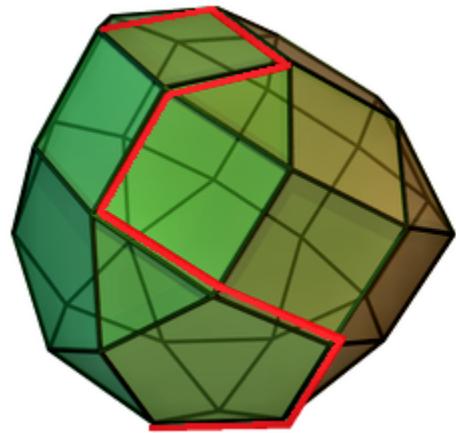


# Appunti di Ricerca Operativa

Andrea Mansi – UniUD

I° Semestre a.a. 2020/2021



# Contents

<b>1</b>	<b>Informazioni sugli appunti</b>	<b>4</b>
<b>2</b>	<b>Introduzione</b>	<b>5</b>
2.1	Problemi e Istanze . . . . .	5
2.2	Mathematical Programming . . . . .	7
<b>3</b>	<b>Modellazione tramite programmazione lineare intera – ILP</b>	<b>9</b>
3.1	Scelta delle variabili . . . . .	9
3.2	Scelta della funzione obiettivo . . . . .	10
3.3	Scelta dei vincoli . . . . .	11
3.4	Metodo del "Big-M" . . . . .	16
<b>4</b>	<b>Problemi classici di ottimizzazione combinatoria</b>	<b>18</b>
4.1	Binary Knapsack problem . . . . .	18
4.2	Set Covering . . . . .	18
4.3	Sottoinsieme indipendente . . . . .	18
4.4	SAT - Satisfiability . . . . .	19
4.5	Graph vertex coloring . . . . .	19
<b>5</b>	<b>Programmazione lineare</b>	<b>21</b>
5.1	Algoritmo del simplex . . . . .	23
5.2	Esempio esecuzione algoritmo del simplex . . . . .	28
5.3	Interpretazione geometrica del simplex . . . . .	30
5.4	Programmazione Lineare: Dualità . . . . .	35
<b>6</b>	<b>Programmazione intera</b>	<b>45</b>
6.1	Tipologie di programmi lineari interi . . . . .	45
6.2	Geometria della programmazione lineare intera . . . . .	47
6.3	Come risolvere un problema di programmazione lineare intera . . . . .	48
6.4	Problemi di programmazione lineare naturalmente interi . . . . .	48
6.5	Branch and Bound . . . . .	51
6.6	Branch & Bound - pseudocodice . . . . .	53
6.7	Metodo dei piani di taglio . . . . .	60
<b>7</b>	<b>Problemi di Ottimizzazione combinatoria e problemi sui grafi</b>	<b>68</b>
7.1	Problemi di ottimizzazione combinatoria . . . . .	68
7.2	Rappresentazione dei grafi e strutture dati . . . . .	68
7.3	Visite dei grafi . . . . .	70
7.4	Grafi aciclici e ordine topologico . . . . .	71
<b>8</b>	<b>Minimo Albero di Copertura – Minimum Spanning Tree (MST)</b>	<b>74</b>
8.1	Casi speciali di Spanning Trees . . . . .	81

<b>9 Cammini ottimali</b>	<b>82</b>
9.1 Cammino minimo – Shortest path . . . . .	83
9.2 Modelli di programmazione lineare intera . . . . .	84
9.3 Algoritmo di Dijkstra – NNC con pesi non negativi (1) . . . . .	88
9.4 NNC: caso grafi aciclici (2) . . . . .	92
9.5 Floyd-Warshall – NNC con archi negativi ma nessun ciclo negativo (3) . . . . .	97
<b>10 Reti di flusso</b>	<b>99</b>
10.1 Problema del massimo e del minimo flusso . . . . .	102
10.2 Restrizioni e generalizzazioni . . . . .	105
10.3 Insiemi di taglio e problema del minimo taglio . . . . .	106
10.4 Problema del taglio minimo . . . . .	109
10.5 Formulazioni di programmazione lineare . . . . .	109
10.6 Metodo di Ford e Fulkerson per il problema Max-Flow . . . . .	111
<b>11 Accoppiamenti (matchings)</b>	<b>117</b>
<b>12 Esercizi</b>	<b>123</b>
<b>13 Appendice: recap problemi e algoritmi</b>	<b>126</b>
<b>14 Appendice: ripasso nozioni</b>	<b>127</b>
14.1 Lineare indipendenza . . . . .	127
14.2 Matrice in forma di Echelon . . . . .	127

# 1 Informazioni sugli appunti

Lo scopo di questi appunti è fornire un riassunto dei principali argomenti trattati durante il corso di Ricerca Operativa (anno accademico 2020/2021).

Gli appunti non sono ufficiali, inoltre, potrebbero non ricoprire interamente il programma svolto. Quanto prodotto è inoltre frutto di una traduzione dall'inglese all'italiano, potrebbero quindi essere presenti errori di traduzione, typo etc.

Gli appunti sono tratti dalle lezioni frontali del Prof. Giuseppe Lancia e dal relativo libro (materiale ufficiale del corso di cui si consiglia la lettura per una copertura esaustiva della materia) "A First Course in Operations Research - 2020" (ISBN: 9798683871949).

---

**Ultima modifica: 20 gennaio 2021 ver. 1**

TODO:

Add to appendice: determinante, NP-Hard, ds tipo unionfind

Traduci alcune parti lasciate in inglese

Aggiungi alcune dimostrazioni

---

## 2 Introduzione

La Ricerca Operativa (nota anche come teoria delle decisioni o, in inglese operational research) è una branca della matematica applicata in cui problemi decisionali e di ottimizzazione complessi vengono analizzati e risolti mediante modelli matematici avanzati.

La ricerca operativa si occupa dunque di formalizzare un problema in un modello matematico e calcolare una soluzione ottima, quando possibile, o approssimata.

Esempio: un possibile esempio è quello del "Traveling Salesman Problem (TSP)": un venditore ambulante deve visitare  $n$  città, conoscendo il costo  $c(i, j)$  (ovvero la distanza) tra ogni coppia di città  $i$  e  $j$ . Vogliamo ottenere la sequenza di visite che copre tutte le città di minor costo. Le sequenze (permutazioni) possibili sono  $(n-1)!$ , di conseguenza, al crescere di  $n$ , un approccio naïve risulterebbe computazionalmente troppo costoso. La ricerca operativa mira a trovare algoritmi per risolvere problemi di questa natura.

Il focus della ricerca operativa è principalmente la risoluzione di problemi di ottimizzazione: è presente un obiettivo che vogliamo minimizzare (un costo, ad esempio) o massimizzare (un profitto).

I campi di applicazione della ricerca operativa sono numerosi, tra i principali troviamo:

- Problemi di Scheduling e di produzione;
- Time-Tabling (orari);
- Routing & Shipping;
- Progettazione di Reti;
- Assegnamenti;
- Applicazioni militari;
- Finanza;
- Medicina etc.

### 2.1 Problemi e Istanze

Esistono due principali categorie di problemi a cui possono essere applicate le tecniche della ricerca operativa:

- Problemi di **ottimizzazione**: ciascuna soluzione è associata a un costo o a un profitto. Siamo interessati in una soluzione che minimizzi il costo o massimizzi il profitto (in base alla natura del problema);
- Problemi di **decisione**: non si cerca una soluzione ottimale, ma ci interessa determinare se esiste almeno una soluzione valida al problema.

Un **problema** (di ottimizzazione o di decisione) è definito in modo astratto descrivendo una serie di dati che rappresentano il suo input e le proprietà che deve soddisfare la potenziale soluzione per essere corretta.

Entrambi i dati e le proprietà vanno descritti attraverso parametri generici. Ad esempio, nel problema TSP, l'input consiste in  $n(n - 1)$  numeri che denotiamo con  $c(i, j)$  che rappresentano la distanza (i costi) tra tutte le possibili coppie delle  $n$  città.

La soluzione è invece un grafo Hamiltoniano rappresentato da una permutazione degli  $n$  nodi, tale che la somma dei costi tra i nodi adiacenti è la minore possibile tra tutte le possibili permutazioni, ovvero tale che il valore della sommatoria sia il minore possibile.

$$\sum_{i=1}^{n-1} c(v_i, v_{i+1}) + c(v_n, v_1)$$

Una **istanza di un problema** è qualsiasi specifico esempio del problema ottenuto specificando i precisi valori di tutti i parametri generici del modello rappresentativo.

**Circuito Hamiltoniano:** dato un grafo, un cammino è detto hamiltoniano se esso tocca tutti i vertici del grafo una e una sola volta. Problema NP-completo. Se questo cammino è un ciclo, allora è un ciclo hamiltoniano.

**Problema di ottimizzazione:** un problema di ottimizzazione  $\Pi$  è definito da:

- O1. un insieme  $I(\Pi)$  di istanze;
- O2. una famiglia  $S_\Pi = \{S_I, I \in I(\Pi)\}$  di soluzioni fattibili, dove  $S_I$  appartiene a  $I(\Pi)$  è l'insieme di soluzioni fattibili per l'istanza  $I$ ;
- O3. una funzione obiettivo  $f_\Pi : \{(x, I), I \in I(\Pi), x \in S_I\} \mapsto \mathbb{R}$  che associa a ciascuna soluzione un valore costo.

Il generico problema di ottimizzazione  $\Pi$  consiste nell'ottenere, per ciascuna istanza  $I \in I(\Pi)$ , una soluzione  $x^*$  (soluzione ottimale) t.c.  $f_\Pi(x^*, I) \leq f_\Pi(x, I)$  per ciascun  $x \in S_I$  (o  $\geq$  nel caso di massimizzazione).

**Problema di decisione:** un problema di decisione  $\Pi$  è definito da:

- D1. un insieme  $I(\Pi)$  di istanze;
- D2. un sottoinsieme  $Y_{es}(\Pi) \subseteq I(\Pi)$  di istanze Yes.

Il generico problema di decisione  $\Pi$  consiste nel, fornita una istanza  $I \in I(\Pi)$  verificare se  $I \in Y_{es}(\Pi)$ .

**Formato:** in generale descriviamo un problema di ottimizzazione seguendo il seguente formato:

**PROBLEMA:** nome del problema

**INPUT:** descrizione delle istanze generiche del problema

**OBIETTIVO:** descrizione della soluzione e il suo costo

Similmente descriviamo un generico problema di decisione utilizzando la seguente forma:

**PROBLEMA:** nome del problema

**INPUT:** descrizione delle istanze generiche del problema

**OBIETTIVO:** le proprietà che deve soddisfare l'istanza generica per essere di tipo YES

## 2.2 Mathematical Programming

Dato un problema di ottimizzazione  $\Pi$ , se le soluzioni fattibili appartengono a  $\mathbb{R}^n$  (ovvero sono vettori) diciamo che il problema  $\Pi$  è un problema di mathematical programming.

Sia  $X \subseteq \mathbb{R}^n$  l'insieme delle soluzioni di un'istanza di  $\Pi$  e  $f$  la funzione obiettivo. Allora un problema di mathematical programming  $\Pi$  può essere definito come:

*massimizzare  $f(x)$  per  $x \in X$  per ogni istanza di  $\Pi$*

Se esiste  $x^*$  tale che  $f(x^*) = \max\{f(x) : x \in X\}$  allora  $x^*$  è chiamata soluzione ottimale e  $f(x^*)$  è chiamato valore ottimale.

La funzione obiettivo, può, in linea generale, essere qualsiasi funzione a valori reali (lineare, quadratica, trigonometrica, etc.). I casi principali sono quello lineare e quello quadratico.

**Funzione lineare:** somma di variabili, ciascuna pesata da un coefficiente, come ad esempio:

$$3x_1 + 2x_2 - 4x_3$$

e può essere espressa come un prodotto  $c^T x$  per un  $c \in \mathbb{R}^n$ .

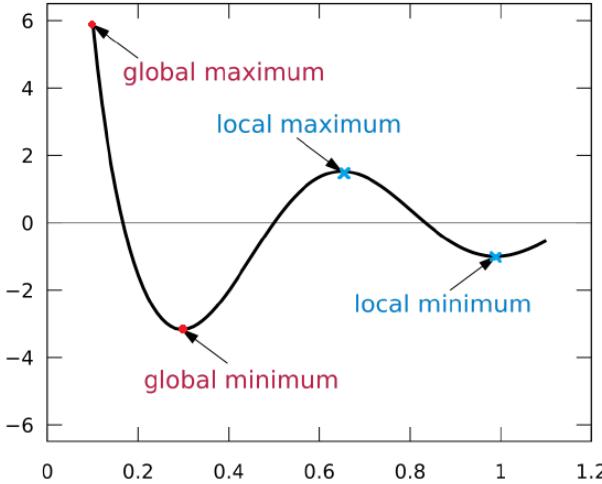
**Funzione quadratica:** somma di variabili pesate e di prodotti pesati di due variabili, come ad esempio:

$$2x_1x_3 - x_2^2 + 4x_2x_3 - x_1 + 3x_3$$

e può essere espressa come  $c^T x + x^T C x$  per un  $c \in \mathbb{R}^n$  e una matrice non nulla  $C \in \mathbb{R}^{n \times n}$ .

**Località ottimale:** Sia  $\hat{x} \in X$ . Se esiste  $\epsilon > 0$  t.c.  $f(\hat{x}) \geq f(x)$  per ciascun  $x \in X$  con  $\|\hat{x} - x\| \geq \epsilon$  allora  $\hat{x}$  è chiamato **ottimo locale** (massimo locale).

Chiaramente un ottimo globale è anche un ottimo locale, ma in generale il contrario non è vero.



Di particolare importanza sono i casi in cui l'insieme delle soluzioni fattibili è convesso e la funzione obiettivo è convessa (per i problemi di minimizzazione) o concava (per i problemi di massimizzazione).

La massimizzazione di funzioni concave (o la minimizzazione di funzioni convesse) su un insieme convesso ha una proprietà molto importante. Per ottenere un ottimale globale è sufficiente trovare un ottimale locale.

**TEOREMA:** sia  $\max\{f(x) \text{ t.c. } x \in X\}$  un problema di mathematical programming tale che  $X$  è un insieme convesso e  $f$  è una funzione concava. Sia  $\hat{x}$  un ottimale locale. Allora  $\hat{x}$  è anche un ottimale globale.

I problemi in cui  $X$  è un insieme convesso e in cui cerchiamo di massimizzare una funzione concava (o minimizzare una funzione convessa) sono detti problemi di **convex optimization** (problem convessi). L'esempio più importante è la programmazione lineare (linear programming), ovvero, problemi in cui la funzione obiettivo è lineare.

In generale i problemi convessi sono più semplici da risolvere.

### 3 Modellazione tramite programmazione lineare intera – ILP

Uno dei principali contributi della ricerca operativa consiste nello sviluppo di una ricca teoria chiamata programmazione lineare intera (Integer Linear Programming - ILP) per modellare e risolvere diversi problemi di ottimizzazione.

Per adesso considereremo il solver dei programmi ILP come una black box. Ci occuperemo quindi, solo di come fornire in maniera corretta un modello matematico e la rappresentazione del relativo input.

Dobbiamo sostanzialmente utilizzare un linguaggio di modellazione che ci permetta di scrivere in maniera corretta un modello matematico rappresentativo del problema (alcuni modelli sono migliori di altri). Nello specifico dobbiamo definire:

- le variabili;
- la funzione obiettivo;
- i vincoli.

Prima di parlare di come scegliere queste tre componenti, introduciamo il concetto di programmazione lineare.

**Programmazione lineare:** è un caso particolare dell’ottimizzazione convessa in cui la funzione obiettivo è lineare e tutti i vincoli definiti sono diseguaglianze. In questo caso, essendo un problema convesso, gli ottimi locali sono anche ottimi globali. Un programma lineare in  $n$  variabili e  $m$  vincoli è quindi della forma:

$$\max / \min \sum_{i=1}^n c_i x_i$$

soggetto ai vincoli:

$$\sum_{j=1}^m a_{ij} x_j \geq b_i \text{ per } i = 1, \dots, m$$

Nel caso in cui alcune variabili debbano essere intere (Mixed Integer Linear Programming - MILP) o nel caso in cui tutte le variabili debbano essere intere (Integer Linear Programming - ILP), i vincoli descritti non possono essere usati per forzare queste proprietà.

#### 3.1 Scelta delle variabili

Nei modelli matematici le variabili sono componenti numeriche reali. Per alcuni problemi però le quantità di cui vogliamo trovare il valore ottimale sono spesso non negative. Inoltre, in alcuni casi le quantità numeriche richiedono di essere intere. Questo vincolo di "interezza", come vedremo, rende il modello molto più difficile da risolvere.

In altri casi, le variabili rappresentano decisioni booleane, oppure rappresentano la scelta tra un insieme di etichette (labels), che devono essere opportunamente codificate.

Per le variabili booleane la scelta tipica è l'utilizzo di variabili binarie (che possono assumere solo il valore 1 o 0) associando l'1 al VERO e lo 0 al valore booleano FALSO.

### 3.2 Scelta della funzione obiettivo

La funzione obiettivo di un programma lineare è una funzione lineare. Ciascuna variabile  $x_i$  ha un costo che dipende, linearmente, dal valore della variabile. Sia  $c_i$  il costo unitario di una variabile. Il costo totale di una soluzione sarà quindi:

$$\sum_{i=1}^n c_i x_i$$

Si noti come il costo lineare non sempre rappresenta la realtà: si pensi ad esempio ad un rivenditore che effettua degli sconti all'aumentare della quantità di prodotti comprati: ad esempio se compri più di  $n$  oggetti ricevi uno sconto del 10%.

