Perfect Matching

Marco Daresta, Chiara Langella,
Elisa Pontiroli and Alessandro Zeggiotti

2017-2018

Introduzione

La teoria dei grafi ha una data di nascita precisa: il 1736. In quella data, il matematico svizzero Leonhard Euler risolse il problema noto come i sette ponti di Königsberg. Ci si chiedeva se fosse possibile fare una passeggiata in città, che partisse e arrivasse allo stesso punto, in modo da attraversare tutti i ponti esattamente una volta.

Successivamente, nel 1859 Hamilton propose un gioco che, per diversi aspetti, era legato alla sua teoria dei quaternioni: il gioco dell'icosaedro (che in realtà si giocava su un dodecaedro) era il seguente: Hamilton aveva assegnato a ogni vertice il nome di una città e richiedeva di trovare un percorso che facesse il giro del mondo, ossia visitasse tutte le città una sola volta, per poi tornare al vertice di partenza.

Una variante del gioco dell'icosaedro è il problema del commesso viaggiatore (TSP). Si tratta di trovare il percorso chiuso più breve in un grafo completo pesato, ossia i cui lati hanno lunghezze diverse. Questo è il problema per eccellenza nella ottimizzazione combinatoria.

Non è un problema trovare un circuito chiuso: in un grafo completo a n nodi esistono $\frac{1}{2}(n-1)!$ circuiti chiusi. Il problema è trovare il migliore.

Trovare un algoritmo che possa risolvere ogni esempio di TSP sarebbe un cambio di orizzonte importante in matematica: usando questo metodo, saremmo in grado di risolvere efficientemente ogni problema computazionale per cui la risposta sia facilmente verificabile. Molti lo ritengono impossibile.

In particolare, in matematica, informatica e, precisamente, geometria combinatoria, la teoria dei grafi si occupa di studiare i grafi, che sono oggetti discreti che permettono di schematizzare una grande varietà di situazioni e di processi e spesso di consentirne delle analisi in termini quantitativi e algoritmici.

Per grafo si intende una struttura costituita da:

- oggetti semplici, detti vertici o nodi;
- collegamenti tra i vertici; tali collegamenti possono essere:

- non orientati (cioè dotati di una direzione, ma non dotati di un verso): in questo caso sono detti spigoli, e il grafo è detto "non orientato";
- orientati (cioè dotati di una direzione e di un verso): in questo caso sono detti archi o cammini, e il grafo è detto "orientato" o digrafo;
- eventuali dati associati a nodi e/o collegamenti; un grafo pesato è un esempio di grafo in cui a ogni collegamento è associato un valore numerico, detto "peso".

Un grafo viene generalmente raffigurato sul piano da punti o cerchi, che rappresentano i nodi; i collegamenti tra i vertici sono rappresentati da segmenti o curve che collegano due nodi; mentre, nel caso di un grafo orientato, il verso degli archi è indicato da una freccia. Lo stesso grafo può essere disegnato in molti modi diversi senza modificarne le proprietà.

Le strutture che possono essere rappresentate da grafi sono presenti in molte discipline e molti problemi di interesse pratico possono essere formulati come questioni relative a grafi. In particolare, le reti possono essere descritte in forma di grafi. I grafi orientati sono anche utilizzati per rappresentare le macchine a stati finiti e molti altri formalismi, come ad esempio diagrammi di flusso, catene di Markov, schemi entità-relazione e reti di Petri.

Lo sviluppo di algoritmi per manipolare i grafi è una delle aree di maggiore interesse dell'informatica.

Nell'ambito della Ricerca Operativa si vanno a risolvere problemi di minimo (e viceversa di massimo) sotto opportune restrizioni poste dal problema preso in esame e con particolari metodi che sono tuttora oggetto di studio. Si parla comunque quasi sempre di ottimizzazione di un problema piuttosto complesso. Gli algoritmi creati per la risoluzione di problemi all'apparenza irrisolubili costituiscono l'ossatura di tutta la Ricerca Operativa e semplici ragionamenti possono essere pensati da potenti computers per la risoluzione di problemi con centinaia o migliaia di variabili.

Tornando però ad analizzare la Teoria dei Grafi, a differenza di molte altri rami della Ricerca Operativa, questa opera sicuramente sotto la visualizzazione grafica di archi, nodi e flussi. Si nota che qualsiasi problema di Grafi e Reti apparentemente descrivibile solo in forma grafica, ha invece una sua possibile descrizione matematica e in particolare, una formulazione di programmazione lineare, lineare intera o non lineare.

In questo scritto tratteremo in particolare, dopo un breve capitolo introduttivo sui grafi e sul problema del Perfect Matching, volto a dare chiarimenti su definizioni e teoremi per una maggiore comprensione e una visione generale del problema, di un algoritmo per la risoluzione del problema del Perfect

Matching usando sia il linguaggio di Gurobi che di Python, correlato da alcuni esempi.

Indice

Capitolo 1

Introduzione al problema del Perfect Matching

La teoria dei grafi è una branca della matematica, in particolare della geometria combinatoria, che si occupa dello studio dei grafi, i quali sono oggetti che permettono di schematizzare una grande varietà di situazioni e di processi e spesso di consentirne delle analisi in termini quantitativi e algoritmici. Essi hanno anche una notevole interesse nell'informatica, soprattutto nello sviluppo di algoritmi specifici. Tra i tanti problemi studiati in teoria dei grafi, uno dei più famosi è quello del Perfect Machting (in italiano "Accoppiamento perfetto"). Prima di introdurre il problema, è doveroso fare un piccolo richiamo su alcuni concetti base importanti che serviranno in seguito

1.1 Generalità sui grafi

Definizione 1.1. Si dice grafo una coppia ordinata $G = (V, E)$ di insiemi, con V insieme dei vertici (o nodi) ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di V.

L'ordine di un grafo, denotato con |V(G)|, indica il numero dei vertici di G; la dimensione di un grafo G, denotato con |E(G)|, indica il numero degli archi di un grafo G.

Un vertice $v \in V$ è incidente a un arco $e \in E$ se $v \in e$.
Due archi $e_1, e_2 \in E$ sono incidenti (o adiacenti) se hanno un vertice in comune.
Due vertici $v_1, v_2 \in V$ sono adiacenti se esiste un arco e $\in E$ tale che $e = v_1v_2$.

Definizione 1.2. Siano $G = (V, E)$ e $G' = (V', E')$ due grafi. Se $V' \subseteq V$ e $E' \subseteq E$, allora G' è un sottografo di G. Inoltre, se $G \neq G$, allora diciamo che G' è un sottografo proprio di G.

