file `8_flusso_costo_minimo.txt`. Etichette sugli archi: flusso / costo; nei nodi: divergenza. Le capacità restano quelle del file.

7.3 Trasporto

Il problema del trasporto determina le quantità da inviare dalle origini alle destinazioni, soddisfacendo offerte e domande al minimo costo. Siano O le origini e P le destinazioni. La variabile x_{ij} indica la quantità trasportata da $i \in O$ a $j \in P$; a_i è l'offerta e d_j la domanda.

$$\min \sum_{i \in O} \sum_{j \in P} c_{ij} x_{ij}$$

$$\sum_{j \in P} x_{ij} \leq a_i \quad (i \in O), \qquad \sum_{i \in O} x_{ij} = d_j \quad (j \in P), \qquad x_{ij} \geq 0.$$

Il programma lo risolve tramite `Flusso_a_costo_min` su una rete bipartita, assegnando divergenze positive alle origini e negative alle destinazioni. La chiamata sul file di esempio è:

```
tipo, nodi, archi, costi, u, l, b = readInput_daFile(
    "9_Problema_di_Trasporto.txt")
flussi, costo, dizionario = Flusso_a_costo_min(
    nodi, archi, costi, b, cap_max=u, cap_min=l)
```

Esempio. Nel file `9_Problema_di_Trasporto.txt`, il nodo `Fittizio` assorbe le dieci unità di offerta eccedenti. La distribuzione restituita ha costo 135.

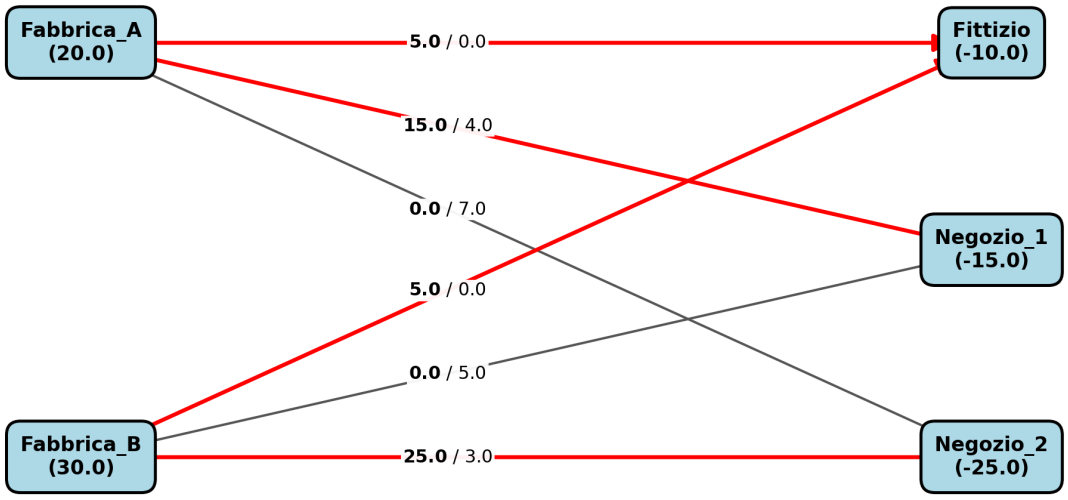


Figura 13: Soluzione del file `9_Problema_di_Trasporto.txt`, di costo 135. Etichette: flusso / costo; nei nodi: offerta o domanda con segno.

8 Controlli effettuati dal solver

8.1 Ammissibilità e coerenza della richiesta

- Verifica la compatibilità del problema con il tipo di rete e la presenza degli attributi necessari: costi, capacità e divergenze.
- Controlla che i nodi indicati appartengano alla rete e verifica la raggiungibilità richiesta per i cammini minimi.
- Prima di calcolare l'albero ricoprente minimo, verifica la connessione del grafo.
- Per i cammini minimi, controlla aciclicità e segno dei costi per scegliere l'algoritmo; Bellman–Ford prevede inoltre una segnalazione dei cicli negativi.

8.2 Validità dell'input

- Controlla i cinque indicatori binari dell'intestazione e la corrispondenza tra attributi dichiarati e campi delle righe.
- Intercetta gli errori di conversione numerica di costi, capacità e divergenze.
- Verifica che le divergenze siano specificate per tutti i nodi e riferite a nodi presenti nella rete.
- Segnala l'input vuoto e i comandi non validi.

8.3 Gestione dell'interazione

- Gestisce l'annullamento del caricamento e dell'inserimento testuale.
- Interrompe la visualizzazione oltre 80 nodi o 100 archi.
- In caso di errore di esecuzione, mostra un messaggio diagnostico e permette il riavvio.