Anche se il costo è proporzionale al valore delle variabili, questo potrebbe essere vero solo sopra una certa soglia (threshold). Si pensi ad esempio ad un caso di produzione, in cui esistono dei costi fissi (di avvio produzione) indifferentemente da quanti oggetti andremo a produrre. Questo fa sì che siano possibili tre casi:

- $f(x) = 0$  se  $x = 0$
- $f(x) = V$  se  $0 < x \leq \tau$
- $f(x) = V + c(x - \tau)$  se  $x > \tau$

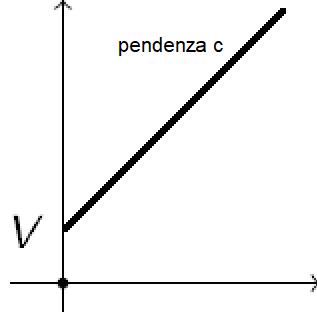
dove  $V$  sono i costi fissi (di avvio produzione, ad esempio) e  $\tau$  la soglia minima (n° di oggetti da produrre) affinchè il numero di oggetti influisca sul costo. Da  $\tau$  in poi il costo diventa lineare.

**Modellazione dei costi di avvio produzione (start-up costs):** i costi di avvio produzione possono essere modellati nella programmazione lineare, sostituendo la decisione del valore di  $x$  con quella di  $y$ , rappresentante il numero di oggetti da produrre in eccesso a  $\tau$ , ovvero  $x = \tau + y$ . A questo punto il modello si semplifica in due casi:

- $f(y) = 0$  se  $y = 0$
- $f(y) = V + cy$  se  $y > 0$

Possiamo anche introdurre una variabile  $z$  che vale 0 se  $y$  non è maggiore di 0, e 1 se lo è. A questo punto possiamo modellare il tutto con un solo caso:

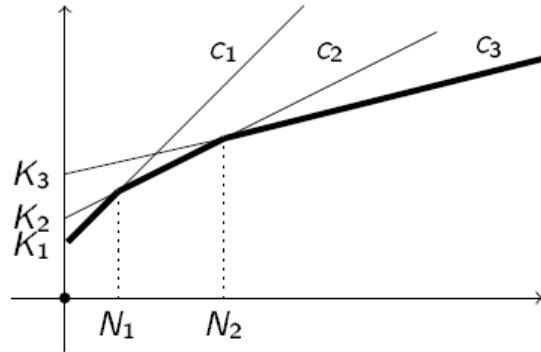
$$Vz + cy$$



**Esempio: Economia di scala:** il precedente modello è collegato al principio di "economia di scala", che rappresenta il seguente concetto: al crescere delle unità prodotte di un certo bene, più economica diventa la singola unità (perchè si ammortizza il costo di produzione).

Il seguente modello può essere rappresentato da una funzione lineare a tratti. In questo modello possiamo avere i valori  $N_0, N_1, N_2, \dots, N_n$  tale che produrre un numero  $x \in (N_{i-1}, N_i]$  ha un costo unitario  $c_i$  t.c.  $c_i \geq c_j$  per  $i < j$ .

Sia allora  $K_1$  il costo fisso di inizio produzione, allora per  $i = 2, \dots, n$  definiamo il valore  $K_i = K_{i-1} + (c_{i-1} - c_i)N_i$ . Otteniamo una funzione lineare a tratti di questo tipo:



### 3.3 Scelta dei vincoli

I vincoli di un modello di programmazione lineare (intera e non) sono sempre disequazioni lineari, come ad esempio:

$$a_1x_1 + \dots + a_nx_n \geq b \quad \text{oppure} \quad a_1x_1 + \dots + a_nx_n \leq b$$

o equazioni lineari (ottenibili con due disequazioni), come:

$$a_1x_1 + \dots + a_nx_n = b$$

con  $a_1, \dots, a_n, b \in \mathbb{R}$ .

NB: qualsiasi vincolo che non sia una diseguaglianza non è consentito. Bisogna quindi (quando possibile) tradurre vincoli booleani o di altro tipo in disequazioni.

In alcuni problemi, alcune diseguaglianze sono ovvie dalla definizione del problema: ad esempio limiti inferiori e superiori di variabili, come:

$$l_i \leq x_i \leq u_i$$

con  $l_i$  limite inferiore e  $u_i$  limite superiore (lower, upper).

Le diseguaglianze lineari possono essere usate per modellare relazioni tra variabili come condizioni booleane o implicazioni logiche (soprattutto nel caso di variabili binarie, come nel caso booleano).

**Diseguaglianze (vincoli) loose e non-trivial:** quando forziamo delle relazioni tra variabili tramite disequazioni, i vincoli possono essere di due tipologie:

- **loose (laschi):** non vincolano nulla tra le variabili;
- **non-trivial (stretti):** vincolano/limitano la libertà dei valori assumibili da alcune variabili.

Una relazione può essere **loose** o **non-trivial**, in base ai valori assunti: supponiamo di voler forzare una relazione tra  $k$  variabili  $x_1, \dots, x_k$  e le rimanenti variabili  $n - k$ : vogliamo che l'assegnamento degli specifici valori  $v_1, \dots, v_k$  alle variabili  $x_1, \dots, x_k$  abbia un diretto impatto alle rimanenti variabili  $x_{k+1}, \dots, x_n$ . Nulla però viene implicato se le prime  $k$  variabili non valgono i valori  $v$  specificati.

Allora specifichiamo una diseguaglianza:

$$a_{k-1}x_{k+1} + \dots + a_nx_n \leq b - \sum_{i=1}^k a_i k_i$$

Sia  $I$  l'insieme delle restanti disequazioni del modello. Dato qualsiasi assegnamento  $\lambda \in \mathbb{R}$ , diciamo che la diseguaglianza precedente è **loose** per l'assegnamento  $(x_1, \dots, x_k) \leftarrow \lambda$  se ogni volta che  $(\lambda, x_{k+1}, \dots, x_n)$  è una soluzione per  $I$  lo è anche per la diseguaglianza precedente. In caso opposto, possono esistere soluzioni  $(x_1, \dots, x_k) \leftarrow \lambda$  per  $I$  ma non per la disequazione precedente, in questo caso la diseguaglianza è detta **non-trivial**.

Il nostro scopo è quindi scrivere la diseguaglianza voluta in modo tale che sia non trivial per l'assegnamento  $(v_1, \dots, v_k)$  e loose per tutti gli altri.

**OR logico:** siano  $x$  e  $y$  variabili binarie. Possiamo imporre che  $x \vee y$  sia VERO tramite il vincolo:

$$x + y \geq 1$$

Per soddisfare il vincolo basta che  $x$  o  $y$  (o entrambi) valgano 1. Si può generalizzare (covering) l'OR di  $n$  variabili tramite il vincolo:

$$x_1 + x_2 + \dots + x_n \geq 1$$

Per definire una variabile  $z$  t.c. valga il valore dell'OR, possiamo usare  $n+1$  diseguaglianze del tipo:

$$z \geq x_i \quad \text{per} \quad i = 1, \dots, n \quad \text{e} \quad z \leq x_1 + \dots + x_n$$

L'ultima diseguagliaanza forza  $z$  ad essere 0 quando la somma delle  $x$  vale 0 (quando sono tutte false). Le altre  $n$  diseguaglianze forzano  $z$  ad essere 1 se una delle  $x_i$  vale 1.

**OR logico esclusivo (XOR):** l'OR esclusivo (vale TRUE solo quando uno dei due è VERO) si può rappresentare con quattro diseguaglianze:

- $z \leq x + y$  forza  $z$  a 0 se  $x$  e  $y$  sono 0;
- $z \leq 2 - x - y$  forza  $z$  a 0 se  $x$  e  $y$  sono 1;
- $z \geq x - y$  forza  $z$  a 1 quando esattamente  $x=1$  e  $y=0$ ;
- $z \geq y - x$  forza  $z$  a 1 quando esattamente  $x=0$  e  $y=1$ .

Si può però forzare lo XOR tra due variabili con una sola disequazione, introducendo una variabile in più:

$$z = x + y - 2w$$

con  $w = 0$  se  $x = y = 0$  o  $x = y = 1$  (ovvero  $x$  e  $y$  hanno la stessa parità).

Si noti che per imporre che lo XOR di  $n$  variabili sia VERO, il seguente vincolo deve essere soddisfatto:

$$x_1 + \dots + x_n = 1$$

Questo tipo di vincolo può essere anche usato per selezionare esattamente un elemento tra un insieme di  $n$ , infatti viene chiamato anche vincolo di partizione o selezione.

**Vincolo di assegnamento:** nel contesto in cui ci sono  $n^2$  variabili e ciascuna di esse è indicizzata da due indici  $i, j$ , un vincolo di partizione come:

$$\sum_{j=1}^n x_{ij} = 1$$

viene anche detto di assegnamento, in quanto può essere interpretato come l'assegnamento a  $i$  dell'elemento  $p(i) \in [n]$ , ovvero l'unico indice per cui  $x_{i,p(i)} = 1$ . Se ci sono  $n^2$  vincoli del tipo:

$$\sum_{j=1}^n x_{ij} = 1 \quad \text{per ogni} \quad i = 1, \dots, n$$

$$\sum_{i=1}^n x_{ij} = 1 \quad \text{per ogni } j = 1, \dots, n$$

possiamo interpretare ciascuna soluzione binaria come un sistema di diseguaglianze che definiscono una permutazione  $(p(1), \dots, p(n))$  di  $[n]$ .

Si pensi a una matrice quadrata, il primo vincolo dice che in ogni riga c'è un solo 1, nel secondo vincolo si dice che c'è solo un 1. La matrice identifica quindi una matrice di permutazione (utile ad esempio nel problema TSP).

**AND logico:** siano  $x$  e  $y$  due variabili binarie. Per imporre che  $x \wedge y$  sia VERO usiamo il vincolo  $x + y = 2$ . In questo caso il modello è errato in quanto il vincolo predispone che  $x$  e  $y$  siano vincolate al valore 1 (siano costanti).

Un modo migliore per modellare l'AND è il seguente: introduciamo una terza variabile  $z$  che vale l'AND di  $x$  e  $y$ . Usiamo quindi tre vincoli:

$$z \leq x \quad z \leq y \quad z \geq x + y - 1$$

Nel caso dell'AND di  $n$  variabili si può generalizzare nel seguente modo:

$$z \leq x_i \quad \text{per ogni } i = 1, \dots, n$$

$$z \geq \sum_{i=1}^n x_i - (n - 1)$$

**NOT logico:** per settare una variabile  $y$  come la negazione di  $x$ , basta introdurre un vincolo:

$$y = 1 - x$$

Si può comunque utilizzare  $1 - x$  al posto di  $y$ , senza dover introdurre  $y$ . Forzare  $\neg x$  suggerisce una cattiva modellazione.

**Implicazione logica:** vogliamo forzare la relazione  $x \implies y$  tra  $x$  e  $y$ . A livello di algebra booleana coincide con:

$$\neg x \vee y$$

Quindi si può utilizzare il seguente vincolo:

$$(1 - x) + y \geq 1$$

che può essere scritto come:

$$y \geq x$$

In questa ultima forma è chiaro che  $x$  implica  $y$  dato che  $x$  fornisce un limite inferiore e che quando  $x$  vale 1  $y$  può valere solo 1.

**Relazioni logiche complesse:** supponiamo di voler forzare qualche condizione complessa tra variabili boleane, ad esempio:

”Se  $x=\text{TRUE}$  allora  $y$  implica  $z$ , altrimenti o  $y=\text{FALSO}$  o  $z=\text{FALSO}$ ”

Il modo migliore è di considerare una tabella di tutti i casi per  $x,y,z$  e avere condizioni che impediscono i casi che non soddisfano la condizione.

case n.	x	y	z	valid
1	0	0	0	ok
2	0	0	1	ok
3	0	1	0	ok
4	0	1	1	NO
5	1	0	0	ok
6	1	0	1	ok
7	1	1	0	NO
8	1	1	1	ok

Scriveremo quindi due vincoli che vietino le configurazioni 4 e 7 dell'esempio.

Supponiamo di voler vietare una specifica configurazione di valori booleani. Assumiamo di avere  $n$  variabili  $x_i$  e di voler vietare la sequenza degli assegnamenti  $b_i \in \{0, 1\}$  seguente:

$$b_1, \dots, b_n$$

Se tutti gli  $x_i$  valessero il corrispettivo  $b_i$  avremmo:

- $x_i = 1$  quando  $b_i = 1$
- $1 - x_i = 1$  quando  $b_i = 0$

In totale avremmo quindi:

$$\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} (1 - x_i) = n$$

Ma questo è quello che NON vogliamo accadere, quindi ci basta dire che ALMENO uno degli addendi tra gli 1 sia 0, imponendo:

$$\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} (1 - x_i) \leq n - 1$$

semplificabile come:

$$\sum_{i=1}^n c_i x_i \leq n - K - 1$$

dove  $K$  è il numero di coefficienti  $i$  tale che  $b_i = 0$  e:

$$c_i = \begin{cases} 1 & \text{se } b_i = 1 \\ -1 & \text{se } b_i = 0 \end{cases}$$

Ritornando all'esempio, per vietare l'assegnamento di (0,1,1) del caso 4 per x,y,z, introduciamo il vincolo:

$$-x + y + z \leq 1$$

Per il caso 7 (1,1,0) introduciamo:

$$x + y - z \leq 1$$

**Limiti superiori e inferiori condizionali per una variabile:** se una variabile non negativa deve assumere valori compresi in un intervallo  $[L, U]$ , si può ottenere con estrema facilità inserendo i vincoli seguenti:

$$x \geq L \quad x \leq U$$

Possono essere possibili però casi in cui vogliamo che una variabile  $x$  sia = 0 o appartenente all'intervallo  $[L, U]$  (da qui il concetto di condizionale). Ovvero  $x \in \{0\}$  oppure  $x \in [L, U]$ . Posso aggiungere una variabile binaria  $y$  t.c.:

$$y = \begin{cases} 1 & \text{se } x \in [L, U] \\ 0 & \text{se } x \in \{0\} \end{cases}$$

### 3.4 Metodo del "Big-M"

Il metodo del big-M va utilizzato quando non ci sono alternative, in quanto risulta un modello debole e instabile numericamente.

Supponiamo che il nostro problema abbia due o più condizioni e che una soluzione fattibile ne deve soddisfare almeno una. Ad esempio:

$$X = \{x : (a_i^T \leq b_1) \vee \dots \vee (a_k^T \leq b_k)\}$$

Si noti come  $X$  è l'unione di  $n$  semi-spazi. Una diseguaglianza infatti definisce un iperipiano. Nella programmazione lineare, l'intersezione di  $m$  spazi convessi è convessa. L'unione di  $m$  spazi convessi, invece, tipicamente NON è convessa. Vogliamo quindi rappresentare dei punti che appartengono a uno spazio non convesso tramite intersezione di diseguaglianze. Questo si può fare con un trick matematico chiamato **Big-M**.

Sia  $M$  un numero "sufficientemente grande" (almeno t.c.  $-M \leq a^T x \leq M$ ).  $M$  si comporta come se fosse pari a infinito. Per ragioni pratiche bisogna prendere l' $M$  più piccolo possibile ma che soddisfi la precedente condizione (per problemi di stabilità numerica, arrotondamento). Si noti come un  $M$  adeguato rende sempre vera quella disequazione, anche quando vengono rispettati da  $x$  tutti gli altri vincoli del modello.

Se  $y$  è una variabile binaria, vincoli come i seguenti sono sempre loose quando  $y = 1$  e quando  $y = 0$  diventano "veri" vincoli (rispettivamente  $a^T x \leq b$  e  $a^T x \geq b$ ).

$$a^T x \leq b + My \quad \text{e} \quad a^T x \geq b - My$$

Nell'esempio dell'unione di  $k$  semi-spazi, possiamo introdurre  $k$  variabili binarie  $y_1, \dots, y_k$  e  $k + 1$  vincoli del tipo:

$$a_i^T x \leq b_1 - My_i \quad \text{per ogni } i = 1, \dots, k \quad \text{e} \quad \sum_{i=1}^k y_i \leq k - 1$$

Semplificato all'unione di due semi-spazi possiamo introdurre la variabile  $y$  e due vincoli come:

$$a_1^T x \geq b_1 - M(1 - y) \quad \text{e} \quad a_2^T x \geq b_1 - My$$

Si noti come se  $y = 1$  il secondo vincolo è loose e il primo è non-trivial, nel caso in cui  $y = 0$  abbiamo che il primo è non-trivial e il secondo loose. In entrambi i casi uno dei due deve essere soddisfatto.

**Esempio: exclusive "either-or":** esistono casi in cui due vincoli sono incompatibili, ovvero solo uno dei due può e deve essere soddisfatto. In questo caso l'intersezione dei due vincoli è vuota. Il metodo del big-M risulta utile in questo caso.

Esempio: problema di scheduling di  $n$  lavori e una macchina. Ciascun lavoro  $i$  deve usare la macchina per un tempo  $p_i$  senza prelazione. La variabile  $x_i$  denota il tempo di partenza del lavoro  $i$ . Quando il lavoro  $i$  è sotto processo la macchina non può essere usata per un nuovo lavoro  $j$  finché  $i$  non è completato. Abbiamo quindi vincoli del tipo:

$$[x_j, x_j + p_j) \cap [x_i, x_i + p_i) = \emptyset \quad \text{per ogni } i, j$$

Che corrisponde a "o  $j$  è processato dopo  $i$ , o  $i$  è processato dopo  $j$ ". Quindi abbiamo  $x_j \geq x_i + p_i$  o  $x_i \geq x_j + p_j$ .