Definizione 1.3. Sia $G = (V, E)$ un grafo e sia $v \in V$ un vertice. Il grado di un vertice v è definito come

$$d_G(V) := |N_G(v)|$$

dove $N_G(v) := \{w \in V : vw \in E\}$ è detto intorno di v. In altre parole è il numero di archi indicenti nel vertice $v \in V$. Definiamo poi

$$\delta(G) := \min_{v \in V} d_G(v) \in \mathbb{Z}$$

il grado minimo di G, cioè il grado del vertice con meno archi incidenti e

$$\Delta(G) := \max_{v \in V} d_G(v) \in \mathbb{Z}$$

il grado massimo di G, cioè il grado del vertice con più archi incidenti. Inoltre, denotiamo il grado medio di G come

$$d(G) := \frac{1}{|V|} \sum_{v \in V} d_G(v)$$

Chiaramente si ha che $\delta(G) \leq d(G) \leq \Delta(G)$.

Definizione 1.4. Un grafo $G = (V, E)$ è detto K-regolare se

$$d_G = k \qquad \forall v \in V$$

In altre parole, se tutti i vertici hanno lo stesso grado. Di conseguenza, si ha $\delta = d = \Delta$. Nello specifico, i grafi 3-regolari sono detti anche grafi cubici.

Definizione 1.5. Un cammino è un grafo $P = (V, E)$ nella forma $V = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$ dove
x_0 e x_k sono i vertici esterni di P;
$x_2, \ldots, x_{k-1}$ sono i vertici interni di P;
Il numero di archi definisce la lunghezza di P;
Con la notazione P_k viene indicato un cammino di lunghezza k.

Definizione 1.6. Un grafo è detto completo se ogni vertice è collegato a tutti gli altri vertici rimanenti. Esso viene denotato con K_n (con $n \in N$ il numero dei vertici)

Definizione 1.7. Un grafo $G = (V, E)$ è bipartito se il suo insieme di vertici V può essere partizionato in due sottoinsiemi disgiunti $V = V_1 \cup V_2$ tali che ogni arco $e \in E$ ha la forma v_1v_2, con $v_1 \in V_1$ e $v_2 \in V_2$.

Definizione 1.8. Si definisce grafo planare un grafo G tale che può essere raffigurato in un piano in modo che non abbiano archi che si intersecano.
I grafi completi K_5 e $K_{3,3}$ sono esempi di grafi non planari.

Di seguito, vengono riportati alcuni esempi

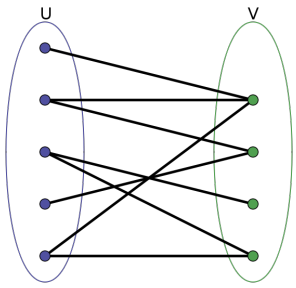

Figura 1.1: Grafo bipartito

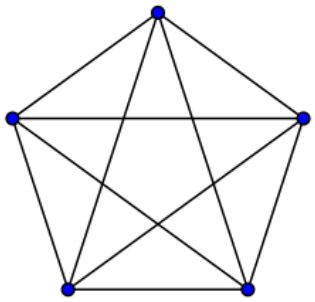

Figura 1.2: Grafo completo (K_5)

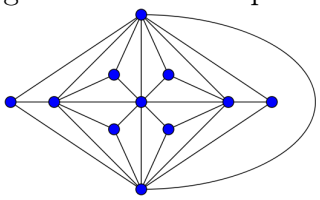

Figura 1.3: Grafo planare

1.2 Matching e Perfect Matching

Dato un certo grafo G, si vuole trovare quanti più archi indipendenti possibili.

Definizione 1.9. Dato un grafo $G = (V, E)$, un matching ("accoppiamento") $M(G)$ in G è un sottografo di G composto da un insieme di archi a coppie non incidenti, cioè non esistono due archi che condividono un vertice in comune. Oppure, in maniera equivalente, se ogni vertice di G è incidente con al massimo un arco in M, cioè $deg(v) \leq 1 \ \forall v \in G$.
Questo significa che, in un matching M(G), i vertici possono essere o di grado 1 o 0. In particolare,

- se deg(v) = 1, allora il vertice v è detto accoppiato (o saturato), cioè v è incidente in uno degli archi in M
- se deg(v) = 0, allora il vertice v non è accoppiato.

In un matching, due archi non sono incidenti: se lo fossero, allora il grado del vertice che collega questi due archi avrebbe grado 2, il che viola la definizione.

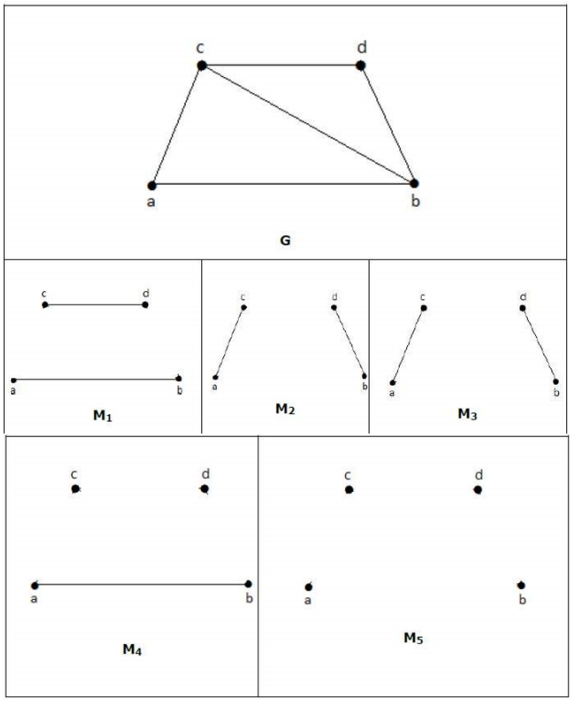

Figura 1.4: Esempi di Matching di un grafo G

Definizione 1.10. Un grafo $G = (V, E)$ è detto matching massimale se nessun arco di G può essere aggiunto ad un matching M. In altre parole, se aggiungiamo un qualsiasi arco (non in M) in M, esso non è più un matching.