Introduciamo allora  $n(n-1)/2$  variabili binarie  $y_{ij}$  con il seguente significato:

- 1 se  $i$  è processato prima di  $j$
- 0 se  $j$  è processato prima di  $i$

e per ciascuna coppia di lavori  $i$  e  $j$  introduciamo il vincolo:

$$x_i \leq x_j - p_i + M(1 - y_{ij})$$

e

$$x_j \leq x_i - p_j + My_{ij}$$

Il big M farà sì che uno dei due vincoli sarà loose e uno non-trivial.

**Forzare vincoli attraverso la funzione obiettivo:** analizzando la funzione obiettivo si può concludere che alcuni vincoli siano ridondanti (e quindi inutili) in quanto sono violati solo da soluzioni non ottimali (ad esempio: in un problema di minimizzazione di  $x$ , vincoli come  $x < y$  sono inutili,  $x > y$  sono utili). Questi vincoli quindi si possono rimuovere dal modello, la soluzione ottima rimane la stessa, quelle ammissibili potrebbero però cambiare (ma a noi interessa quella ottima).

## 4 Problemi classici di ottimizzazione combinatoria

### 4.1 Binary Knapsack problem

- INPUT: numeri positivi detti profitti  $p_1, \dots, p_n$  e pesi  $w_1, \dots, w_n$  e una capacità  $C$ .
- OBIETTIVO: trovare il sottoinsieme  $S$  di  $[n]$  tale che  $w(S) \leq C$  e  $p(S)$  più grande possibile.

MODELLO: introduciamo una variabile binaria  $x_i$  per ciascun oggetto, rappresentante se  $i$  è in  $S$  o no. La funzione obiettivo è:

$$\max \sum_{i=1}^n p_i x_i$$

con il vincolo:

$$\sum_{i=1}^n w_i x_i \leq C$$

### 4.2 Set Covering

- INPUT: una famiglia di sottoinsiemi  $S_1, \dots, S_n$  di  $[m]$  e pesi positivi  $c_1, \dots, c_n$ .
- OBIETTIVO: trovare un sottoinsieme  $F$  di  $[n]$  tale che  $\cup_{j \in F} S_j = [m]$  e  $\sum_{j \in F} c_j$  più piccolo possibile.

MODELLO: per  $X \subseteq [n]$  definiamo  $S_X := \{S_j | j \in X\}$ .  $S_X$  è una copertura se per ogni  $i \in [m]$  esiste qualche  $S_j \in S_X$  tale che  $i \in S_j$ . Abbiamo variabili binarie  $x_i$  che rappresentano se  $S_i$  è nella copertura. L'obiettivo è quindi coprire l'intero insieme dei valori utilizzando i sottoinsiemi di costo complessivo minore:

$$\min \sum_{j=1}^n c_j x_j$$

Per ciascun  $i \in [m]$  almeno uno degli insiemi che contiene  $i$  deve essere preso, quindi imponiamo i vincoli:

$$\sum_{j: S_j \ni i} x_j \geq 1 \quad \text{per ogni } i = 1, \dots, m$$

### 4.3 Sottoinsieme indipendente

- INPUT: un grafo  $G = ([n], E)$  e pesi positivi  $w_1, \dots, w_n$  per i vertici.
- OBIETTIVO: trovare un sottoinsieme  $S \subseteq [n]$  di nodi tale che a due a due non sono adiacenti (non ci sono archi) e il peso  $S$  è più alto possibile.

MODELLO: introduciamo variabili  $x_i$  per rappresentare se l'elemento  $i$  è in  $S$  o no.  
L'obiettivo è:

$$\max \sum_{i=1}^n w_i x_i$$

con i vincoli seguenti:

$$x_i + x_j \leq 1 \quad \text{per ogni } ij \in E$$

c'è un vincolo per ogni arco che specifica che non possono essere presenti entrambi i nodi nella soluzione. Ovvero: da ogni arco posso prendere un solo estremo o nessuno; non due.

## 4.4 SAT - Satisfiability

- INPUT: un insieme di  $m$  clausole  $C_i$  su  $n$  variabili boleane  $X_i$ . Ciascuna clausola è un OR di alcuni letterali, dove ciascun letterale è una variabile  $X_i$  o la sua negazione.
- OBIETTIVO: trovare un assegnamento  $\tau : \{X_1, \dots, X_m\} \mapsto \{T, F\}$  che rende vera (soddisfa) più clausole possibili.

MODEL: abbiamo variabili binarie  $x_i$  rappresentanti se impostiamo  $\tau(X_i)$  a VERO ( $x_i = 1$ ) o a FALSO ( $x_i = 0$ ). Abbiamo anche variabili binarie  $y_j$  impostata a 1 se e solo se  $C_j$  è soddisfatta da  $\tau$ . L'obiettivo è massimizzare il numero di clausole soddisfatte, ovvero:

$$\max \sum_{j=1}^m y_j$$

con i vincoli seguenti:

$$y_j \leq \sum_{i: X_i \in C_j} x_i + \sum_{i: \neg X_i \in C_j} (1 - x_i) \quad \text{per ogni } j = 1, \dots, m$$

che impostano a 0  $y_j$  quando l'i-esima clausola non è soddisfatta.

## 4.5 Graph vertex coloring

- INPUT: un grafo  $G = (V, E)$  non bipartito.
- OBIETTIVO: mappare una colorazione dei vertici  $c : V \mapsto \{1, \dots, k\}$  tale che  $c(i) \neq c(j)$  per ogni  $ij \in E$  e che  $k$  sia minore possibile (numero di colori).

Questo in generale è un problema NP-HARD.

Sia  $V = [n]$  e supponiamo che non ci siano nodi isolati (non creano problemi, basta colorarli tutti dello stesso colore). Vediamo due modelli ragionevoli:

**1° MODELLO (scarso):** ad ogni nodo associamo una variabile intera  $x_i \geq 1$  tale che i rappresenta il colore assegnato. Abbiamo anche una variabile  $w = \max x_1, \dots, x_n$ .

La funzione obiettivo è quindi  $\min w$ .

Abbiamo vincoli che forzano  $w$  ad essere il massimo di tutte le  $x_i$ :

$$x_i \leq w \text{ per ogni } i = 1, \dots, n$$

Abbiamo bisogno che per ogni  $ij \in E$  valga  $x_i \neq x_j$ . Ovvero:

$$(x_i \geq x_j + 1) \vee (x_j \geq x_i + 1)$$

Per modellare questo tipo di vincolo (non possiamo introdurre degli OR) possiamo introdurre variabili binarie  $y_{ij}$  per  $ij \in E$  e imporre:

$$x_i \geq x_j + 1 - My_{ij} \quad \forall ij \in E$$

$$x_j \geq x_i + 1 - M(1 - y_{ij}) \quad \forall ij \in E$$

Uno dei due sarà sempre loose; a seconda del valore di  $y_{ij}$  "scatta" il primo o il secondo vincolo. Il big-M in questo modello basta essere imposto al valore di  $n$  in quanto la "distanza" massima tra due etichette (colori) è tra 0 e  $n-1$ .

**2° MODELLO (buono):** abbiamo variabili binarie a doppio indice  $x_{vc}$  per ciascun  $v \in V$  e colore  $c \in \{1, \dots, n\}$ .  $x_{vc} = 1$  se il vertice  $v$  ha colore  $c$ . Inoltre abbiamo  $n$  variabili  $y_c$  che valgono 1 se e solo se il colore  $c$  è stato utilizzato almeno per un nodo.

La funzione obiettivo è la seguente:

$$\min \sum_{c=1}^n y_c$$

soggetta ai vincoli:

$$1) \quad \min \sum_{c=1}^n x_{cv} = 1 \quad \forall v \in V$$

$$2) \quad x_{vc} \leq y_c \quad \forall v \in V, \forall c = 1, \dots, n$$

$$3) \quad x_{ic} + x_{jc} \leq y_c \quad \forall ij \in E, \forall c = 1, \dots, n$$

il vincolo 1 garantisce che ciascun nodo riceva esattamente un colore. Il vincolo 2 dice che un colore non usato ( $y_c = 0$ ) allora nessun vertice può essere colorato di quel colore. Il vincolo 3 implica che per ciascun arco al massimo uno dei due può essere di uno specifico colore (no colori uguali adiacenti).

## 5 Programmazione lineare

Premessa: la programmazione lineare intera è più complessa della programmazione lineare, ma si basa su di essa. Iniziamo parlando della programmazione lineare.

Un problema di programmazione lineare è la massimizzazione o minimizzazione di una funzione lineare su un insieme di (soluzioni) vettori di numeri reali sotto un numero finito di vincoli lineari (diseguaglianze e/o equazioni).

Un'istanza di un problema di programmazione lineare è chiamato un programma lineare (LP - Linear Program).

La forma più generale di un LP con  $m$  vincoli e  $n$  variabili può essere espressa come:

$$\max c^T x$$

(NB:  $T \mapsto$  Trasposta,  $c$  = vettore riga,  $c^T$  = vettore colonna).  
soggetta a vincoli:

$$a_i^T x \leq b_i \quad i \in I$$

$$a_i^T x = b_i \quad i \in E$$

$$x_j \geq 0 \quad j \in C$$

$$x_j \geq 0 \quad j \in U$$

dove  $c, a_1, \dots, a_m \in \mathbb{R}^n$  e  $b \in \mathbb{R}^m$  sono vettori in input di costanti reali, mentre  $x = \text{col}(x_1, \dots, x_n)$  è il vettore di variabili di cui vogliamo trovare i valori ottimali.

Nel definire i vincoli esiste una convenzione: se l'obiettivo è una massimizzazione, allora esprimiamo i vincoli tutti del tipo  $\geq$ ; se l'obiettivo è una minimizzazione, esprimiamo i vincoli tutti come  $\leq$ .

- Gli insiemi  $I, E$  sono partizioni di  $[m]$ ;
- Gli insiemi  $C, U$  sono partizioni di  $[n]$ ;
- I vincoli in  $I$  sono diseguaglianze lineari;
- I vincoli in  $E$  sono equazioni lineari;
- Le variabili in  $C$  sono vincolate nel segno (non possono assumere valore negativo);
- Le variabili in  $U$  sono chiamate non-vincolate e possono assumere qualsiasi valore reale.

Un altro modo di scrivere il problema è tramite la forma matriciale. Sia  $A$  una matrice le cui righe sono i vettori  $a_1^T, \dots, a_m^T$ , ovvero i vettori dei vincoli precedenti trasposti ( $n$  colonne,  $m$  righe).

Supponiamo di mettere prima le righe relative alle diseguaglianze  $|I| = m_1$ , poi quelle delle equazioni  $|E| = m_2$ . Ripartizioniamo inoltre le colonne tale che riordiniamo le variabili tali che mettiamo prima le negative e poi le non ristrette (che possono essere pos. o neg.).

A questo punto possiamo partizionare  $A$  in quattro sottomatrici:

$$A = \begin{pmatrix} A^{11} & A^{12} \\ A^{21} & A^{22} \end{pmatrix}$$

con le seguenti dimensioni:

- $A^{11} \in \mathbb{R}^{m_1 \times n_1}$
- $A^{12} \in \mathbb{R}^{m_1 \times n_2}$
- $A^{21} \in \mathbb{R}^{m_2 \times n_1}$
- $A^{22} \in \mathbb{R}^{m_2 \times n_2}$

Similmente partizioniamo il vettore dei costi  $c^T$  mettendo prima i costi delle variabili non negative ( $c_1^T$ ) e poi di quelle non ristrette ( $c_2^T$ ) in  $c^T = (c_1^T, c_2^T)$ .

Inoltre,  $x$  lo riordiniamo come  $x = \text{col}(x^1, x^2)$  rispettivamente variabili non neg. e non ristrette.

Infine, partizioniamo  $b$  in  $b^1 \in \mathbb{R}^{m_1}$  e  $b^2 \in \mathbb{R}^{m_2}$ .

Il problema iniziale può essere riscritto nel seguente modo:

$$\max c_1^T x^1 + c_2^T x^2$$

soggetto ai vincoli:

$$\begin{aligned} A^{11}x^1 + A^{12}x^2 &\leq b^1 \\ A^{21}x^1 + A^{22}x^2 &= b^2 \\ x^1 &\geq 0, x^2 \geq 0 \end{aligned}$$

Ora, se  $E = U = \emptyset$  (niente equazioni e niente variabili svincolate in segno) allora rimangono solo variabili non neg. e vincoli di disequazioni. Allora il problema è detto in **forma canonica** e può essere espresso come:

$$\max c^T x$$

soggetto ai vincoli;

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \end{aligned}$$

riscrivibile nella forma compatta seguente:  $\max\{c^T x : Ax \leq b, x \geq 0\}$  (per il caso di massimizzazione, per il caso di minimizzazione il max è sostituito da un min e i  $\leq$  con  $\geq$ ).

Quando  $I = U = \emptyset$  (niente disequazioni e variabili tutte positive) allora il problema è in **forma standard**. Può essere espresso in forma seguente:

$$\max c^T x$$

soggetto a vincoli

$$Ax = b$$

$$x \geq 0$$

riscrivibile nella forma compatta seguente:  $\max\{c^T x : Ax = b, x \geq 0\}$

Riassumendo:

- **forma canonica:** tutte disequazioni, variabili tutte non negative;
- **forma standard:** tutte equazioni, variabili tutte non negative;

**Trasformare una forma in un'altra:** La forma standard e canonica sono equivalenti (ciascuna di esse può essere ricondotta all'altra forma).

Introducendo una variabile non negativa (detta "slack") si può trasformare una disequazione in equazione. Similmente una diseguaglianza si può trasformare in una equazione introducendo una variabile non negativa (detta di "surplus").

(slack): diseq.  $\mapsto$  eq.  $a_i^T x \leq b_i$  diventa  $a_i^T x + z_i = b_i$

(surplus): diseq.  $\mapsto$  eq.  $a_i^T x \geq b_i$  diventa  $a_i^T x - z_i = b_i$

Inoltre, ogni variabile  $x_j$  non vincolata può essere rimpiazzata dalla differenza di due nuove variabili non negative  $x_j^+ - x_j^-$  così da poter lavorare solo con variabili non negative.

Per passare invece da un problema di massimizzazione a uno di minimizzazione basta moltiplicare i costi per  $-1$  (e viceversa).

## 5.1 Algoritmo del simplesso

Sia  $P$  un generico LP (programma lineare) in forma standard.

$$P := \max\{c^T x : Ax = b, x \geq 0\} \quad (1)$$

Assumiamo inoltre, senza perdita di generalità, che le righe di  $A$  siano linearmente indipendenti (lineare indip: vedi 14.1 – nessuna equazione che dipende da altre equazioni: si noti che se anche la parte destra è combinazione lineare, allora l'intera equazione è rimovibile. Nel caso in cui solo la parte sx sia combinazione lineare di un'altra equazione, ma la parte destra no, allora il sistema è impossibile).

Quindi, se  $A$  ha  $m$  righe e  $d$  colonne, allora  $d = m + n$  per qualche  $n \geq 0$  (ho più colonne che righe, per via del rango di  $A$  (rango = numero di colonne e righe lin. indipendenti – coincide) e l'assunzione fatta ( $A$  rango pieno)).

**Base:** qualsiasi insieme  $B$  di  $m$  indici tale che le colonne  $A^j$  per  $j \in B$  sono linearmente indipendenti (una base di  $\mathbb{R}^m$ ) è detta **base**. L'ordine degli elementi in  $B$  conta per l'algoritmo del simplex, quindi assumiamo di poter distinguere quale elemento è il primo, secondo, etc.  $B$  è quindi una sequenza  $B = \{B[1], \dots, B[m]\}$ .

**Variabili basiche e non basiche:** data una base  $B$ , chiamiamo  $N := [m+n] \setminus B$  gli indici delle colonne non nella base. Le variabili  $x_j$  con  $j \in B$  sono dette variabili basiche, quelle per cui vale  $j \in N$  sono dette non basiche.

In corrispondenza di una base  $B$ , posso partizionare la matrice  $A$  in due matrici:

- una matrice  $A_B$  quadrata  $m \times m$  contenente le colonne in  $B$ ;
- una matrice  $A_N$   $m \times n$  contenente le colonne in  $N$ .

Allo stesso modo partizioniamo le variabili in due sottovettori  $x_B$  e  $x_N$ , uno delle variabili basiche e uno delle non basiche.

Ugualmente posso partizionare la funzione di costo  $c$  in  $c_B$  e  $c_N$ . Allora il sistema  $Ax = b$  può essere riscritto come:

$$A_N x_N + A_B x_B = b$$

Notiamo che  $A_B$  è invertibile (le sue colonne sono tutte lin. indip.), quindi, moltiplicando a sx e dx per l'inversa di  $A_B$ , ovvero per  $A_B^{-1}$  ottengo l'equivalente sistema:

$$A_B^{-1} A_N x_N + x_B = A_B^{-1} b$$

Premoltiplicando per  $A_B^{-1}$  porta il sistema nella forma di Echelon (14.2 – ciascuna variabile in  $x_B$  base ha coefficiente 1 in una equazione e 0 nelle altre). Considerando le variabili non basiche  $x_N$  come variabili libere, e le basiche come dipendenti. Se le variabili libere hanno qualche valore  $v_N$ , le variabili basiche devono essere:

$$x_B := A_B^{-1} b - A_B^{-1} A_N v_N$$

Si ha quindi che le variabili di base dipendono da quelle non di base. Ora, supponiamo di rinominare le matrici.  $\bar{A} := A_B^{-1} A_N$  e  $\bar{b} := A_B^{-1} b$ . Allora, data la base  $B$ , il problema (1) può essere riformulato come segue:

$$\max \{ c_B^T x_B + c_N^T x_N : \bar{A} x_N + x_B = \bar{b}, x_N \geq 0, x_B \geq 0 \}$$

In base all'equazione precedente, otteniamo  $x_B = \bar{b} - \bar{A} x_N$  (è forzato). Possiamo quindi rimpiazzare  $x_B$  con  $\bar{b} - \bar{A} x_N$  ovunque, otteniamo quindi (eseguendo tutte le sostituzioni alla forma precedente):

$$\max \{ c_B^T (\bar{b} - \bar{A} x_N) + c_N^T x_N : \bar{A} x_N \leq \bar{b}, x_N \geq 0 \}$$

Se definiamo:

$$\bar{c}_N^T := c_N^T - c_B^T \bar{A}$$

(dove  $\bar{c}_N$  è il **vettore dei costi ridotti delle variabili non basiche**) allora la funzione obiettivo della forma precedente può essere riscritta come:

$$c_B^T \bar{b} + \bar{c}_N^T x_N$$

dove la parte  $c_B^T \bar{b}$  è una costante (dipendente dalla specifica base  $B$  scelta – essendo una parte costante si può portare fuori dalla funz. obiettivo) mentre  $\bar{c}_N^T x_N$  dovrebbe essere ottimizzata su tutti i possibili valori di  $x_N$ , risolvendo un nuovo problema  $\mathcal{P}(B)$  definito come:

$$\mathcal{P}(B) := \max \{ \bar{c}_N^T x_N : \bar{A}x_N \leq \bar{b}, x_N \geq 0 \} \quad (2)$$

**In conclusione:** per ciascuna base  $B$  scelta possiamo riformulare il problema iniziale

$$(1) : P := \max \{ c^T x : Ax = b, x \geq 0 \}$$

come un nuovo problema LP di dimensione inferiore, in forma canonica, sulle variabili non basiche  $x_N$ , definito da:

$$(2) : \mathcal{P}(B) := \max \{ \bar{c}_N^T x_N : \bar{A}x_N \leq \bar{b}, x_N \geq 0 \}$$

Una volta trovata la soluzione  $x_N^*$  di  $\mathcal{P}(B)$ , la soluzione ottimale di  $P$  è ottenuta impostando  $x_B$  a  $\bar{b} - \bar{A}x_N^*$  e  $x_N$  a  $x_N^*$ . Il valore ottimale di  $P$  è  $c_B^T \bar{b} + \bar{c}_N^T x_N^*$ .

Sostanzialmente, il simplex afferma che per risolvere un problema LP in forma standard di dimensione  $n$  vincoli e  $m$  variabili, allora data la base  $n \times n$ , il problema iniziale può essere trasformato in un nuovo problema con sempre  $n$  vincoli e  $m - n$  variabili (quelle fuori base) in forma canonica.

(2) :  $\mathcal{P}(B)$  è semplicemente una riscrittura di (1) :  $\mathcal{P}$ , ma potrebbe essere molto più semplice da risolvere, se scegliamo la giusta base  $B$ . Consideriamo una base per cui l'origine è la più semplice soluzione ed è una soluzione ottima per  $\mathcal{P}(B)$ .

Assumiamo  $B$  tale che:

- (i)  $\bar{b} \geq 0$
- (ii)  $\bar{c}_N \leq 0$

Dato che  $\bar{A}\bar{0} = 0 \leq \bar{b}$ , la soluzione  $x_N := 0$  è fattibile per  $\mathcal{P}(B)$ . Inoltre,  $x_N := 0$  è anche ottimale per  $\mathcal{P}(B)$ . Infatti, il valore obiettivo è 0, dato che  $\bar{c}_j \leq 0$  per ogni  $j \in N$ , per ogni  $x_N \geq 0$   $\bar{c}_N^T x_N = \sum_{j \in N} \bar{c}_j x_j \leq 0$ .

La scelta della base  $B$  migliore di tutte (rispetto alla quale  $\mathcal{P}(B)$  diventa banale) è quella che fa sì che rispetti la condizione (i) e (ii).

La soluzione di  $\mathcal{P}$  corrispondente alla soluzione banale di  $\mathcal{P}(B)$  è  $x_N := \mathbf{0}, x_B := \bar{b}$ . Chiamiamo una soluzione di questo tipo **soluzione basica**.

**Soluzione di base (basica):** una soluzione di base di  $\mathcal{P}$  si ottiene scegliendo una base  $B$ , impostando a 0 tutti gli  $x_j$  per  $j \notin B$ , e risolvendo il sistema per le variabili rimanenti. Se  $x_j \geq 0$  per ogni  $j \in B$ , allora  $B$  è chiamata base fattibile e la soluzione basica  $x_B := \bar{b}, x_N := 0$

è una soluzione basica fattibile/ammissibile ( bfs). Inoltre, se  $x_j > 0$  per ogni  $j \in B$  diciamo che la base  $B$  e la bfs  $x$  sono **non-degeneri**, se  $x_j = 0$  per qualche  $j \in B$ , entrambi  $B$  e la bfs  $x$  sono **degeneri**. Si può dimostrare che le condizioni (i) e (ii) sono sempre soddisfatte da qualche base  $B$  purché esista una soluzione ottima, ovvero:

Se  $P$  ammette un ottimo, allora esiste sempre una base  $B$  tale che  $P(B)$  ha soluzione ottima banale 0 (esiste una bfs che è una soluzione ottima per  $P$ ).

Sotto questa tesi, possiamo reinterpretare la ricerca di una soluzione ottima per  $P$  come la ricerca di una base  $B^*$  per cui  $P(B^*)$  sia banale. In particolare, il nostro obiettivo è di cercare una base fattibile per cui i costi ridotti delle variabili non basiche siano tutti non positivi. Per trovare questa base usiamo il metodo del simplex.

Il numero di basi possibili è limitato da  $\binom{m+n}{m}$ , quindi abbiamo tradotto un problema continuo con infinite soluzioni in un problema discreto con un numero finito (ma enorme) di soluzioni da considerare. L'algoritmo del simplex è una procedura iterativa di ricerca, che parte da una base fattibile  $B$  e si muove di base in base affinché una base ottimale  $B^*$  non viene raggiunta.

Il metodo del simplex non itera tra le varie bfs ma piuttosto rispetto le varie basi. Ovvero, il simplex cambia la base ma non necessariamente la bfs, in quanto ciascuna base determina unicamente una bfs, ma non è vero il contrario (due basi possono determinare la stessa bfs – sarebbe vero se fossero tutte non degeneri). Quindi, per presenza di degenerazione, più basi degeneri possono identificare la stessa bfs.

**Pivoting da base a base:** durante la ricerca della base ottimale, il simplex visita una sequenza di basi  $B^1, B^2, \dots, B^k$ . Ciascuna  $B^k$  è ottenuta dalla base  $B^{k-1}$  rimpiazzando una colonna  $A^j, j \in B^{k-1}$  con una colonna  $A^t, t \notin B^{k-1}$ . Diciamo che  $t$  entra nella base e  $j$  esce dalla base, e questo scambio è detto **pivoting**. Ogni coppia di base che differisce di un indice è detta **coppia di basi adiacenti**.

Per scegliere quale base entra, il simplex usa una regola di pivoting. La regola più adottata segue un approccio greedy: La variabile non basica  $x_j$  che diventa base è quella per cui il costo ridotto  $\bar{c}_j$  è massimizzato.

Esistono altre tecniche ma è importante che la variabile in entrata sia sempre con un costo ridotto positivo. Questa è una condizione greedy (sempre in meglio). Una volta che il simplex trova questa variabile con costo ridotto positivo, il simplex aggiorna la base in modo da includere questa variabile.

**Impostare il valore della variabile in entrata:** Aumentare  $j$  da 0 a  $\epsilon$  mantenendo a 0 le altre variabili significa che nel problema  $P(B)$  stiamo considerando la soluzione  $\text{col}(0, \dots, 0, \epsilon, 0, \dots, 0) = \epsilon\mathbf{e}^j$ . Questa soluzione è fattibile se:

$$\bar{A}(\epsilon\mathbf{e}^j) \leq \bar{b}$$

ovvero:

$$\epsilon \bar{A}^j = \begin{pmatrix} \epsilon \bar{a}_{1j} \\ \vdots \\ \epsilon \bar{a}_{mj} \end{pmatrix} \leq \begin{pmatrix} \bar{b}_1 \\ \vdots \\ \bar{b}_m \end{pmatrix}$$

Il miglioramento dalla soluzione 0 rispetto  $\epsilon \mathbf{e}^j$  è  $\bar{c}_j \epsilon$  e quindi è meglio prendere  $\epsilon$  più grande possibile. Se  $\bar{a}_{ij} \leq 0$  per ogni  $i = 1, \dots, m$ , allora potremmo prendere  $\epsilon$  più grande possibile finché rimane fattibile. Perciò, il problema è illimitato e il simplex si ferma con un errore. Altrimenti, si consideri tutti i valori  $i$  t.c.  $\bar{a}_{ij} > 0$ . Ciascun di questi definisce un upper bound a  $\epsilon$ , cioè,  $\epsilon$  non può essere più grande di  $\bar{b}_i / \bar{a}_{ij}$ . L'upper bound minimo è il valore massimo  $\hat{\epsilon}$  che  $\epsilon$  può prendere, cioè:

$$\hat{\epsilon}_\Delta := \min \left\{ \frac{\bar{b}_i}{\bar{a}_{ij}} : i \in [m] : \bar{a}_{ij} > 0 \right\}$$

Sia  $k \in \{1, \dots, m\}$  t.c.  $\hat{\epsilon} := \frac{\bar{b}_k}{\bar{a}_{kj}}$ . Allora  $B[k]$  è la variabile che lascia la base, e il simplex aggiorna la base corrente a  $B := B \setminus \{B[k]\} \cup \{j\}$  che, se  $B$  è salvata in memoria come array, può essere raggiunta semplicemente sovrascrivendo il  $k$ -esimo elemento, cioè  $B[k] := j$ .

**Degenerazione e ciclaggio:** Se nessuna base viene mai ripetuta, l'algoritmo deve eventualmente interrompersi, poiché esiste un numero finito di basi possibili. Ad esempio, nessuna base verrà ripetuta se la base attuale non è mai degenerata. Infatti, il valore dell'attuale bfs aumenta di  $\bar{c}_j \hat{\epsilon} > 0$  in modo che i bfs visitati migliorino monotonicamente e quindi devono essere tutti diversi. Tuttavia, possono essere possibili basi degenerate. In presenza di degenerazione, il simplex potrebbe entrare in un loop infinito perché è possibile che, dopo il pivot, la base sia cambiata ma il bfs sia lo stesso. Dopo una sequenza di pivot degeneri  $(j_1, B[k_1]), \dots, (j_t, B[k_t])$  in cui  $j_i$  entra nella base e  $B[k_i]$  esce, potremmo tornare alla base di partenza. A questo punto l'algoritmo andrebbe in loop per sempre. Il cycling può essere evitato scegliendo la variabile che lascia la base a caso (si può dimostrare che, prima o poi, faremo una scelta che interrompe il loop) o utilizzando una regola deterministica nota come regola di Blands: scegli tra tutte le variabili di costo positivo ridotto poi quella con l'indice più piccolo (la variabile entrante) e scelgono il valore più piccolo possibile  $B[k]$  tra tutte le variabili candidate a lasciare la base.

**Simplex a 2 fasi:** Per determinare una bfs iniziale usiamo il simplex a 2 fasi. Supponiamo che tutte le entries di  $b$  siano non negative (possibilmente moltiplicando alcune equazioni per -1). L'algoritmo inizia risolvendo un LP ausiliario, definito come:

$$\max \{ \mathbf{0}^T [x - \mathbf{1}^T y : Ax + y = b, x \geq \mathbf{0}, y \geq \mathbf{0}] \}$$

Se originariamente abbiamo  $m + n$  variabili, il problema ausiliario ha  $2m + n$  variabili e una base ammissibile  $\hat{B}$  sono le ultime  $m$  colonne. Notare che la soluzione ottimale deve essere  $\leq 0$ .

**FASE 1:** Risolvi il problema ausiliario con il simplex iniziando con la base ammissibile  $\hat{B}$ . Il problema originale è fattibile se e solo se il valore ottimale del problema ausiliario è 0. Se il valore ottimale è negativo, l'algoritmo si ferma e segnala l'impossibilità.

**FASE 2:** Altrimenti, alla fine della fase 1  $\hat{B}$  è una base ottimale per il problema ausiliario e  $x$  è una bfs tale che  $x_j = 0$  per tutti  $j > d$ . Supponiamo che  $k$  delle colonne di  $\hat{B}$  siano anche nella matrice originale  $A$ . Completare queste colonne con  $m - k$  colonne da  $A$  per ottenere una base fattibile  $B$  per il problema originale. Procedere quindi con la 2 a fase risolvendo l'LP originale a partire da bfs  $B$ .

## 5.2 Esempio esecuzione algoritmo del simplex

Sia, dati  $m = 3, n = 2, d = 5, P$  il seguente problema LP: sia

$$\max\{2x_1 - 4x_2 + x_3 - 2x_4 - x_5\}$$

la funzione obiettivo, e a seguire i vincoli:

$$\begin{array}{ccccccc} 2x_1 & +3x_2 & -2x_3 & +5x_4 & -x_5 & = 3 \\ x_1 & & +x_3 & +x_4 & -x_5 & = 4 \\ -x_1 & +2x_2 & +x_3 & +x_4 & +3x_5 & = 1 \end{array}$$

$$x_1, \dots, x_5 \geq 0$$

Questi dati, sono così riassunti in forma matriciale:

$$c^T = (2 \ -4 \ 1 \ 2 \ -1) \\ A = \begin{pmatrix} 2 & 3 & -2 & 5 & -1 \\ 1 & 0 & 1 & 1 & -1 \\ -1 & 2 & 1 & 1 & 3 \end{pmatrix} \quad b = \begin{pmatrix} 3 \\ 4 \\ 1 \end{pmatrix}$$

Ricapitoliamo quanto detto nella teoria: definiamo una base come un insieme di  $m$  indici di colonne, tali che esse siano linearmente indipendenti. Se la base è  $B$ ,  $N$  è l'insieme delle altre colonne. Le  $m$  variabili delle colonne  $B$  sono dette variaibiali basiche, le altre (di  $N$ ) non basiche.

Scegliamo le colonne  $B = \{1, 3, 4\}$  e  $N = \{2, 5\}$ .

Una volta identificata la base, partizioniamo  $A$ ,  $x$  e  $c$  come specificato nella teoria. Una volta partizionati, possiamo riscrivere il problema:

$$\max c^T x$$

$$Ax_b = b \quad x \geq 0$$

come:

$$\max c_B^T x_B + c_N^T x_N \\ x_B + A_B^{-1} A_N x_N = A_B^{-1} b, \quad x_B, x_N \geq 0$$

Quindi:

$$A_B = \begin{pmatrix} 2 & -2 & 5 \\ 1 & 1 & 1 \\ -1 & 1 & 1 \end{pmatrix} \quad A_N = \begin{pmatrix} 3 & -1 \\ 0 & -1 \\ 2 & 3 \end{pmatrix} \quad A_B^{-1} = \begin{pmatrix} 0 & 1/2 & -1/2 \\ -1/7 & 1/2 & 3/14 \\ 1/7 & 0 & 2/7 \end{pmatrix}$$

$$c_B^T = (2, 1, 2) \quad c_N^T = (-4, -1)$$

Eseguendo i conti ottengo:

$$A_B^{-1}A = \begin{pmatrix} 1 & -1 & 0 & 0 & -2 \\ 0 & 0 & 1 & 0 & 2/7 \\ 0 & 1 & 0 & 1 & 5/7 \end{pmatrix} \quad A_B^{-1}b = \begin{pmatrix} 3/2 \\ 25/14 \\ 5/7 \end{pmatrix}$$

Riscrivendo il problema iniziale ottengo:

$$\max 2x_1 + x_3 + 2x_4 - 4x_2 - x_5$$

$$\begin{aligned} x_1 &= \frac{3}{2} + x_2 + 2x_5 \\ x_3 &= \frac{25}{14} - \frac{2}{7}x_5 \\ x_4 &= \frac{5}{7} - x_2 - \frac{5}{7}x_5 \end{aligned}$$

$$x_2, x_5 \geq 0, x_1, x_2, x_4 \geq 0$$

Introduciamo nuovi nomi:

- la matrice  $m \times n$   $\bar{A} := A_B^{-1}A_N$   $\bar{A} = \begin{pmatrix} -1 & -2 \\ 0 & 2/7 \\ 1 & 5/7 \end{pmatrix}$
- il vettore  $m \times 1$   $\bar{b} := A_B^{-1}b$   $\bar{b} = \begin{pmatrix} 3/2 \\ 25/14 \\ 5/7 \end{pmatrix}$
- il vettore  $n \times 1$   $\bar{c}_N^T := c_N^T - c_B^T \bar{A}$  chiamato vettore dei costi ridotti sulle variabili non basiche.

$$\bar{c}_N^T = (-4 \ -1) - (2 \ 1 \ 2) \begin{pmatrix} -1 & -2 \\ 0 & 2/7 \\ 1 & 5/7 \end{pmatrix} = (-4 \ 16/7)$$

Poi, rimpiazzando  $x_B$  con  $\bar{b} - \bar{A}_{x_N}$  ovunque, il problema può essere riscritto come:

$$\max c_B^T \bar{b} + \bar{c}_N^T x_N$$

tale che:

$$\bar{A}_{x_N} \leq \bar{b}, x_N \geq 0$$

Si noti come la parte  $c_B^T \bar{b}$  nella funzione obiettivo sia costante (dipendente da  $B$ ). Nel nostro caso è:

$$c_B^T \bar{b} = (2 \ 1 \ 2) \begin{pmatrix} 3/2 \\ 25/14 \\ 5/7 \end{pmatrix} = 87/14$$

Abbiamo quindi riscritto il problema in un nuovo problema in forma canonica (da standard) le cui variabili sono solo quelle fuori base ( $x_N$ ) e in cui la funzione obiettivo possiede una parte costante che si può "portare fuori".