Definizione 1.11. Dato un grafo $G = (V, E)$, M(G) è detto matching massimo se M contiene il massimo numero possibile di spigoli. Esso non è unico. Tale numero è detto numero matching. Notare che ogni matching massimo è massimale ma non il viceversa. La seguente figura mostra esempi di matching massimi

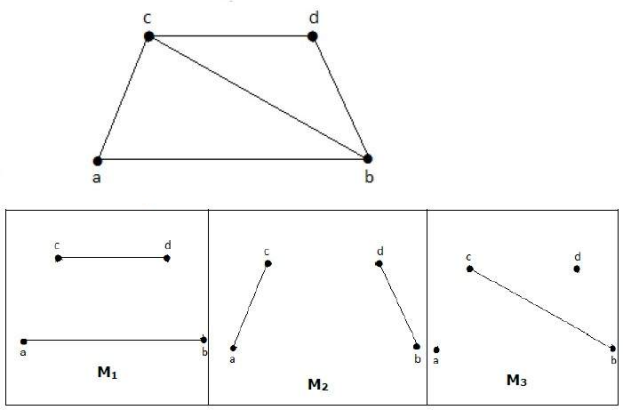

Figura 1.5: M_1, M_2, M_3 sono matching massimale per G

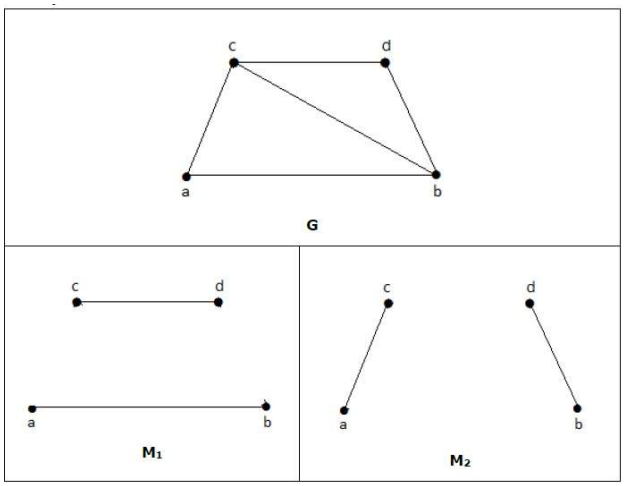

Figura 1.6: M_1 e M_2 sono matching massimi per G e il numero matching è 2. Quindi usando il grafo G, possiamo formare solo i sottografi con solo al massimo 2 archi. Per questo abbiamo 2 come numero matching

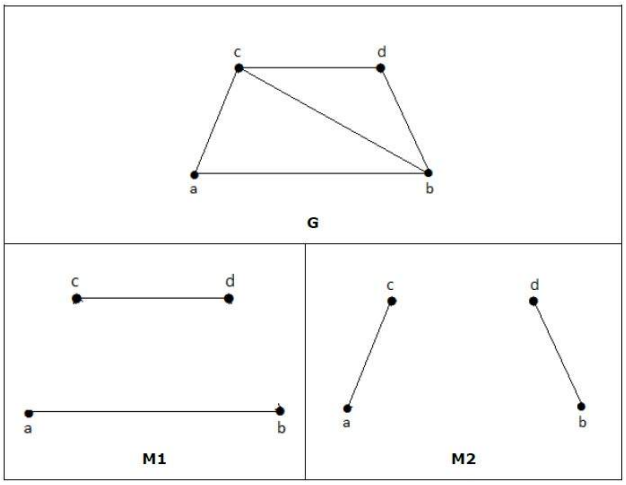

Figura 1.7: M_1 e M_2 sono esempi di matching perfetto per G

Definizione 1.12. Un matching M(G) di un grafo G è detto perfetto se ogni vertice $v \in G$ è incidente ad un arco $e \in M$, cioè se $deg(v) = 1 \forall v \in V$

Osservazione 1.13. Dalle definizioni sopra citate si nota che:

1. Ogni matching perfetto è anche un matching massimo perché non esiste alcuna possibilità di aggiungere un altro arco in più in un perfect matching. Di conseguenza, è anche massimale;

2. Un matching massimo non è necessariamente perfetto.

3. Se un grafo G ha un matching perfetto, allora il numero di vertici $|V(G)|$ è pari: se fosse dispari, allora l'ultimo vertice si accoppia con un altro e alla fine rimarrebbe un singolo vertice che non è abbinato a nessun altro e quindi avrebbe grado 0. Ma questo non è possibile per definizione di matching perfetto.

1.3 Alcuni esempi di Perfect Matching

Per modellare problemi di perfect matching, si utilizzano spesso grafi bipartiti. Si consideri $G = (V, E)$ un grafo bipartito con $\{A, B\}$ una bipartizione di V, cioè $V = A \cup B$ e $A \cap B = \emptyset$ e tutti archi connettono vertici tra A e B. L'obiettivo è trovare un matching M in G con quanti più archi possibili.

Per risolvere alcuni problemi riguardo a possibili combinazioni di cose, si utilizzando questioni e concetti legati alla teoria dei grafi. In questa sezione

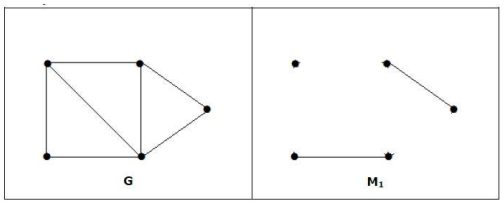

Figura 1.8: Matching non perfetto con un numero dispari di vertici

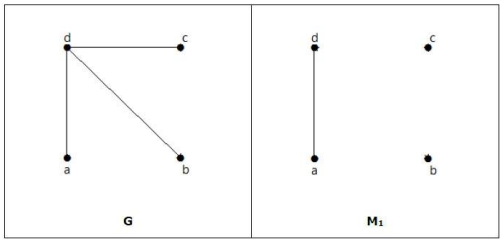

Figura 1.9: E' un matching ma non è perfetto sebbene abbia un numero pari di vertici

tratteremo alcuni semplici esempi di perfect matching e vedremo come si risolvono. A questo proposito, enunciamo un risultato importante che sta alla base di molte applicazioni, tra cui l'esempio che seguirà :

Teorema 1.14 (Hall, 1935)**.** *Un grafo bipartito G ha un matching A se e solo se*

$$|N(S)| \geq |S| \qquad \forall S \subseteq A$$

Dimostrazione. Si veda [1]. □

Esempio 1.15. Supponiamo di avere 6 regali (denotati con 1,2,3,4,5,6) da dare a 5 amici (Alice, Bob, Charles, Dot, Edward). La domanda è: posso distribuire un regalo ad ognuno in modo che tutti ottengano quello che desiderano?
Certamente, questo dipende dalle preferenze dei singoli amici. Se nessuno di loro piacciono i miei regali allora sono stato sfortunato. Anche se a tutti piacciono alcuni dei doni, non sono in grado di soddisfarli tutti: ad esempio se nessuno ama il regalo 5 o 6, allora avrà solo 4 regali da dare ai 5 amici e quindi il problema non è risolvibile. Escludiamo quindi questi casi.

Possiamo vedere questa situazione come un grafo partizionato in due insiemi: quello dei regali e quello degli amici. Per definizione, si vede subito che il grafo in questione è bipartito. Ad ogni persona è associato un arco che indica le proprie preferenze sui regali (ad esempio Alice preferisce ricevere il regalo 1 o 3 e così via). Verifichiamo se soddisfa o meno il Teorema di

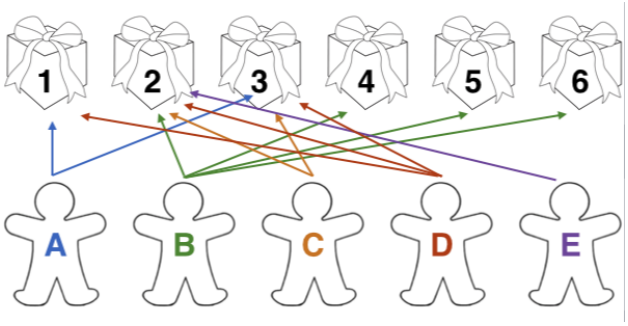

Figura 1.10: Può ogni bambino ricevere un regalo che preferisce?

Hall: consideriamo come sottoinsieme $X = \{A, C, D, E\}$, quindi $|X| = 4$ e di conseguenza $N(X) = \{1, 2, 3\}$, quindi $|X| = 3$. Da ciò si evince che $|X| \geq |N(X)|$ pertanto la condizione di Hall non è soddisfatta e quindi non esiste un matching. In altre parole, non è possibile distribuire a tutti il regalo desiderato e quindi accontentare tutti.

Un'altro esempio è quello del problema del Vertex Cover.

Definizione 1.16. Sia $G = (V, E)$ un grafo. Una copertura di vertici di G è un sottoinsieme $U \subseteq V$ tale che ogni arco $e \in E$ è incidente a un vertice $v \in U$.

Questo è uno dei tanti argomenti sulla teoria dei grafi ed ha molte applicazioni nei problemi di perfect matching e problemi di ottimizzazione. La copertura di vertici può essere un buon approccio per un problema dove tutti gli archi di un grafo devono essere inclusi nella soluzione. In particolare, spesso viene chiesto di trovare il coprimento con il numero più piccolo di vertici.

Osservazione 1.17. Alcune proprietà del Vertex Cover:

- L'insieme di tutti i vertici è un coprimento di vertici
- Gli endpoints di un matching massimale formano un vertex cover
- Il grafo bipartito completo $K_{m,n}$ ha un vertex cover minimo dato da $min\{m, n\}$.

Il seguente teorema stabilisce una relazione tra vertex cover e perfect matching. In particolare

Teorema 1.18 (König, 1931)**.** *In un grafo G bipartito, il numero di archi di un matching massimale è uguale al numero di vertici in un vertex cover minimo.*

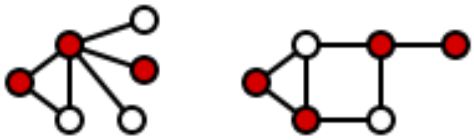

Figura 1.11: Esempi di vertex cover segnati in rosso

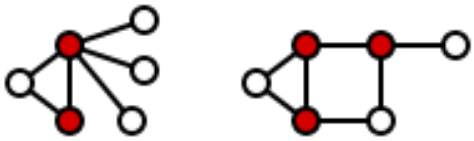

Figura 1.12: Esempi di vertex cover minimo

Dimostrazione. Si veda [1]. □

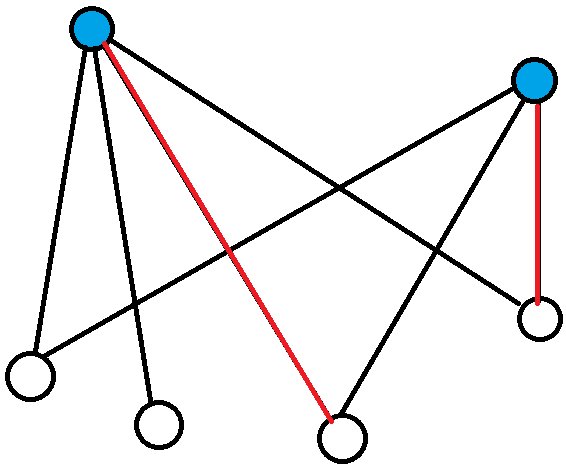

Figura 1.13: Esempio di applicazione del Teorema di König: i vertici rossi rappresentano il vertex cover minimo mentre gli archi blu indicano un matching massimo

In informatica, ci sono algoritmi che implementano questo problema.

Capitolo 2

Algoritmo

Prima di spiegare tutti i dettagli dell'algoritmo che abbiamo sviluppato, è necessario introdurre alcuni concetti teorici.

Si consideri una grafo $G = (V, E)$, definito come coppia di insiemi: V per i vertici, E per gli archi; $\delta_G(S)$ (or $\delta(S)$) è l'insieme di quegli archi in E con esattamente un estremo S, sottoinsieme di V. Definiamo un *innesto* (G, T) dato da un grafo connesso G[1] in cui un numero pari di vertici $T \subseteq V$ vengono distinti come *dispari*. La *T-parità* di un insieme di vertici $S \subseteq V$ è la parità di $|S \cap T|$. Quando $S \subseteq V$ è T-dispari allora $\delta(S)$ un *taglio T-dispari* or *T-taglio*.

Al fine di avere una descrizione completa dell'algoritmo sviluppato, consideriamo il seguente concetto di **albero di Gomory-Hu**[2] di un grafo indiretto, ovvero un grafo non orientato, è un albero pesato che rappresenta il minimo $\{s, t\}$-taglio per ogni coppia $\{s, t\}$ nel grafo. L'albero di Gomory-Hu può essere costruito calcolando $|V| - 1$ volte il massimo flusso.

Definizione 2.1. Sia $G = ((V_G, E_G), c)$ un grafo indiretto in cui $c(u, v)$ rappresenta la capacità dell'arco (u, v) rispettivamente. Denotiamo la capacità minima di un $\{s, t\}$-taglio con $\lambda_{s,t}$ per ogni $\{s, t\} \in V_G$. Sia $T = (V_T, E_T)$ un albero con $V_T = V_G$, denotiamo l'insieme di archi presenti in un cammino $\{s, t\}$ con $P_{s,t}$ per ogni $\{s, t\} \in V_T$. Allora T è detto un **albero di Gomory-Hu** di G ise

$$\lambda_{s,t} = \min_{e \in P_{s,t}} c(S_e, T_e) \quad \text{per ogni} \quad \{s, t\} \in V_G,$$

[1]Un grafo $G = (V, E)$ è detto *connesso* se, per ogni coppia di vertici $(u, v) \in V$ esiste un cammino che collega u a v. Un sottografo connesso massimale di un grafo non orientato è detto *componente connessa* di tale grafo.