Ricapitolando, il nostro problema era:

$$\max 2x_1 - 4x_2 + x_3 + 2x_4 - x_5$$

tale che:

$$\begin{array}{ccccccc} 2x_1 & +3x_2 & -2x_3 & +5x_4 & -x_5 & = 3 \\ x_1 & & +x_3 & +x_4 & -x_5 & = 4 \\ -x_1 & +2x_2 & +x_3 & +x_4 & +3x_5 & = 1 \\ & & & & x_1, \dots, x_5 \geq 0 & \end{array}$$

che è diventato, dopo aver scelto di esprimere secondo la base  $B = \{1, 3, 4\}$ :

$$\left( \frac{87}{14} + \right) \max -4x_2 + \frac{16}{7}x_5$$

tale che:

$$\begin{array}{lll} -x_2 - 2x_5 \leq & \frac{3}{2} \\ \frac{2}{7}x_5 \leq & \frac{25}{14} \\ x_2 + \frac{5}{7}x_5 \leq & \frac{5}{7} \\ x_2, x_5 \geq & 0 \end{array}$$

(con  $x_1 = \frac{3}{2} + x_2 - x_5, x_3 = \frac{25}{14} - \frac{2}{7}x_5, x_4 = \frac{5}{7} - x_2 - \frac{5}{7}x_5$ ). Passando da dimensione 5 a dimensione 2. (Abbiamo proiettato il nostro problema da spazio 5 a spazio 2).

### 5.3 Interpretazione geometrica del simplex

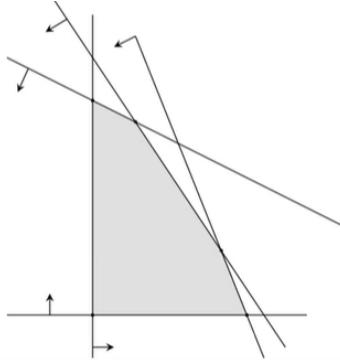
Dato  $\alpha \in \mathbb{R}^n$  e  $\beta \in \mathbb{R}$ , consideriamo la diseguaglianza  $\alpha^T x \leq \beta$ . L'insieme  $HS(\alpha, \beta) := \{v : \alpha^T v \leq \beta\}$  è chiamato semi spazio di  $\mathbb{R}^n$  e i suoi confini  $H(\alpha, \beta) := \{v : \alpha^T v = \beta\}$  determinano quello che viene chiamato iperpiano con vettore normale  $\alpha$ .

Un **poliedro** è l'intersezione di un numero finito di semi spazi. L'insieme fattibile di ogni programma lineare è un poliedro. Un poliedro  $P$  è limitato se  $\exists R \in \mathbb{R}$  tale che  $\max_{i=1, \dots, n} \|x_i\| \leq R$  per ogni  $x \in P$  (ogni punto sta all'interno di una distanza  $R$ ). Un poliedro limitato è anche chiamato **politopo**. Segue ad immagine una versione bidimensionale di un poliedro.

**Dimensione di un poliedro:** Sia  $P \subseteq \mathbb{R}^n$  un poliedro tale che  $\mathbf{0} \in P$ . Diciamo che  $P$  ha dimensione  $d$  se il più piccolo sotto spazio di  $\mathbb{R}^n$  contenente  $P$  ha dimensione  $d$ .

Se  $d = n$  allora  $P$  è di **dimensione piena**. Se  $\mathbf{0} \notin P$  definiamo la dimensione di  $P$  attraverso una traslazione rigida che muove  $P$  tale che includa l'origine. Vale a dire che se  $P = \{x : Ax \leq b\} \neq \emptyset$  è un poliedro qualsiasi e  $v_0 \in P$  è qualsiasi dei suoi punti, allora

$$P_0 := \{x \in \mathbb{R}^n : v_0 + x \in P\} = \{x : A(v_0 + x) \leq b\} = \{x : Ax \leq b'\}$$



dove  $b' := b - Av_0$ , è a sua volta un poliedro (possiamo scrivere anche  $P_0 := P - v_0$ ) , che per sua definizione, contiene l'origine. Allora la dimensione di  $P$  è definita essere la stessa di  $P_0$ .

Esempi:

- Un cubo è un politopo a 3 dimensioni, di dimensione piena in  $\mathbb{R}^3$  ma non in  $\mathbb{R}^4$ ;
- Un quadrato in  $\mathbb{R}^2$ , tale che  $\{(x, y) : 1 \leq x \leq 2, 1 \leq y \leq 2\}$  è un politopo di dimensione piena. Un quadrato in  $\mathbb{R}^3$  t.c.  $\{(x, y, z) : 1 \leq x \leq 2, 1 \leq y \leq 2, z = 1\}$  è ancora un politopo bi-dimensionale ma non è pienamente dimensionato;
- Il segmento tra due punti è un politopo a 1 dimensione pienamente dimensionato solo in  $\mathbb{R}$ , che in questo caso è un intervallo chiuso;
- Un punto è un politopo a 0 dimensioni e non è mai a dimensione piena.

**Esempio:** Sia  $P$  il politopo in  $\mathbb{R}^3$  definito da:

$$P = \{(x_1, x_2, x_3) : x_1 + x_2 + x_3 = 1, x_1 \geq 0, x_2 \geq 0, x_3 \geq 0\} \quad (1)$$

Sia  $v_0 = (0, 0, 1)$ . Poi:

$$\begin{aligned} P_0 &= \{(x_1, x_2, x_3) : (x_1, x_2, x_3) + (0, 0, 1) \in P\} \\ &= \{(x_1, x_2, x_3) : x_1 + x_2 + x_3 + 1 = 1, x_1 \geq 0, x_2 \geq 0, x_3 + 1 \geq 0\} \\ &= \{(x_1, x_2, x_3) : x_1 + x_2 + x_3 = 0, x_1 \geq 0, x_2 \geq 0, x_3 \geq -1\} \\ &= \{(x_1, x_2, -(x_1 + x_2)) : x_1 \geq 0, x_2 \geq 0, x_1 + x_2 \leq 1\} \end{aligned}$$

Chiaramente,  $P_0$  è contenuto nello spazio bidimensionale  $V$  generato da  $(1,0,-1)$  e  $(0,1,-1)$ , ovvero:

$$\{x_1(1, 0, -1) + x_2(0, 1, -1) : x_1, x_2 \in \mathbb{R}\}$$

e, prendendo, ad esempio,  $x_1 = x_2 = 1/2$  abbiamo due vettori linearmente indipendenti di  $P_0$ . Quindi, il poliedro  $P_0$  e  $P$  sono bidimensionali (in particolare,  $P$  è un triangolo in  $\mathbb{R}^3$ ).

Sia  $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ . Una diseguaglianza  $a_i x \leq b_i$  di  $Ax \leq b$  è chiamata equazione implicita se non può essere soddisfatta come una diseguaglianza strict da ogni punto di  $P$ . Se il sistema contiene la diseguaglianza  $3x_1 - 6x_2 \leq 6$  e anche  $-4x_1 + 8x_2 \leq -8$ , allora sono entrambe implicite,  $x_1 - 2x_2 = 2$ . Riorganizziamo le diseguaglianze così da avere  $A$  partizionata in  $A^<$  e  $A^=$ , dove le equazioni implicite sono in  $A^=$ . Poi  $P$  può essere riscritta come:

$$P = \{x : A^<x \leq b^<, A^=x = b^=\}$$

Si può dimostrare che  $\dim(P) = n - \text{rank}(A^=)$ . Questo implica che per un politopo pienamente dimensionato non possono esserci equazioni implicite, e che per un singolo punto ci sono  $n$  equazioni linearmente indipendenti da soddisfare. Nell'esempio precedente  $A^=$  ha solo una equazione e  $\dim(P) = 3 - 1 = 2$ .

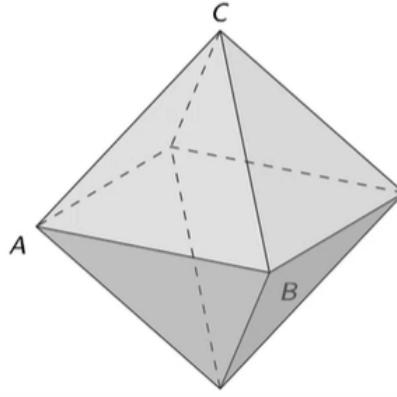
### Facce di un poliedro

Sia  $\alpha^T x \leq \beta$  una valida diseguaglianza per  $P$ , cioè, soddisfatta da tutti gli  $x \in P$ , ovvero  $P$  è contenuto nel semispazio indotto dalla disequazione.

Se l'iperpiano relativo  $H(\alpha, \beta) \cap P \neq \emptyset$  allora l'insieme  $F := H(\alpha, \beta) \cap P$  è chiamato una **faccia** di  $P$ , definita (o indotta) dalla diseguaglianza  $\alpha^T x \leq \beta$ .

È importante notare come la faccia di un poliedro (o di un politopo) è di per sé un poliedro (o politopo). Una faccia può avere dimensione 0 (se è un singolo punto – vertice), dimensione 1 (un segmento – spigolo), dimensione 2 etc.

Le facce più importanti di  $P$  sono quelle di dimensione  $\dim(P) - 1$  chiamate **faccette** (facets). In figura, i punti  $A, B$  sono facce di dimensione 0. Il segmento tra  $A$  e  $B$  è uno spigolo del poliedro come il segmento  $\overline{BC}$ . Il triangolo con vertici  $A, B, C$  è una faccetta.



Le diseguaglianze che definiscono le faccette di un poliedro devono essere presenti in ogni descrizione del poliedro; non sono mai implicate da altre diseguaglianze e quindi essenziali.

Si ha che un poliedro  $P = \{x : Ax \leq b\}$  può essere identificato da più di una coppia  $(A, b)$ . Si ha, ad esempio, che l'insieme delle soluzioni per:

$$x_1 + x_2 \leq 1, x_1 \geq 0, x_2 \geq 0$$

è lo stesso di quelle per:

$$x_1 + x_2 \leq 1, x_1 \leq 1, x_2 \leq 1, x_1 \geq 0, x_2 \geq 0$$

Alcune diseguaglianze possono essere ridondanti (implicate da altre) e quindi possono essere rimosse dal poliedro (senza alterarlo). Alcune diseguaglianze però sono fondamentali per la definizione del poliedro e se le rimuoviamo, il poliedro cambia. Ad esempio, se rimuoviamo la diseguagliaza  $x_1 + x_2 \leq 1$  da entrambi i sistemi sopra, l'insieme delle soluzioni cambia.

Le diseguaglianze fondamentali nella definizione di un poliedro sono quelle che definiscono le facette; inoltre, due poliedri sono gli stessi se e solo se le facette sono le stesse. Le diseguaglianze che inducono a facette non sono mai implicate da altre diseguaglianze.

### Vertici e soluzioni fattibili di base ( bfs)

Un vertice, o punto estremo di  $P$  è un punto  $v \in P$  tale che non ci sono  $u, w \in P$  con  $u \neq w$  tale che  $v = \lambda u + (1 - \lambda)w$  con  $\lambda \in (0, 1)$ .

Può essere mostrato che un vertice è una faccia di  $P$  di dimensione 0. Denotiamo con  $\text{ext}(P)$  l'insieme dei vertici di  $P$ . Diciamo che  $P$  è appuntito (pointed) se  $\text{ext}(P) \neq \emptyset$ , e questo succede se e solo se  $P$  non contiene nessuna linea infinita in entrambe le direzioni. Si noti come ci siano diseguaglianze  $x_i \geq 0$  per  $i = 1, \dots, n$  allora  $P$  è sempre appuntito, e ciascun politopo è sempre appuntito.

Il seguente teorema fondamentale caratterizza ciascun politopo come un insieme convesso di combinazioni di vettori piuttosto che un insieme di soluzioni di un sistema lineare di diseguaglianze.

Indichiamo con  $\text{conv}(S)$  l'insieme delle combinazioni convesse di tutti i vettori in  $S$ , detto **convex hull – involuppo convesso**.

### Teorema (Minkowski-Weil):

Un insieme  $P \subseteq \mathbb{R}^n$  è un politopo se e solo se  $P = \text{conv}(S)$  per qualche insieme finito  $S$  di vettori in  $\mathbb{R}^n$ .

Considerando il teorema, ci sono due possibili rappresentazioni di un politopo  $P$ :

- **rappresentazione esterna:** un insieme di iperpiani tali che  $P$  è intersezione dei loro semi spazi;
- **rappresentazione interna:** un insieme di vettori tali che  $P$  è la loro involuzione convessa.

Consideriamo un poliedro  $P = \{x : Ax = b, x \geq 0\}$ . Allora un teorema fondamentale che lega il simplesso ai poliedri è:

### Teorema:

Un punto  $\hat{x} \in P$  è un vertice di  $P$  se e solo se  $\hat{x} = (\hat{x}_B, \mathbf{0})$  è la bfs per una base ammissibile  $B$  di  $A$ .

### Teorema:

Siano  $v, v'$  vertici del poliedro  $P = \{x : Ax = b, x \geq 0\}$  e sia  $B$  e  $B'$  le basi corrispondenti. Allora i vertici  $v$  e  $v'$  sono connessi da uno spigolo di  $P$  se e solo se le basi  $B$  e  $B'$  sono adiacenti, ovvero:

$$|B \cap B'| = m - 1$$

---

Possiamo quindi interpretare l'algoritmo del simplesso come una visita ai vertici del poliedro delle soluzioni. Quando la bfs corrente cambia, il simplesso si muove su un vertice adiacente, con un valore migliore della funzione obiettivo, muovendosi attraverso uno spigolo di  $P$ .

Quando la base cambia ma la bfs rimane invariata (è degenero) il simplesso rimane sullo stesso vertice, ma lo considera in maniera differente (generato da una base differente). Possiamo interpretare la degenerazione come il fatto che lo stesso vertice può essere identificato in più di un modo dall'intersezione di n iperpiani.

Infatti, in dimensione 2, due linee sono sufficienti per identificare un vertice. Se un poliedro contiene ad esempio le diseguaglianze  $x_1 \leq 1, x_2 \leq 1$  e  $x_1 + x_2 \leq 2$ , allora il punto  $(1, 1)$  può essere espresso dall'intersezione di qualsiasi delle due e se fosse stato un vertice del poliedro sarebbe stato una bfs degenera.

---

Il teorema di Minkowski-Weil ci può dare dimostrazione che ogni politopo, se visto come insieme di soluzioni di un problema lineare, il problema relativo avrà sempre l'ottimo in un vertice. Abbiamo provato attraverso l'algoritmo del simplesso che, per ogni soluzione fattibile e problema LP non limitato, esiste sempre un vertice che è soluzione ottimale. Le condizioni sono certamente vere se l'insieme delle soluzioni è un politopo non vuoto, in questo caso possiamo dare il seguente teorema:

**Teorema:**

Sia  $\max \{c^T x, x \in P\}$  un LP,  $P \neq \emptyset$  un politopo. Siano  $\{v_1, \dots, v_t\}$  i vertici di  $P$ . Allora esiste  $v \in V$  t.c  $v$  è un minimo globale.

**Dimostrazione:**

Dal teorema di Minkowski-Weyl,  $P = \text{conv} (v_1, \dots, v_t)$  e per ogni  $x \in P$  esiste  $\lambda_1, \dots, \lambda_t \geq 0$ , con  $\sum_{i=1}^t \lambda_i = 1$  t.c.:

$$x = \sum_{i=1}^t \lambda_i v_i$$

Sia  $v := v_j$  dove  $c^T v_j \geq c^T v_i$  per ogni  $i = 1, \dots, t$ . Allora, per ogni  $x \in P$  è:

$$c^T x = c^T \sum_{i=1}^t \lambda_i v_i = \sum_{i=1}^t \lambda_i c^T v_i \leq \sum_{i=1}^t \lambda_i c^T v = (c^T v) \sum_{i=1}^t \lambda_i = c^T v$$

**Cambio base greedy del simplesso**

Il simplesso cambia base sempre con un approccio greedy (fa entrare una variabile a costo ridotto positivo).

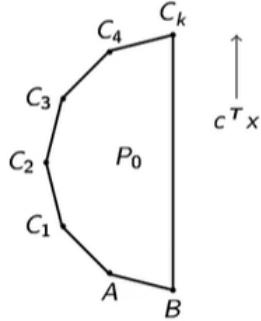
Si noti che  $P_0$  è definito da una specifica base  $B$ , e il vertice di  $P_0$  corrispondente a questa base è l'origine.

I vertici rimanenti corrispondono alle basi diverse da  $B$ . I vertici adiacenti all'origine corrispondono alle basi adiacenti a  $B$ .