[2]Ricordiamo che un grafo aciclico è chiamato *foresta* e una foresta connessa è detta **albero**

dove

1. S_e e T_e sono due componenti connesse di $T \setminus \{e\}$, nel senso che (S_e, T_e) forma un s,t-taglio in G.

2. $c(S_e, T_e)$ è la capacità di taglio in G.

Sia (G,T) un innesto e sia $c : E \to \mathbb{R}_+$ una funzione di *costo*. Un minimo T-taglio per (G,T,c) è un T-taglio $\delta(X)$ di (G,T) per cui:

$$c(\delta(X)) = \lambda_{G,T} = \min\{c(\delta(S)) : \delta(S) \quad \text{è un } T\text{-taglio di} \quad (G,T)\}$$

dove il costo $c(F)$ di un insieme di archi F è defiinito come $\sum_{e \in F} c(e)$. In particolare, ricordiamo alcune nozioni basilari su submodularity e uncrossing. Il complementare di $S \subseteq V$ in V è definito come $\bar{S} = V \setminus S$. *Switching* S significa rimpiazzare S con $\bar{S}$. Per esempio, se $S = X$, allora dopo lo switching di S, otteniamo che $S = \bar{X}$ e $\bar{S} = X$.
Siano (G,T) un innesto e $S \subseteq V$ un insieme.

Osservazione 2.2. Switching S non cambia la T-parità di S (nemmeno $\delta(S)$), dato che $|S \cap T|$ e $|\bar{S} \cap T|$ hanno la stessa parità poiché $|T|$ è pari.

Proposizione 2.3. *Siano (G,T) un innesto e $S, X \subseteq V$ sottoinsiemi di V. Allora abbiamo:*

1. *Switching S cambia la T-parità di $S \cap X$ se e solo se X è T-dispari.*

2. *$S \cap X$ e $S \cup X$ hanno la stessa T-parità se e solo se S e X hanno la stessa T-parità.*

3. *Switching S cambia la T-parità di $S \cup X$ se e solo se X è T-dispari.*

Dimostrazione. Si noti che $|(S \cap X) \cap T| = |S \cap (X \cap T)|$ la cui parità è influenzata dallo switching di S se e solo se $|X \cap T|$ è dispari, ovvero, se e solo se X è T-dispari. Questo prova (1). Per ottenere (2) si noti che $|(S \cap X) \cap T| + |(S \cup X) \cap T| = |S \cap T| + |X \cap T|$. In fine, (3) è una conseguenza di (1) e (2). □

I lemmi che seguono caratterizzano una proprietà dei tagli, conosciuta come *submodularity*.

Lemma 2.4. *Dato G un grafo con funzione di costo $c : E \longmapsto \mathbb{R}_+$. Siano $S_1, S_2 \subseteq V$.*

$$c(\delta(S_1 \cap S_2)) + c(\delta(S_1 \cup S_2)) \leq c(\delta(S_1)) + c(\delta(S_2)) \tag{2.1}$$

Dimostrazione. Supponiamo che ogni arco uv contribuisca sia al lato destro come a quello sinistro di (2.1). Dalla Propositione 2.2, se $S_1 \cap S_2$ ha diverse $\{u, v\}$-parità, lo stesso vale per S_1 e S_2. Quindi, avendo supposto il falso, allora entrambi $S_1 \cap S_2$ e $S_1 \cup S_2$ saranno $\{u, v\}$-dispari. Assumiamo senza perdita di generalità che $u \in S_1 \cup S_2$ e $u \notin S_1 \cup S_2$. Allora, $u \in S_1, S_2$ e $u \notin S_1, S_2$ □

Se $S \cap X \neq$ per ogni possibile switching di S e X, allora si dice che S e X si *incrociano*. Ora tutto ciò di cui abbiamo bisogno è che le funzioni di taglio obbediscano al seguente lemma.

Lemma 2.5. *Siano T_1, T_2 sottoinsiemi di V, aventi cardinalità pari. Sia $\delta(S_1)$ un minimo T_1-taglio e assumiamo che S_1 sia T_2-pari. Allora esiste un minimo T_2-taglio $\delta(S_2)$ tale che S_1 e S_2 non si incrociano.*

Dimostrazione. Sia $\delta(X)$ un minimo T_2-taglio. Ricordiamo che, dalla Proposizione 2.2, lo switching di S_1 cambia la T_2-parità di $S_1 \cup X$ mentre lo switching di X cambia la T_1-parità di $S_1 \cap X$ lasciando la T_2-parità di $S_1 \cup X$ intatta. Pertanto, attuando possibilmente lo switching di S_1, è possibile assumere che $S_1 \cup X$ è T_2-dispari. In seguito, attuando possibilmente lo switching di X, è possibile assumere che $S_1 \cap X$ is T_1-dispari non modificando la T_2-parità di $S_1 \cup X$. A questo punto, $c(\delta(S_1 \cup X)) \geq c(\delta(S_1))$ dato che $\delta(S_1)$ è un minimo T_1-taglio. Grazie alla proprietà di sub-modularity, $c(\delta(S_1 \cup X)) = c(\delta(X))$. Così, $\delta(S_1 \cup X)$ è un minimo T_2-taglio. E chiaramente, S_1 e $S_1 \cup X$ non si incrociano. □

2.1 Minimo T-taglio: un algoritmo semplice

Ora, consideriamo un T-taglio. Consideriamo un innesto (G, T) e un insieme $S \subseteq V$ che sia T-pari, indichiamo con G_S il grafo ottenuto da G identificando tutti i nodi di S con un unico nodo e definiamo $T_S := T \setminus S$. Si noti che (G_S, T_S) sia un innesto. Quando $S = \{s, t\} \subseteq T$ allora facciamo uso della seguente notazione $G_{s,t} = G_{\{s,t\}}$ and $T_{s,t} = T_{\{s,t\}}$.
Dato che l'identificazione di un nodo non influenza l'insieme di archi di un grafo, una funzione di costo c per G è a sua volta una funzione di costo per G_S e $G_{s,t}$. Allora, qui di seguito sono illustrati i quattro passi per calcolare

$\lambda_{G,T}$:

`MinT-cut(G,T,c)`

1. Se $T =$ allora risulta ∞, che significa che (G, T) non contiene alcun T-taglio;

2. siano s e t due nodi differenti in T;

3. sia $\delta(S)$ un minimo $\{s, t\}$-taglio;

4. se S è T-dispari allora viene restituito $\min(c(\delta(S)), \texttt{MinT-cut}(G_{s,t}, T_{s,t}, \texttt{c}))$; altrimenti $\min(\texttt{MinT-cut}(G_S, T_S, \texttt{c}); \texttt{MinT-cut}(G_{\bar{S}}, T_{\bar{S}}, \texttt{c}))$

2.2 Correttezza

Dato (G, T, c), siano s e t due differenti nodi in T e sia $\delta(S)$ un minimo $\{s, t\}$-taglio. La correttezza della procedura sopra descritta si basa sui seguenti due lemmi:

Lemma 2.6. *Se $\delta(S)$ è T-dispari, allora $\lambda_{G,T} = \min(c(\delta(S)), \lambda_{G_{s,t},T_{s,t}})$.*

Dimostrazione. Infatti, i $T_{s,t}$-tagli di $G_{s,t}$ sono esattamente i T-tagli di G che non sono $\{s, t\}$-dispari. □

Lemma 2.7. *Se $\delta(S)$ è T-pari, allora $\lambda_{G,T} = \min(\lambda_{G_S,T_S}, \lambda_{G_{\bar{S}},T_{\bar{S}}})$.*

Dimostrazione. Per prima cosa si noti che ogni T_S-taglio in (G_S, T_S) e ogni $T_{\bar{S}}$-taglio in $(G_{\bar{S}}, T_{\bar{S}})$ è anche un T-taglio in (G, T). Questo implica che $\lambda_{G,T} \leq \min(\lambda_{G_S,T_S}, \lambda_{G_{\bar{S}},T_{\bar{S}}})$. D'altra parte, sia $\delta(X)$ un minimo T-taglio per (G, T, c). Dal Lemma 2.4, possiamo assumere che S e X non si incrociano. Questo significa che l'insieme degli archi $\delta_G(X)$ è sia T_S-taglio in G_S che un $T_{\bar{S}}$-taglio in $G_{\bar{S}}$. □

2.3 Calcolo di T-parings ottimali

Sia (G, T) un innesto con funzione di costo $c : E \longmapsto \mathbb{R}_+$. Un *$T$-pairing* è una partizione di T in coppie. Il valore del T-pairing $\mathcal{P}$ è definita come:

$$val_G(\mathcal{P}) = \min_{\{u,v\} \in \mathcal{P}} \lambda_G(u, v)$$

dove $\lambda_G(u,v)$ identifica il costo di un minimo $\{u,v\}$-taglio, Sia $\mathcal{P}$ un T-pairing e sia $\delta(S)$ un T-taglio. Dato che $\delta(S)$ è T-dispari, $\mathcal{P}$ contiene una coppia $\{u,v\}$ tale che $\delta(S)$ è $\{u,v\}$-dispari. Pertanto, $c(\delta(S)) \geq \lambda_G(u,v) \geq val_G(\mathcal{P})$ e il valore di $\mathcal{P}$ rappresenta un limite inferiore di $\lambda_{G,T}$.
In questa sezione, mostriamo che l'algoritmo *MinT-cut* trova un T-pairing di valore $\lambda_{G,T}$.A tal proposito, consideriamo una sola iterazione dell'algoritmo. Siano s e t due nodi dispari. Sia $\delta(S)$ un minimo $\{s,t\}$-taglio.

Lemma 2.8. *Sia $\delta(S)$ un minimo $\{s,t\}$-taglio in (G,c). Allora abbiamo:*

$$\lambda_G(u,v) \geq \min(c(\delta(S)), \lambda_{G_{s,t}}(u,v)) \qquad \forall u,v \in V(G) \setminus \{s,t\}$$

Dimostrazione. Infatti, gli $\{u,v\}$-tagli di $G_{s,t}$ sono esattamente gli $\{u,v\}$-tagli di G che non sono $\{s,t\}$-dispari. □

Lemma 2.9. *Sia $\delta(S)$ un minimo $\{s,t\}$-taglio in (G,c). Allora abbiamo:*

$$\lambda_G(u,v) = \lambda_{G_s}(u,v) \qquad \forall u,v \in V(G) \setminus S$$

Dimostrazione. Siano u e v due nodi qualsiasi di $V(G) \setminus S$. Ovviamente $\lambda_G(u,v) \leq \lambda_{G_s}(u,v)$. D'altro canto, sia $\delta(X)$ un minimum $\{u,v\}$-taglio in G. Dal Lemma 2.4, possiamo assumere che S e X non si incrociano. allora, l'insieme degli archi $\delta_G(X)$ è un $\{u,v\}$-taglio in G_S e $\lambda_G(u,v) = \lambda_{G_s}(u,v)$. □

Ogni iterazione dell'algoritmo prevede due possibilità:

1. **Caso 1**: $\delta(S)$ è T-dispari. Dal Lemma 2.5, $\lambda_{G,T} = \min\{c(\delta(S)), \lambda_{G_{s,t},T_{s,t}}\}$. Dal Lemma 2.7, se $\mathcal{P}'$ è un $T_{s,t}$-pairing con $val_{G_{s,t}}(\mathcal{P}') = \lambda_{G_{s,t},T_{s,t}}$, allora $\mathcal{P} = \mathcal{P}' \cup \{s,t\}$ è un T-pairing con$val_G(\mathcal{P}) \geq \min(c(\delta(S)), val(\mathcal{P}')) = \min(c(\delta(S)), \lambda_{G_{s,t},T_{s,t}}) = \lambda_{G,T}$.

2. **Caso 2**: se $\delta(S)$ è T-pari. Dal Lemma 2.6, $\lambda_{G,T} = \min\ \lambda_{G_S,T_S}, \lambda_{G_{\bar{S}},T_{\bar{S}}}\}$. Dal Lemma 2.8, se $\mathcal{P}_\mathcal{S}$ è un T_S-pairing con $val_{G_S}(\mathcal{P}_\mathcal{S}) = \lambda_{G_S,T_S}$ e $\mathcal{P}_{\bar{\mathcal{S}}}$ è un $T_{\bar{S}}$-pairing con $val_{G_{\bar{S}}}(\mathcal{P}_{\bar{\mathcal{S}}}) = \lambda_{G_{\bar{S}},T_{\bar{S}}}$ allora $\mathcal{P} = \mathcal{P}_\mathcal{S} \cup \mathcal{P}_{\bar{\mathcal{S}}}$ è un T-pairing con $val_G(\mathcal{P}) \geq \min(val(\mathcal{P}_\mathcal{S}), val(\mathcal{P}_{\bar{\mathcal{S}}})) = \min(\lambda_{G_S,T_S}\lambda_{G_{\bar{S}},T_{\bar{S}}}) = \lambda_{G,T}$.