L'esplorazione dei vertici di  $P$  svolta dal simplesso può essere vista come l'esplorazione dei vertici di  $P_0$  partendo dall'origine e muovendosi verso il vertice migliore.

Un esempio dove l'approccio greedy di muoversi sempre verso un vertice adiacente migliore può essere negativo: assumiamo che la funzione obiettivo sia  $\max x_2$  e che la bfs corrente sia  $A$ . Il vertice  $B$  è peggiore di  $A$ , ma se ci muoviamo da  $A$  a  $B$  poi possiamo andare a  $C_k$ , la soluzione ottimale, in due salti.

Se miglioriamo sempre (approccio greedy) la funzione obiettivo, impiegheremmo  $k$  salti.



## 5.4 Programmazione Lineare: Dualità

Sia  $P = \{x : Ax = b, x \geq 0\}$  un poliedro definito da  $m$  equazioni su  $n$  variabili. Una diseguaglianza:

$$\alpha^T x \leq \beta$$

è **valida** se  $\alpha^T \bar{x} \leq \beta$  per ogni  $\bar{x} \in P$ . Vogliamo caratterizzare tutte le diseguaglianze valide.

Quindi, una diseq. è valida se è soddisfatta da tutti i punti del poliedro (l'iperpiano lascia il poliedro tutto a sx o dx) mentre non è valida se ci sono punti del poliedro che non la soddisfano (l'iperpiano interseca il poliedro). Ci sono:

(1) **Diseguaglianze valide dirette**, ovvero combinazioni lineari delle equazioni in  $Ax = b$ . Sia  $u \in \mathbb{R}^m$  un vettore di moltiplicatori e definiamo:

$$\alpha^T := u^T A, \quad \beta := u^T b$$

Allora l'equazione  $\alpha^T x = \beta$  è sempre soddisfatta da tutti i punti in  $P$ , e quindi anche la disequazione:

$$\alpha^T x \leq \beta$$

cioè:

$$u^T A x \leq u^T b \quad (1)$$

è una diseguaglianza valida.

(2) **Diseguaglianze valide indirette**, ottenute rilassando una diseguaglianza valida diretta, cioè decrementando le parti sinistre e/o aumentando le parti destre di (1). Dato che

$x \geq \mathbf{0}$ , se  $\gamma^\top x \leq \delta$  è una diseguaglianza valida, allora per ogni  $\hat{\gamma} \in \mathbb{R}^n$  e  $\hat{\delta} \in \mathbb{R}$  t.c.  $\hat{\gamma} \leq \gamma$  e  $\hat{\delta} \geq \delta$ , abbiamo:

$$\hat{\gamma}^\top x \leq \gamma^\top x \leq \delta \leq \hat{\delta} \quad \forall x \in P$$

cioè, la diseguaglianza:

$$\hat{\gamma}^\top \mathbf{x} \leq \hat{\delta}$$

è valida.

Allora, per ciascuna scelta di  $u_1, \dots, u_m \in \mathbb{R}$ , se  $\hat{\alpha} \leq u^\top A$  e  $\hat{\beta} \geq u^\top b$  la diseguaglianza indiretta:

$$\hat{\alpha}^\top x \leq \hat{\beta}$$

è valida per  $P$ .

Ricapitolando, una diseguaglianza valida è o diretta o indiretta (ottenuta rilassando una diseguaglianza diretta).

La condizione sufficiente per ottenere una diseguaglianza valida per  $P$  (ovvero scegliere qualche  $u \in \mathbb{R}^m$  e rilassare (1)) è anche una condizione necessaria (unico modo di ottenerla), cioè, per ogni diseguaglianza valida per  $P$  c'è qualche scelta fattibile di moltiplicatori  $u_i$  tali che la diseguaglianza è una diseq. indiretta ottenuta da (1).

Non possiamo rimpiazzare "diretta" per "indiretta" nella precedente affermazione. Si consideri ad esempio:

$$2x_1 + x_3 = 1$$

$$2x_2 + 2x_3 = 1$$

Per ogni scelta  $u_1, u_2$  la diseguaglianza indiretta è ottenuta è:

$$(2u_1)x_1 + (2u_2)x_2 + (u_1 + 2u_2)x_3 \leq u_1 + u_2$$

La diseguaglianza:

$$x_1 + x_2 + x_3 \leq 1 \quad (2)$$

è valida, ma non può essere una diseguaglianza diretta perché dovrebbe essere  $u_1 = u_2 = 1/2$ , ma anche  $u_1 + 2u_2 = 1$ . Inoltre, prendendo  $u_1 = u_2 = 1/2$  abbiamo la diseguaglianza diretta:

$$x_1 + x_2 + \frac{3}{2}x_3 \leq 1$$

e rilassando la parte sinistra possiamo ottenere (2).

### Esempio di diseguaglianza diretta:

Si prenda il sistema:

$$\begin{aligned} 2x_1 - 3x_2 + x_3 &= 1 \\ -x_1 + 2x_2 + 2x_3 &= -2 \end{aligned}$$

Se moltiplichiamo la prima equazione con  $u_1 = 3$  diventa:

$$6x_1 - 9x_2 + 3x_3 = 3$$

Se moltiplichiamo la seconda equazione con  $u_1 = -1$  diventa:

$$x_1 - 2x_2 - 2x_3 = 2$$

Sommando otteniamo:

$$7x_1 - 11x_2 + x_3 = 5$$

e finalmente la diseguaglianza diretta rilassando = con  $\leq$ :

$$7x_1 - 11x_2 + x_3 \leq 5$$

### Esempio di diseguaglianza indiretta:

Con i moltiplicatori  $(u_1, u_2) = (3, -1)$ , dal sistema

$$\begin{aligned} 2x_1 - 3x_2 + x_3 &= 1 \\ -x_1 + 2x_2 + 2x_3 &= -2 \end{aligned}$$

otteniamo la diseguaglianza diretta:

$$7x_1 - 11x_2 + x_3 \leq 5$$

Ora possiamo rilassarla e ottenere, le seguenti diseguaglianze indirette:

$$5x_1 - 11x_2 + \frac{1}{2}x_3 \leq 6$$

o

$$3x_1 - 15x_2 \leq 8$$

o

$$x_1 - 11x_2 - x_3 \leq 5$$

etc. etc.

Vogliamo provare il risultato principale riguardo le diseguaglianze valide, ad esempio che  $\alpha^T x \leq \beta$  è una diseguaglianza valida sse esistono moltiplicatori  $u \in \mathbb{R}^m$  tali che  $\alpha^T \leq u^T A$  e  $\beta \geq u^T b$ . Dichiariamo questo teorema in una forma chiamata **lemma di Farkas**, o il **lemma delle alternative**, che dice che un dato sistema di diseguaglianze ha soluzione sse un relativo sistema di diseguaglianze non ha soluzioni.

**Teorema (Farkas' Lemma):** Sia  $A \in \mathbb{R}^{m \times n}$ ,  $b \in \mathbb{R}^m$ ,  $\alpha \in \mathbb{R}^n$  e  $\beta \in \mathbb{R}$  e si assuma che il poliedro  $P = \{x : Ax = b, x \geq 0\}$  sia non vuoto. Allora il sistema:

$$\begin{array}{ll} Ax &= b \\ \alpha^T x &> \beta \\ x &\geq 0 \end{array}$$

nelle variabili  $x_1, \dots, x_n$  non ha soluzioni (cioè nessun  $x$  in  $P$  (dalla 1 e 3) t.c.  $\alpha^T x > \beta$ , e quindi che  $\alpha^T x \leq \beta$  è una diseguaglianza valida per  $P$ ) sse il sistema:

$$\begin{array}{l} u^T A \geq \alpha^T \\ u^T b \leq \beta \end{array}$$

nelle variabili  $u_1, \dots, u_m$  ha (almeno una) soluzione.

**Interpretazione:** Una diseguaglianza è valida per un poliedro sse è un rilassamento di una diseguaglianza diretta.

**Riscrittura compatta:**

$$\{x \in \mathbb{R}^n : Ax = b, \alpha^T x > \beta, x \geq 0\} = \emptyset \iff \{u \in \mathbb{R}^m : u^T A \geq \alpha^T, u^T b \leq \beta\} \neq \emptyset$$

### Diseguaglianze valide per LPs canonici

Similmente possiamo mostrare che tutte le diseguaglianze valide per un poliedro  $P^c$  che sono soluzioni fattibili di un programma lineare in forma canonica, ovvero:

$$P^c = \{x : Ax \leq b, x \geq 0\}$$

possono essere ottenute come diseguaglianze indirette da (1) solo che questa volta i moltiplicatori  $u$  devono essere tutti non negativi.

### Duale di un programma lineare

Si consideri un problema LP con  $m$  vincoli e  $n$  variabili  $\geq 0$  in forma standard:

$$z_P = \max\{c^T x : Ax = b, x \geq 0\} \quad (8)$$

Denotiamo l'insieme delle sue soluzioni con  $P := \{x : Ax = b, x \geq 0\}$ .

Siamo ora interessati a disequazioni valide la cui parte sinistra sia  $c^T x$ , perchè ciascuna di esse comporta un **limite superiore (upper bound)** al valore ottimale  $z_P$  (ad esempio la diseq.  $c^T x \leq c_0$  valida per  $P$  implica che la soluzione ottimale sia  $z_P \leq c_0$ ).

La diseguaglianza più interessante è quella per cui l'upper bound di  $z_P$  sia il più piccolo possibile.

Formuliamo il problema di cercare la miglior diseguaglianza: le sue soluzioni sono tutte diseguaglianze del tipo  $\xi^T x \leq c_0$  e il valore di ciascuna di esse è la sua parte destra  $c_0$ , che vogliamo minimizzare.

Dal teorema di Farkas sappiamo che ciascuna di queste diseguaglianze è indiretta per qualche moltiplicatore  $u \in \mathbb{R}^m$ , allora il problema diventa il trovare i migliori moltiplicatori.

Inoltre, dato che vogliamo la parte destra minore, per ottenere la migliore diseguaglianza indiretta non rilassiamo mai la parte destra, ovvero sarebbe  $c_0 = u^T b$  invece che  $c_0 > u^T b$ .

Il nuovo problema può essere riscritto come:

$$\min u^T b$$

su tutti  $u \in \mathbb{R}^m$  tale che:

$$c^T \leq u^T A$$

Si noti che è un nuovo programma lineare. Nella sua forma canonica, solo che le variabili sono non vincolate in segno. Questo nuovo problema è detto il **duale** di quello di partenza (8). Il problema iniziale è detto **primale**.

Se denotiamo il suo valore ottimale con  $z_D$ , il problema duale (le variabili sono in vettore colonna  $u \in \mathbb{R}^m$ ), è:

$$z_D = \min \{ b^T u : A^T u \geq c \}$$


---

### PRIMALE:

$$z_P = \max \{ c^T x : Ax = b, x \geq 0 \}$$

### DUALE:

$$z_D = \min \{ b^T u : A^T u \geq c \}$$

C'è una simmetria tra il primale e il duale:

- il vettore  $c$  dei costi del primale è la parte destra del duale;
- la parte destra  $b$  del primale è il vettore dei costi del duale;
- c'è un vincolo nel duale per ogni variabile del primale;
- la matrice dei vincoli del duale è la trasposta della matrice dei vincoli del primale;
- la matrice dei vincoli del primale è la trasposta della matrice dei vinoli del duale.

Invece di trasporre la matrice  $A$  e scrivere il duale nella forma precedente, è consigliato tenere la matrice  $A$  come nel primale e scrivere le diseguaglianze del duale via pre moltiplicazione con il vettore riga delle variabili  $u^T$ . Quindi, il duale diventa:

$$z_D = \min \{ u^T b : u^T A \geq c^T \}$$

Mostriamo come ogni volta che  $z_P \neq -\infty$  (ovvero, un primale fattibile), deve essere  $z_D = z_P$  (include la situazione in cui  $z_P = z_D = +\infty$ , ovvero quando il primale è illimitato e il duale è fattibile).

Inoltre, se  $z_P = +\infty$ , ovvero, primale illimitato, allora nessuna diseguaglianza del tipo  $c^T x \leq c_0$  può essere valida, quindi non c'è soluzione al duale e  $z_D = \min \{ u^T b : u \in \emptyset \} = +\infty$  inoltre,  $z_P \in \mathbb{R}$  (dove  $z_P = c^T x^*$  e  $x^*$  è una soluzione ottimale).

Inoltre  $c^T x \leq z_P$  è una diseguaglianza valida (o  $x^*$  non sarebbe ottimale) e inoltre la diseguaglianza  $c^T x \leq c_0$  con  $c_0 < z_P$  può essere valida, dato che sarebbe violata da  $x^*$ . Inoltre la diseguaglianza migliore possibile quando  $P \neq \emptyset$  è

$$c^T x \leq z_P$$

che significa che il valore ottimale del duale  $z_D$  è di fatto  $z_P$ .

### Dualità forte e debole

**Dualità forte:** Sia  $\max \{c^T x : Ax = b, x \geq 0\}$  un programma lineare (LP) standard, e denotiamo con  $\min \{u^T b : u^T A \geq c^T\}$  il suo duale. Quindi, se il problema primale ha una soluzione ottimale, allora anche il duale ne ha una, e l'ottimo del problema primale e del problema duale sono gli stessi.

Inoltre abbiamo:

**Dualità debole:** Siano primale e duale come sopra. Allora, se  $\bar{x}$  è una soluzione fattibile del primale e sia  $\bar{u}$  qualsiasi soluzione fattibile del duale, vale:

$$c^T \bar{x} \leq \bar{u}^T b$$

dato che  $\bar{u}$  è fattibile per il duale, la diseguaglianza  $c^T x \leq \bar{u}^T b$  è valida e deve quindi essere soddisfatta da  $\bar{x}$ .

Esistono, ipoteticamente, 9 combinazioni dello stato del duale e del primale. Delle 9 combinazioni alcune sono impossibili (come il caso del primale illimitato ma il duale è infattibile). Costruiamo una tabella:

	Primale impossibile	Primale illimitato	Primale ha un ottimo
Duale impossibile	(1) YES	(2) YES	(3) NO
Duale illimitato	(4) YES	(5) NO	(6) NO
Duale ha ottimo	(7) NO	(8) NO	(9) YES

Dalla dualità forte, sappiamo che l'unica coppia possibile nell'ultima colonna e riga avviene nella cella (9). Questo esclude (3), (6), (7), (8). Per la dualità debole, se uno dei problemi è illimitato, l'altro deve essere irrealizzabile. Questo esclude la cella (5) mentre le celle (2) e (4) sono possibili. L'unica cella rimanente da riempire è la cella (1), cioè entrambi i problemi sono impossibili. Questo però è possibile, come da esempio:

**Example:** si consideri il problema lineare:

$$\max x_1 + 2x_2$$

soggetto a:

$$\begin{aligned} x_1 - x_2 &= 1 \\ x_1 - x_2 &= 2 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Il suo duale è:

$$\min u_1 + 2u_2$$

soggetto a:

$$\begin{aligned} u_1 + u_2 &\geq 1 \\ -u_1 - u_2 &\geq 2 \end{aligned}$$

Può essere visto facilmente che entrambi i problemi sono impossibili.

### Ricapitolando:

Sia  $\mathcal{P}$  il problema primale e  $\mathcal{D}$  il suo duale. Uno dei quattro casi deve essere vero:

1. Entrambi impossibili;
2.  $\mathcal{P}$  è illimitato e  $\mathcal{D}$  è impossibile;
3.  $\mathcal{D}$  è illimitato e  $\mathcal{P}$  è impossibile;
4. Entrambi hanno soluzione ottima, e i valori ottimali coincidono.

### Tipologie di duali

Abbiamo descritto la forma del duale quando il primale è in forma standard. Ma se è in forma canonica? consideriamo il problema nella sua forma canonica:

$$\max c^T x$$

tale che:

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \end{aligned}$$

Per calcolare i suoi duali, dobbiamo prima trasformarlo in forma standard con l'aggiunta di una variabile di slack  $y$ :

$$\max c^T x + \mathbf{0}^T y$$

tale che:

$$\begin{aligned} Ax + y &= b \\ xy &\geq 0 \end{aligned}$$

La matrice dei vincoli è  $(A \mid I)$ . Possiamo prendere il duale di questo problema, che è:

$$\min u^T b$$

s.t.

$$\begin{aligned} u^T A &\geq c^T \\ u^T &\geq 0^T \end{aligned}$$

Perciò il duale di un problema di massimizzazione in forma canonica è un problema di minimizzazione in forma canonica.