I risultati seguenti riassumono questa sezione.

Teorema 2.10. *Per ogni (G,T,c) il valore massimo di un T-pairing è uguale al costo minimo di un T-taglio.*

Teorema 2.11. *Per ogni nodo u in T esiste un nodo $v \in T \setminus \{u\}$ tale che $\{u,v\}$ è utile.*

Dimostrazione. Si applichi l'algoritmo *MinT-cut* a (G,T,c). Ad ogni ricorsione, si continui a scegliere $s = u$ finché il minimo $\{s,t\}$-taglio $\delta(S)$ è T-dispari. Dal Lemma 2.8, $\{u,v\}$ è utile rispetto a (G,T,c). □

2.4 Algoritmo

Abbiamo sviluppato, quindi, un algoritmo in Python usando sia il linguaggio di Gurobi che quello di Python. È possibile avviare il codice usando Spyder[3](la totalità del codice si trova in Appendice A). Qui di seguito analizzeremo i principali aspetti del codice. L'algoritmo può essere suddiviso in due:

1. Nella prima parte, abbiamo definito quattro funzioni fondamentali: `Contraction`, `minCutValue`, `findsubsets` e `minCutEdges`;

2. Nella seconda parte, abbiamo sviluppato il modello (ovvero funzione obiettivo e vincoli). Il ciclo `while` alla fine permette di ottimizzare il modello.

Per facilitare la comprensione di ciò che è stato fatto, partiamo analizzando la seconda parte del codice. Per prima cosa, abbiamo creato le variabili: ponendo come limite inferiore zero, le definiamo di tipo continuo. Inizialmente, le avevamo considerate variabili binarie, ma in tal modo l'algoritmo calcolava già soluzioni intere. Pertanto, per risolvere problemi reali, abbiamo deciso di usare variabili continue. In questo caso, all'inizio si ottiene una soluzione non intera. Pertanto otteniamo la funzione obiettivo:

$$\min \sum_{ij} \texttt{edges} \cdot \texttt{var}$$

che minimizza la somma dei prodotti tra le variabili continue `var` e `edges` che rappresenta il costo dell'arco considerato. L'unico vincolo che consideriamo è il seguente: la somma di tutti gli archi entranti in un dato vertice deve essere pari a 1.
Dopo aver trovato tutti gli archi incidenti in un vertice i e dopo aver imposto il vincolo definito sopra, definiamo il modello di ottimizzazione. Iterativamente, l'algoritmo ricerca la soluzione intera grazie ad un ciclo while. Questo significa che, nel ciclo, l'algoritmo ottimizza il modello, allora trova il valore del minimo taglio dispari e controlla se il suo valore è minore o maggiore di 1.
Se il minimo taglio dispari è minore di 1, l'algoritmo identifica gli archi del minimo taglio dispari e aggiunge un vincolo, ovvero che la somma delle variabili associate agli archi del minimo taglio dispari sia maggiore o uguale ad 1. E così via, fino a raggiungere la soluzione intera.

[3]Presente in Anaconda.

Nella prima parte del codice, Per prima cosa importiamo i moduli necessari a sviluppare il modello di ottimizzazione. Questa parte si basa sulla ricerca del T-taglio.
Abbiamo quindi definito una funzione per la contrazione, che crea un minore di G. La funzione `minCutValue(graph, T)` calcola il valore del minimo taglio dispari di un dato grafo; `T` è un dato usato nella ricorsione che deve essere inizializzato a '`vertices`'. La terza funzione che definiamo è `findsubset`, che trova tutti i sottoinsiemi di una fissata cardinalità di un dato insieme. Dato un grafo e il valore di uno dei suoi minimi tagli dispari, abbiamo creato una funzione che trova gli archi del minimo taglio dispari `minCutEdges`.

In un altro file Python abbiamo definito alcuni esempi di grafi su cui testare l'algoritmo, come il grafo di Petersen, TSP problem, un grafo che non ha perfect matching and il grafo $K(5)$ che sono riportati nell'Appendice B.

Capitolo 3

Conclusione

Nonostante i vantaggi dell'algoritmo presentato, il codice può essere migliorato. Per esempio, si possono fare le seguenti modifiche:

- I dizionari (.keys all'interno del codice) potrebbero essere usati per i vertici e non per gli archi, ma in questo caso si dovrebbe risolvere il problema dei pesi da assegnare agli archi;
- Gli esempi sono stati inseriti in un file python a parte e si potrebbe riuscire a strutturarlo come file di testo e richiamarlo nel file python matching;

L'idea iniziale era quella di utilizzare le funzioni Call-Back, cioè di utilizzarla all'interno del ciclo while come nel caso del TSP (Problema del commesso viaggiatore). In questo caso, tuttavia, le funzioni Call-Back non si potevano usare in quanto Gurobi non permette all'utente di utilizzare il valore delle variabili se non nella fase MIP, cioè in una fase del processo in cui si hanno solo valori interi delle soluzioni, ma il problema che si vuole aggirare con questo algoritmo è proprio quello di usare solo valori interi delle soluzioni.

Si è giunti quindi ad una contraddizione e si è dovuta scartare l'idea di utilizzare Call-Back all'interno del codice.

In conclusione, l'algoritmo può essere migliorato ed è stato aggirato il problema di utilizzare un certo tipo di funzioni all'interno del codice a causa di restrizioni del programma.

Appendice A

Perfect-Matching Codice

```
@author: Alessandro Zeggiotti
# ------------------------------------------------------------------
# USEFUL MODULES

import math
import random
from gurobipy import *

import networkx as nx
import itertools

# Matching_examples.py contains some useful examples to test this script
# QUESTION: Is "import" the best way to run an external script?
from Matching_examples import *

# ------------------------------------------------------------------
# FUNCTIONS DEFINITION

# Given a graph, this function idefintifies a set of nodes into a single vertex

def Contraction(graph, nodes):
    V = set()
    for i,j in graph:
        V = V.union({i})
        V = V.union({j})
    new_vertex = max(V) + 1
    # Create a minor of the graph
    minor = {}
```

```
    for i,j in graph.keys():
        if i in nodes and j in nodes:
            pass
        elif i in nodes and (j,new_vertex) not in minor.keys():
            minor[j,new_vertex] = graph[i,j]
        elif i in nodes:
            minor[j,new_vertex] += graph[i,j]
        elif j in nodes and (i,new_vertex) not in minor.keys():
            minor[i,new_vertex] = graph[i,j]
        elif j in nodes:
            minor[i,new_vertex] += graph[i,j]
        else:
            minor[i,j] = graph[i,j]
    return minor

# Given a graph, this function finds the value of a minimum odd cut
# T is a datum used during the recursion:
#it must be initially set equal to "vertices"

def minCutValue(graph, T):
    # Base step
    if T == set():
        return float('inf')
    # Create the graph using the module NETWORKX
    G = nx.DiGraph()
    for (i,j) in graph.keys():
        G.add_edge(i, j, capacity = graph[i,j])
        G.add_edge(j, i, capacity = graph[i,j])
    # Let s and t be two random nodes in T
    s = min(T)
    t = max(T)
    # NETWORKX contains a function that allows to find the minimum s-t cut
    # This function is based on Ford-Fulkerson algorithm
    cut_value, partition = nx.minimum_cut(G,s,t)
    S, S_bar = partition
    if len(S.intersection(T)) % 2 == 1: # if S is T-odd: ...
        return min(cut_value,
                   minCutValue(Contraction(graph, {s,t}), T - {s,t}))
    else:
        return min(minCutValue(Contraction(graph, S), T - S),
                   minCutValue(Contraction(graph, S_bar), T - S_bar))
```

```
# Given a set, this function finds all its subsets of a fixed cardinality

def findsubsets(S,m):
    # From ITERTOOLS module
    return set(itertools.combinations(S, m))

# Given a graph and the value of a minimum odd cut,
# this function finds a minimum odd cut

def minCutEdges(graph, cut_value):
    for cardinality in range(1,n,2):
        subsets = findsubsets(vertices, cardinality)
        # "vertices" is global variable
        for S in subsets:
            # Create the cut
            dS = {}
            for i,j in graph.keys():
                if i in S and j not in S:
                    dS[i,j] = graph[i,j]
                if i not in S and j in S:
                    dS[i,j] = graph[i,j]
            dS_value = sum(dS.values())
            if dS_value == cut_value:
                return dS
    print('No odd cut of value %g has been found' % cut_value)
    return None

# -------------------------------------------------------------------
# MODEL DEFINITION

m = Model()
# Suppress the standard output, it would be invoked too many times
m.setParam('OutputFlag', 0)

# Create the variables
vars = {}
for (i,j) in edges.keys():
   vars[i,j] = m.addVar(lb=0, vtype=GRB.CONTINUOUS, name='e[%d,%d]'%(i,j))

# To create the model data structure only once, after variables creation
```

```
m.update()

# Add the objective function
m.setObjective(sum(vars[i,j]*edges[i,j] for i,j in edges.keys()), GRB.MINIMIZE)

# Add degree-1 constraint
for i in vertices:
    # Look for the edges incident to i
    neighbors = []
    for j in vertices:
        if (i,j) in edges.keys():
            neighbors.append((i,j))
        elif (j,i) in edges.keys():
            neighbors.append((j,i))
    m.addConstr(sum(vars[i,j] for i,j in neighbors) == 1)

# ------------------------------------------------------------------
# MODEL OPTIMIZATION

linked = set()
for edge in edges.keys():
    linked = linked.union(set(edge))

# Check the presence of isolated vertices
if vertices != linked:
    print(str(vertices - linked) + ' is a set of isolated vertices!')

# Check the parity of the graph
elif n % 2 == 1:
    print('The graph is odd!')

else:
    # Now we are sure that the graph admits a solution
    while True:
        # Solve the problem
        m.optimize()
        print('\n-----------------------------------------------------\n')
        solution = m.getAttr('x', vars)
        selected = [(i,j) for i,j in solution.keys() if solution[i,j] != 0]
        print('New incumbent solution:\n')
        for i,j in selected:
```

```
    print('    ' + str((i,j)) + ' : ' + str(solution[i,j]))
print('\nObjective function: %g' % m.objVal)
print('Checking the presence of rationals...')
# Find the value of a minimum odd cut
cut_value = minCutValue(solution, vertices)
if cut_value < 1:
    # Find the edges of a minimum odd cut
    cut_edges = minCutEdges(solution, cut_value)
    # Add a constraint
    m.addConstr(sum(vars[i,j] for (i,j) in cut_edges.keys()) >= 1)
    print('Minimum odd cut: %g' % cut_value)
    print('Related edges:\n')
    for i,j in cut_edges.keys():
        print('    ' + str((i,j)) + ' : ' + str(cut_edges[i,j]))
    print('\nNew constraint added...')
else:
    print('This is a feasible solution!')
    break
```

Appendice B

Perfect-Matching Code Esempi

```
@author: Alessandro Zeggiotti
"""

# Create vertices and edges for our example
# The user can choose one among the four following graphs
print('\n1 --> TSP problem')
print('2 --> Petersen graph')
print('3 --> Unfeasible graph')
print('4 --> K(5)')
choice = int(input('Choose your favourite example: '))

if choice == 1:
    vertices = {0,1,2,3,4,5}
    edges = {
        (0,1): 5,
        (0,2): 5,
        (0,3): 4,
        (1,4): 2,
        (1,5): 3,
        (2,3): 3,
        (2,5): 8,
        (3,4): 4,
        (4,5): 4}
elif choice == 2:
    vertices = {0,1,2,3,4,5,6,7,8,9}
    edges = {
        (0,1): 1,
        (0,4): 1,
```

```
        (0,5): 1,
        (1,2): 1,
        (1,6): 1,
        (2,3): 1,
        (2,7): 1,
        (3,4): 1,
        (3,8): 1,
        (4,9): 1,
        (5,7): 1,
        (5,8): 1,
        (6,8): 1,
        (6,9): 1,
        (7,9): 1}
elif choice == 3:
    vertices = {0,1,2,3}
    edges = {
        (0,2): 1,
        (0,3): 1,
        (2,3): 1}
else:
    vertices = {0,1,2,3,4}
    edges = {
        (0,1): 1,
        (0,2): 1,
        (0,3): 1,
        (0,4): 1,
        (1,2): 1,
        (1,3): 1,
        (1,4): 1,
        (2,3): 1,
        (2,4): 1,
        (3,4): 1}

n = len(vertices)
```

Bibliografia

[1] Reinhard Diestel, "Graph Theory", Electionic Edition 2005

[2] John M. Harris, Jeffry L. Hirst, Michael J.Mossinghoff, "Combinatorics and Graph Theory", Springer Edition 2008

[3] Romeo Rizzi, "A Simple Minimum T-Cut Algorithm", October 23, 2002