### Duale di un duale

Il duale di un LP è un LP: possiamo prendere i suoi duali. Per un primale in forma canonica il duale è:

$$\min u^T b$$

tale che:

$$\begin{aligned} u^T A &\geq c^T \\ u^T &\geq 0^T \end{aligned}$$

Quindi preso il suo duale, possiamo metterlo in una forma facile da maneggiare, ad esempio:

$$-\max \{-b^T u : (-A^T) u \leq -c, u \geq \mathbf{0}\}$$

Se chiamiamo  $y$  le variabili duali di questo problema, il duale è:

$$-\min \{-c^T y : -y^T A^T \geq -b, y \geq 0\}$$

tale che:

$$\max \{c^T y : Ay \leq b, y \geq 0\}$$

Ma questo è esattamente il problema prima da cui siamo partiti. Quindi abbiamo la seguente affermazione, che è vera per tutte le forme del primale:

**Lemma:** Sia  $P$  un problema di programmazione lineare e  $D$  il suo duale. Allora il duale di  $D$  è  $P$ .

### Complementary Slackness (condizioni di complementarietà)

**Teorema:** si consideri un problema primale  $\max \{c^T x : Ax \leq b, x \geq 0\}$  in forma canonica e il suo duale  $\min \{u^T b : u^T A \geq c^T, u \geq 0\}$ . Allora  $\hat{x} \in \mathbb{R}^n$  e  $\hat{u} \in \mathbb{R}^m$  sono soluzioni ottime (rispettivamente del primale e del duale) sse:

1.  $A\hat{x} \leq b, \hat{x} \geq 0$  (fattibilità di  $\hat{x}$  per primale)
2.  $\hat{u}^T A \geq c^T, \hat{u} \geq 0$  (fattibilità di  $\hat{u}$  per duale)
3.  $\hat{u}_i (b_i - A_i \hat{x}) = 0$  per ogni  $i = 1, \dots, m$
4.  $(\hat{u}^T A^j - c_j) \hat{x}_j = 0$  per ogni  $j = 1, \dots, n$

**Dimostrazione:** ( $\Rightarrow$ ) Sia  $\hat{x}$  e  $\hat{u}$  le soluzione ottimale primale e duale. Chiaramente devono essere fattibili così che (1) e (2) valgano. Dalla non negatività e fattibilità di  $\hat{x}$  e  $\hat{u}$  abbiamo:

$$c^T \hat{x} \leq (\hat{u}^T A) \hat{x} = \hat{u}^T (A\hat{x}) \leq \hat{u}^T b$$

Dalla dualità forte, otteniamo l'uguaglianza dappertutto, quindi  $\hat{u}^T b = \hat{u}^T (A\hat{x})$  e  $(\hat{u}^T A) \hat{x} = c^T \hat{x}$ . La prima condizione è equivalente a:  $\hat{u}^T (b - A\hat{x}) = 0$ , che, per via della non negatività di  $\hat{u}$ , implica (3). La seconda condizione è equivalente a  $(\hat{u}^T A - c^T) \hat{x} = 0$ , che, per via della non negatività di  $\hat{x}$ , implica (4).

( $\Leftarrow$ ) Assumendo che le condizioni 1 – 4 reggano. Da (1) e (2)  $\hat{x}$  e  $\hat{u}$  sono fattibili. Da (4) è  $(\hat{u}^T A - c^T) \hat{x} = \mathbf{0}$  tale che  $\hat{u}^T A\hat{x} = c^T \hat{x}$ . Similmente, da (3) è  $\hat{u}^T (b - A\hat{x}) = \mathbf{0}$  tale che  $\hat{u}^T b = \hat{u}^T A\hat{x}$ . Otteniamo che  $c^T \hat{x} = \hat{u}^T b$  e, dalla dualità debole, concludiamo che entrambe le soluzioni sono ottime.

(3) e (4) sono chiamate condizioni di complementarietà (complementary slackness conditions): Per le soluzioni ottime di problemi duali reciproci, ogni qual volta un vincolo di uno dei due non è soddisfatto, essendo un'equazione (ad esempio c'è uno slack positivo) la corrispondente variabile nel duale deve essere 0.

Contrariamente, ogni volta che una variabile ottimale è positiva, il vincolo corrispondente nel problema duale deve essere soddisfatto essendo un'equazione.

Queste condizioni sono anche chiamate "condizioni di ortogonalità". Infatti, dato  $\hat{x}$  e  $\hat{u}$ , si consideri il vettore  $s \in \mathbb{R}^m$  di slacks per i vincoli primali (ad esempio:  $s_i := b_i - A_i \hat{x}$  per  $i = 1, \dots, m$ ) e il vettore  $t \in \mathbb{R}^n$  di slacks per i vincoli del duale (ad esempio:  $t_j := u^T A^j - c_j$  per  $j = 1, \dots, n$ ). Quindi, le condizioni dicono che il vettore  $\hat{u}$  è ortogonale a  $s$  e  $t$  è ortogonale a  $\hat{x}$ , ad esempio:

$$\hat{u}^T s = 0 \quad \wedge \quad t^T \hat{x} = 0$$

Le condizioni di complementarietà permettono di **validare una soluzione** come ottimale senza dover risolvere un problema LP in se. Come da esempio.

**Esempio:**

$$\max x_1 - 2x_2 + 3x_3$$

soggetto a:

$$\begin{array}{lllll} x_1 & +2x_2 & -2x_3 & \leq 1 \\ 2x_1 & -x_2 & -3x_3 & \leq 4 \\ x_1 & +x_2 & +5x_3 & \leq 2 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Suppose we're told that  $\hat{x} := (\frac{9}{7}, 0, \frac{1}{7})$  is optimal and we want to verify this. For starters, we check that  $\hat{x}$  is feasible for the LP, and this is indeed the case. We then compute the dual, which is

$$\min u_1 + 4u_2 + 2u_3$$

subject to

$$\begin{array}{lllll} u_1 & +2u_2 & +u_3 & \geq 1 \\ 2u_1 & -u_2 & +u_3 & \geq -2 \\ -2u_1 & -3u_2 & +5u_3 & \geq 3 \\ & u_1, u_2, u_3 \geq 0 \end{array}$$

By complementary slackness, since  $\hat{x}_1 > 0$  and  $\hat{x}_3 > 0$ , the first and third dual constraints must be satisfied as equations by any optimal dual solution. We obtain a system of two equations in three variables valid for all optimal dual solutions  $\hat{u}$ :

$$\begin{array}{lllll} u_1 & +2u_2 & +u_3 & = 1 \\ -2u_1 & -3u_2 & +5u_3 & = 3 \end{array}$$

If we compute the slack of the primal inequalities at  $\hat{x}$  we get

$$\begin{aligned} 1 - \hat{x}_1 - 2\hat{x}_2 + 2\hat{x}_3 &= 1 - \frac{9}{7} + \frac{2}{7} = 0 \\ 4 - 2\hat{x}_1 + \hat{x}_2 + 3\hat{x}_3 &= 4 - \frac{18}{7} + \frac{3}{7} = \frac{13}{7} > 0 \\ 2 - x_1 - x_2 - 5x_3 &= 2 - \frac{9}{7} - \frac{5}{7} = 0 \end{aligned}$$

Still by complementary slackness, since the 2nd inequality of the primal has slack  $> 0$ , the component  $\hat{u}_2$  of any optimal dual solution must be 0. We can then rewrite (1) as follows:

$$\begin{array}{lllll} u_1 & +u_3 & = 1 \\ -2u_1 & +5u_3 & = 3 \end{array}$$

We solve, obtaining the 1st and 3rd component of the optimal dual solution:  $\hat{u}_1 = \frac{2}{7}$  and  $\hat{u}_3 = \frac{5}{7}$ . So, by complementary slackness, if  $\hat{x}$  is the optimal primal solution, then  $\hat{u} = (\frac{2}{7}, 0, \frac{5}{7})$  must be the optimal dual solution. Now we just check that  $\hat{u}$  is dual feasible (true) and that for each  $j$  s.t.  $\hat{u}_j > 0$  the corresponding primal constraint has no slack. To perform this check we use the primal slacks that we already computed and see that indeed the 1st and 3rd constraints have no slack. We conclude that  $\hat{x}$  was indeed optimal for the starting LP.

### Calcolare i duali in pratica

Il seguente insieme di regole ci permette di calcolare il duale  $D$  di qualsiasi primale generico  $P$  che contiene sia disequazioni che equazioni e variabili non negative e variabili non vincolate. Assumiamo che tutte le diseguaglianze di  $P$  siano rivolte nella direzione corretta (cioè, sono " $\leq$ " per problemi di massimizzazione e " $\geq$ " per problemi di minimizzazione).

1. se l'obiettivo in  $P$  è max (rispettivamente, min) allora l'obiettivo in  $D$  è min (rispettivamente, max);
2. Se  $P$  ha  $m$  vincoli e  $n$  variabili (diciamo,  $x_1, \dots, x_n$ ), allora  $D$  ha  $n$  vincoli su  $m$  variabili (ad esempio  $u_1, \dots, u_m$ );
3. la parte destra di  $D$  è il vettore dei costi di  $P$ ;
4. il vettore dei costi di  $D$  è la parte destra di  $P$ ;
5. per ogni variabile  $x_j \geq 0$  di  $P$  c'è una diseguaglianza  $\sum_{i=1}^m a_{ij}u_i \sim c_j$  in  $D$ , dove  $\sim$  ha la direzione corretta (cioè  $\leq$  o  $\geq$ ) per l'obiettivo di  $D$ ;
6. per ogni variabile  $x_j \leq 0$  di  $P$  c'è un'equazione  $\sum_{i=1}^m a_{ij}u_i = c_j$  in  $D$ ;
7. per ogni diseguaglianza  $\sum_{j=1}^n a_{ij}x_j \sim b_i$  di  $P$  c'è una variabile  $u_i \geq 0$  in  $D$ ;
8. per ogni equazione  $\sum_{j=1}^n a_{ij}x_j = b_i$  di  $P$  c'è una variabile  $u_i \leq 0$  in  $D$ .

**Example:** ADD -i no mathpix... doesn't do well

## 6 Programmazione intera

La programmazione intera è "costruita" sulla programmazione lineare, e si pone l'obiettivo di risolvere problemi di programmazione lineare in cui le variabili devono avere valori interi (che rispecchiano molti problemi della realtà – che sono NP-HARD – si congettura non ammettano algoritmi polinomiali – P vs NP).

### 6.1 Tipologie di programmi lineari interi

Problema di programmazione lineare intera (ILP) II: alcune delle variabili devono essere intere.

Se non tutte le variabili sono richieste di essere intere, parliamo di un problema di programmazione lineare intera di tipo **misto (Mixed-ILP – MILP)**.

Se tutte le variabili devono essere intere, allora II viene detto problema di programmazione intera **puro**.

Supponiamo che il problema abbia:

- $n$  variabili, partizionate come  $x = (x_1, \dots, x_k) \in \mathbb{Z}^k$  e  $y = (y_1, \dots, y_{n-k}) \in \mathbb{R}^{n-k}$ ;
- $m$  vincoli, partizionati in due matrici  $A$  e  $B$ , dove  $A$  è  $m \times k$  e  $B$  è  $m \times (n - k)$ ;
- vettore dei costi partizionato come  $c_x \in \mathbb{R}^k$  e  $c_y \in \mathbb{R}^{n-k}$ , mentre la parte destra è  $b \in \mathbb{R}^m$ .

Quindi, un generale problema di programmazione intera misto (MILP) è definito da:

$$\max c_x^T x + c_y^T y$$

soggetto a:

$$\begin{aligned} Ax + By &\leq b \\ x, y &\geq 0 \\ x &\in \mathbb{Z}^k \end{aligned}$$

Se  $k = n$  abbiamo un problema di programmazione lineare intera puro, che è definito come segue:

$$\max c^T x$$

soggetto a:

$$\begin{aligned} Ax &\leq b \\ x &\geq 0 \\ x &\in \mathbb{Z}^n \end{aligned}$$

Un caso particolare di ILP puro è quando le variabili sono vincolate a prendere valori solo in {0, 1}. In quel caso si parla di 01-ILP o BLP (Binary Linear Programming).

### Esempio di MILP:

$$\max x_1 - 2x_2 + 3x_3 - x_4 + 2x_5$$

Soggetto a:

$$\begin{array}{l} x_1 + 2x_2 - 2x_3 - 2x_4 - x_5 \leq 1 \\ 2x_1 - x_2 - 3x_3 + x_4 + 2x_5 \leq 4 \\ x_1 + x_2 + 5x_3 + x_5 \leq 2 \\ x_1, x_2, x_3 \geq 0 \in \mathbb{Z} \quad x_4, x_5 \geq 0 \end{array}$$

Abbiamo:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad y = \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 4 \\ 2 \end{pmatrix}$$

$$c_x^T = (1 \ -2 \ 3) \quad c_y^T = (-1 \ 2)$$

$$A = \begin{pmatrix} 1 & 2 & -2 \\ 2 & -1 & -3 \\ 1 & 1 & 5 \end{pmatrix} \quad B = \begin{pmatrix} -2 & -1 \\ 1 & 2 \\ 0 & 1 \end{pmatrix}$$

La programmazione intera è molto più difficile della programmazione lineare. Vogliamo sfruttare l'efficacia degli algoritmi della LP anche per l'ILP. Dobbiamo rafforzare l'integralità delle variabili in qualche modo che non siano i normali vincoli lineari. Idee come ignorare l'integralità e approssimare i valori sono troppo grezze e non producono buoni risultati.

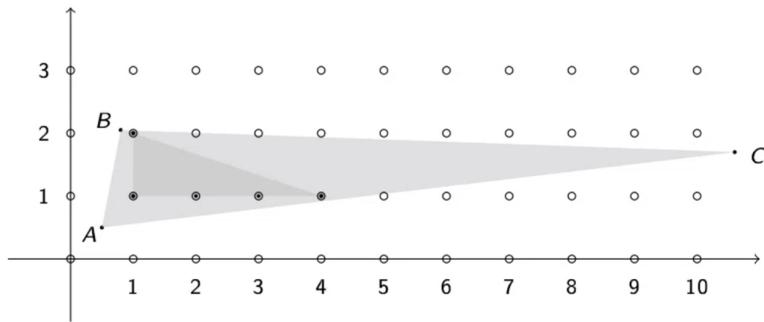


Figura: quando l'obiettivo è  $\max x_2$  si ha  $\hat{x} = B$  e l'arrotondamento è ok. Quando l'obiettivo è  $\max x_1$  si ha  $\hat{x} = C$  e l'arrotondamento non è mai ok. Quando l'obiettivo è  $\min x_1$  si ha  $\hat{x} = A$  e l'arrotondamento è ok con probabilità 0.25.

## 6.2 Geometria della programmazione lineare intera

Per un problema di programmazione intera puro, denotiamo con  $P$  l'insieme fattibile dei suoi rilassamenti di programmazione lineare (il problema lineare che ottengo trascurando i vincoli di interezza).

$$P := \{x : Ax \leq b, x \geq 0\}$$

e da  $\hat{x}$  la soluzione ottimale di tale rilassamento LP. Ovvero:

$$\hat{x} = \operatorname{argmax} \{c^T x : x \in P\}$$

In generale, vogliamo denotare con  $P^i$  un poliedro ottenuto aggiungendo qualche vincolo a  $P$ , e da  $\hat{x}^i$  la soluzione ottima del corrispondente LP. Per convenzione, settiamo:  $P^0 := P$ .

Inoltre:

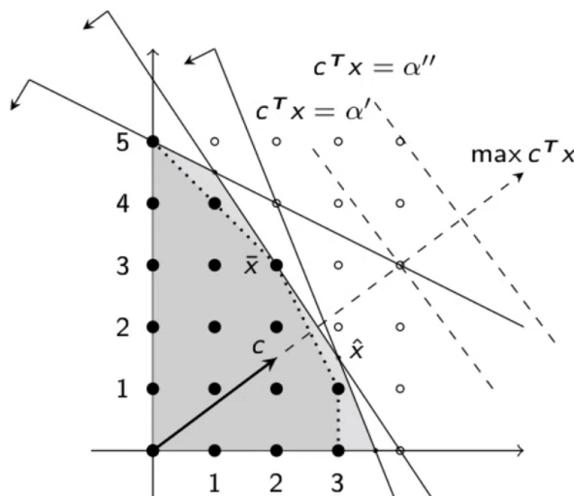
- denotiamo l'insieme di soluzioni intere con  $X = P \cap \mathbb{Z}^n$  (rispettivamente,  $X^i = P^i \cap \mathbb{Z}^n$  – punti interi all'interno di  $P^i$ ).
- denotiamo la soluzione ottima in  $X$  come  $\bar{x}$  (e, rispettivamente, lasciamo essere  $\bar{x}^i$  l'ottimo in  $X^i$  – punti interi all'interno di  $P^i$ ).
- denotiamo con  $P(X)$  il più piccolo poliedro contenente  $X$ , anche chiamato inviluppo convesso (convex hull) di  $X$  (definiamo  $P(X^i)$  analogamente).

**Esempio:**

$$\max 2x_1 + \frac{3}{2}x_2$$

soggetto a:

$$\begin{aligned} 6x_1 + 4x_2 &\leq 24 \\ x_1 + 2x_2 &\leq 10 \\ 5x_1 + 2x_2 &\leq 18 \\ x_1, x_2 &\geq 0, \text{ interi.} \end{aligned}$$



L'insieme  $X$  di punti interi fattibili (punti neri), il suo inviluppo convesso  $P(X)$  (politopo interno) e il rilassamento  $P$  (politopo esterno).

È chiaro che la soluzione ottimale è ottenuta in un vertice di  $P(X)$ . Se avessimo l'insieme di diseguaglianze che definiscono  $P(X)$  piuttosto che quelle che definiscono  $P$  potremmo risolvere un ILP come un LP. Ma l'ILP è NP-HARD, mentre l'LP è polinomiale.

Ridurre un ILP a un LP (ovvero trovare i vincoli che definiscono  $P(X)$  dati i vincoli che definiscono  $P$ ) non è polinomiale, perché la dimensione dell'LP su  $P(X)$  può essere esponenziale rispetto alla dimensione dell'LP su  $P$ .

### 6.3 Come risolvere un problema di programmazione lineare intera

**Approccio naive:** ignorando l'integrità dei vincoli e vedere se la soluzione ottimale  $\hat{x}$  sul rilassamento  $LP$  risulta un intero. In quel caso è anche l'ottimo per  $P$ . In caso contrario, è necessario cambiare le componenti frazionarie (ad esempio arrotondando ciascun componente all'intero più vicino) così da renderli interi e ottenere una soluzione  $\tilde{x} \in X$  che non dovrebbe discostare troppo dalla soluzione ottimale ed avere una buona probabilità di essere l'ottimale.

Questa strategia non è sempre corretta e possono infatti verificarsi tre casi:

1. la soluzione approssimata  $\tilde{x}$  è un ottimale;
2. la soluzione approssimata  $\tilde{x}$  è fattibile ma non ottimale;
3. la soluzione approssimata  $\tilde{x}$  è infattibile;

Il costo computazionale per risolvere un problema di programmazione lineare intera eccede la risoluzione di un programma lineare.

D'altra parte, la soluzione di un programma lineare sarebbe sufficiente se conoscessimo i vincoli che definiscono  $P(X)$ . Possiamo usare  $P^0 := P$  come prima approssimazione di  $P(X)$ , e poi ottenere una sequenza di miglioramenti  $P^0 \supset P^1 \supset P^2 \supset \dots$  fino a che si ottiene  $P(X)$ , oppure si può dimostrare che il vertice ottimale della corrente approssimazione  $P^i$  è di fatto  $\bar{x}$ .

Possiamo ottenere  $P^i$  da  $P^{i-1}$  dall'aggiunta di diseguaglianze valide per  $X$ . In particolare vogliamo aggiungere a  $P^{i-1}$  una diseguagliaza che "taglia" attraverso  $P^{i-1}$  (ovvero, è valida per  $X$  ma non per  $P^{i-1}$ ). Il metodo del piano che taglia ("cutting planes method" – che descriveremo), e tale che  $\hat{x}^i \notin P^i$  dato che aggiungiamo una diseguagliaza violata da  $\hat{x}^i$ .

Le migliori diseguaglianze in questo processo di raffinamento sarebbero le facette di  $P(X)$ , poiché ogni rappresentazione esplicita di  $P(X)$  deve includere tutte le sue facette ma non ha bisogno di includere altre diseguaglianze.

Grandi sforzi vengono fatti nell'ambito di ricerca nel tentativo di caratterizzare le facette di  $P(X)$  per problemi IP importanti come TSP, Vertex Cover, Clique, etc.

### 6.4 Problemi di programmazione lineare naturalmente interi

Un poliedro è intero se non ha vertici frazionari. Se un poliedro  $P$  di un rilassamento LP di un programma intero è intero, allora  $P = P(X)$  è l'ILP, chiamato **naturalmente intero** e può

essere risolto semplicemente trovando la soluzione ottimale del suo rilassamento. Descriviamo ora le condizioni sufficienti affinché questo accada.

**Definizione:** Una matrice  $m \times n$   $A$  è totalmente unimodulare (TUM) se il determinante di ciascuna sottomatrice quadrata  $A$  è 0, 1 o -1.

Si noti che una matrice TUM deve avere tutti i valori in  $\{0, 1, -1\}$  dato che ogni valore è una sottomatrice  $1 \times 1$ .

Esempio:

$$\begin{pmatrix} +1 & 0 & -1 & -1 \\ -1 & +1 & +1 & 0 \\ 0 & -1 & 0 & +1 \end{pmatrix}$$


---

Abbiamo, data una matrice TUM, le seguenti proprietà:

**Teorema:** Sia  $A$  una matrice TUM e  $B$  una sotto matrice quadrata non singolare di  $A$ . Allora, tutti i componenti di  $B^{-1}$  sono in  $\{0, 1, -1\}$ .

**Dimostrazione:** Sappiamo dall'algebra lineare che:

$$B^{-1} = \frac{1}{\det(B)} \begin{pmatrix} \hat{\beta}_{11} & \dots & \hat{\beta}_{1m} \\ \vdots & \ddots & \vdots \\ \hat{\beta}_{m1} & \dots & \hat{\beta}_{mm} \end{pmatrix}^T$$

dove  $\hat{\beta}_{ij} = (-1)^{i+j} \det(\bar{B}_{ij})$  e  $\bar{B}_{ij}$  è la matrice ottenuta da  $B$  rimuovendo la sua riga  $i$  e colonna  $j$ . Dato che  $B$  è non singolare,  $A$  è TUM, e  $\left| \frac{1}{\det(B)} \right| = 1$ . Inoltre, dato che  $\det(\bar{B}_{ij}) \in \{0, 1, -1\}$  per ogni  $i$  e  $j$ , concludiamo che tutti i componenti di  $B^{-1}$  sono in  $\{0, 1, -1\}$ .

---

**Teorema:** Sia  $\max \{c^\top x : Ax = b, x \geq 0, x \in \mathbb{Z}^n\}$  un problema di programmazione intera in forma standard. Allora, se la matrice  $A$  è TUM e  $b \in \mathbb{Z}^m$ , il problema è naturalmente intero.

**Dimostrazione:** Abbiamo bisogno di mostrare che tutti i vertici sono interi. Sappiamo che ogni vertice  $\hat{x}$  è una bfs, ovvero, esiste una base  $B$  t.c.  $\hat{x}_B = A_B^{-1}b, \hat{x}_N = 0$ . Ma dato che  $A_B^{-1}$  è intero e  $b$  è intero, anche  $\hat{x}_B$  è intero.

---

**Corollario:** Può essere mostrato che anche un ILP in forma canonica

$$\max \{c^\top x : Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}$$

con matrice TUM  $A$  e  $b \in \mathbb{Z}^m$  è naturalmente intero.

Si ricordi che ci sono ILP can possono essere risolti come LP.

Condizioni sufficienti perchè una matrice sia TUM:

**Teorema:** Sia  $A \in \{0, 1, -1\}^{m \times n}$  una matrice t.c.:

- in ogni colonna ci sono al massimo due elementi diversi da zero;
- le righe possono essere partizionate in due insiemi  $R_1$  e  $R_2$  t.c. per ogni colonna con due elementi diversi da zero  $d, q \in \{-1, 1\}$  allora:
  - se  $d = -q$  allora sono entrambi in  $R_1$  o entrambi in  $R_2$ ;
  - se  $d = q$  allora uno è in  $R_1$  e l'altro in  $R_2$ .

Allora  $A$  è totalmente modulare.

**Dimostrazione:** Sia  $B$  sottomatrice quadra di  $A$  di ordine  $k$ . Dimostriamo per induzione su  $k$  che  $\det(B) \in \{0, 1, -1\}$ . Se  $k = 1$  allora chiaramente è vero. Si assuma  $k > 1$  e che sia vero il teorema per  $1, \dots, k - 1$ . Allora:

- Se  $\exists$  una colonna di  $B$  che contiene solo zeri, allora  $\det(B) = 0$ ;
- Se  $\exists$  una colonna di  $B$  che contiene solo non zeri, diciamo  $b_{ij}$ , sviluppiamo con Laplace il determinante lungo quella colonna. Otteniamo  $\det(B) = (-1)^{i+j} b_{ij} \det(B_{ij})$ , dove  $B_{ij}$  è  $B$  senza la riga  $i$  e colonna  $j$ . Per introduzione,  $\det(B_{ij}) \in \{0, 1, -1\}$  e quindi  $\det(B) \in \{0, 1, -1\}$ ;
- Se tutte le colonne di  $B$  hanno due non-zeri, allora la somma delle righe di  $B$  in  $R_1$  meno la somma delle righe in  $R_2$  è zero. Così, le righe di  $B$  sono linearmente dipendenti  $\Rightarrow \det(B) = 0$ .

**Corollario:** Se  $A$  è la matrice di incidenza arco/nodo di un grafo diretto  $G = (V, E)$  o è la matrice nodo/vertice di incidenza di un grafo bipartito  $G = (V_1, V_2, E)$ , allora  $A$  è totalmente unimodulare (TUM).

**Dimostrazione:** The node-arc incidence matrix of a directed graph has  $|V|$  rows and  $|E|$  columns, each containing exactly one "+1" and one "-1". By setting  $R_1$  to be all rows and  $R_2 = \emptyset$  the conditions of Theorem 6 are satisfied. Similarly, the incidence matrix of a bipartite graph has  $|V_1| + |V_2|$  rows and  $|E|$  columns, each containing exactly two "+1" in some rows. By setting  $R_1 =$  rows associated to nodes in  $V_1$  and  $R_2 =$  rows associated to nodes in  $V_2$ , the conditions of Theorem 6 are satisfied.

Grazie alla totale unimodularità di queste matrice di incidenza, molti problemi di flusso sulle reti, cammini minimi e trasporto in grafi sono risolvibili in tempi polinomiali portandoli in LP.

## 6.5 Branch and Bound

ILP è NP-HARD. Gli algoritmi esatti per problemi ILP sono nel caso peggiore esponenziali. Un caso peggiore (ma anche migliore) esponenziale è l'enumerazione completa.

Vogliamo un metodo intelligente di eseguire una enumerazione "virtuale" completa, cioè tale che alcune soluzioni siano **enumerate in maniera implicita**.

Idealmente vorremmo poter scartare interi sottoinsiemi di soluzioni (senza dover guardare a ciascuna di esse per verificare che non possano essere l'ottimo globale).

Per identificare un sottoinsieme di soluzioni che possono essere scartate procediamo partizionando le soluzioni in insiemi disgiunti progressivamente sempre più piccoli.

Usiamo **limiti (bounds)** per stimare un **valore ottimistico** per la soluzione migliore in ciascun insieme. Se il limite non è buono a sufficienza da giustificare l'esplorazione di un sottoinsieme, scartiamo quel sottoinsieme.

### Definizione B&B:

È una tecnica per risolvere ILP basata sul concetto di "divide et impera": per risolvere un problema complesso lo scomponiamo in sottoproblemi più semplici, mediante un'analisi del caso che vincola ogni sottoproblema con alcune condizioni particolari (soddisfatte da un solo sottoinsieme dell'insieme ammissibile originale).

Due sono le idee principali su cui si basa il metodo del Branch & Bound:

**Branching:** Partizionare lo spazio delle soluzioni fattibili dividendo un problema in molteplici sottoproblemi più vincolati (idealmente più facili) tali che:

- può essere risolto in modo ottimale, o
- può essere dichiarato infattibile, o
- può essere partizionato in maniera ricorsiva (ad esempio: ramificato – branched).

La soluzione ottimale globale è quindi la miglior soluzione ottimale tra tutti i sottoproblemi.

**Bounding:** Usare qualche ragionamento matematico che porta a concludere che un certo sottoproblema non può contenere una soluzione ottima e che quindi è inutile spendere del tempo su di esso per risolverlo.

Concentriamoci su un ILP puro, cioè, chiamiamo  $X^0 := \{x : Ax \leq b, x \geq 0, x \in \mathbb{Z}^n\}$  e il problema è:

$$[\max c^T x, x \in X^0]$$

Denotiamo con  $P^0$  il poliedro associato al suo rilassamento di programmazione lineare, cioè:

$$P^0 = \{x : Ax \leq b, x \geq 0\}$$

Per semplicità, ci riferiamo a  $P^i$  sia come un poliedro che come un problema LP (cioè, trovare la soluzione ottimale,  $[\max c^T x, x \in P^i]$ ).

Similmente, ci riferiamo a  $X^i$  sia come un insieme di punti interi contenuti in  $P^i$ , sia come un problema ILP (cioè,  $\left[ \max c^T x, x \in X^i \right]$ ).

Al fine di risolvere  $X^0$ , iniziamo risolvendo  $P^0$ . Se  $\hat{x}^0$  è intero, allora è anche la soluzione ottimale di  $X^0$ . Altrimenti, sia  $j$  un componente frazionale di  $\hat{x}$ . Siano  $P^1$  e  $P^2$  definiti da:

$$P^1 = P^0 \cap \{x : x_j \leq \lfloor \hat{x}_j^0 \rfloor\} \quad \text{e} \quad P^2 = P^0 \cap \{x : x_j \geq \lceil \hat{x}_j^0 \rceil\}$$

(si noti che non ci sono punti di  $X^0$  t.c.  $\lfloor \hat{x}_j^0 \rfloor < x_j < \lceil \hat{x}_j^0 \rceil$ ). La soluzione ottimale di  $X^0$  è la migliore tra tutte le soluzioni di  $X^1$  e di  $X^2$ .

Non possiamo sapere quale di  $X^1$  e  $X^2$  contenga la soluzione ottima, quindi consideriamo entrambi. Abbiamo bisogno di risolverli ricorsivamente, iniziando dai loro rilassamenti LP  $P^1$  e  $P^2$ .

Procediamo così:

- alcune volte, la soluzione LP a un sottoproblema risulta essere intera → non c'è bisogno di ulteriore suddivisione;
- altre volte i sottoproblemi non hanno soluzioni → la suddivisione si ferma;
- un'ultima possibilità è quando concludiamo che un sottoproblema  $X^i$  non può contenere una soluzione ottima di  $X^0$  → non c'è bisogno di risolvere  $X^i$ . Facciamo ciò usando i bounds.

### Uso dei bound

Assumiamo di conoscere una soluzione fattibile  $\tilde{x} \in X^0$ , e sia  $X^i$  un sottoproblema di  $X^0$ . dato che  $X^i \subseteq P^i$ , si ha:

$$\max_{x \in X^i} c^T x \leq \max_{x \in P^i} c^T x = c^T \hat{x}^i$$

ovvero, il valore LP ottimale  $c^T \hat{x}^i$  comporta un limite superiore al valore ottimale di  $X^i$ . Se  $c^T \hat{x}^i \leq c^T \tilde{x}$ , concludiamo che ottenere il valore ottimale di  $X^i$  risulta inutile, dato che la soluzione non sarà mai migliore di quella che abbiamo già. Il sottoproblema  $X^i$  può quindi essere rimosso da ogni futura considerazione (ovvero "ucciso" – "killed" o "phantomed").

**Branch:** La parola "branch" (ramo) indica il processo di scomposizione di un problema in sottoproblemi. Associamo a ciascun problema un nodo di un albero (albero di ricerca – "search-tree"), e gli archi da un nodo ai suoi figli indicano l'aggiunta di un vincolo che definisce i sottoproblemi di un problema.

**Bound:** La parola "bound" (legame) indica che a ciascun nodo dell'albero di ricerca calcoliamo un upper bound per il valore ottimale del sottoproblema. Ogni qual volta il bound non è migliore del valore di una soluzione ottimale già conosciuta (valore che prende il nome di **incombente**) allora i sottoproblemi vengono scartati senza bisogno di risolverli o dividerli ulteriormente.

## 6.6 Branch & Bound - pseudocodice

Una versione dello speudocodice del branch and bound:

$\tilde{x}$  the best integer solution found so far (at termination is the global optimal solution)  
 $\tilde{v}$  the value of such solution (i.e., the incumbent)  
 $\mathcal{L}$  the *list of open problems*, i.e., the set of the subproblems which have still to be solved.

---

```

0 |  $\tilde{x} \leftarrow \text{undefined. } \tilde{v} \leftarrow -\infty. \mathcal{L} \leftarrow \{P^0\}$ 
1 | while  $\mathcal{L} \neq \emptyset$  do
2 |   Pick a problem  $P^i \in \mathcal{L}$ , and set  $\mathcal{L} \leftarrow \mathcal{L} \setminus P^i$ 
3 |   Solve the linear programming problem  $P^i$ 
4 |   if  $P^i$  is feasible then begin
5 |     Let  $\hat{x}^i$  be the optimal solution of  $P^i$ 
6 |     if  $c^T \hat{x}^i > \tilde{v}$  then begin
7 |       if  $\hat{x}^i$  is integral then  $\tilde{x} \leftarrow \hat{x}^i; \tilde{v} \leftarrow c^T \hat{x}^i$ 
8 |       else begin
9 |         Let  $k$  be a frational component of  $\hat{x}^i$ .
10 |        Generate two problems sons of  $P^i$ , namely
11 |         $P^{i_1} = P^i \cap \{x : x_k \leq \lfloor \hat{x}_k^i \rfloor\}$  e  $P^{i_2} = P^i \cap \{x : x_k \geq \lceil \hat{x}_k^i \rceil\}$ 
12 |        Add them to the list of open problems:  $\mathcal{L} \leftarrow \mathcal{L} \cup \{P^{i_1}, P^{i_2}\}$ 
13 |     end else
14 |   end if
15 | end if
16 | end while

```

---

La scelta del prossimo problema (step 2) è arbitraria e dipende dalla struttura dati scelta per la lista, le soluzioni più comuni sono:

1. **LIFO:** il prossimo sottoproblema è quello creato più recentemente.  $\mathcal{L}$  è uno stack e l'albero di ricerca è visitato in "depth-first".
2. **FIFO:** il prossimo sottoproblema è quello che è nella lista da più tempo.  $\mathcal{L}$  è una cosa, e l'albero di ricerca è visitato in "breadth-first".
3. **Best First:** associamo a ciascun problema nella lista il bound ottenuto nello step 6 (il bound del padre) e il prossimo sottoproblema è quello con il bound più alto (forse ha probabilità più alta di ottenere la soluzione migliore rispetto agli altri nodi).

Quando nello step 9 la soluzione ottimale  $\hat{x}^i$  ha più di una componente frazionale, scegliamo la componente su cui separare (branching). Possiamo scegliere il componente su cui eseguire lo split. La regola generale di default è di dividere sul componente più frazionale, ovvero, quella per cui  $\hat{x}_k^i - \lfloor \hat{x}_k^i \rfloor$  è più vicina a 1/2. (esempio: 9.36 è più frazionale di 20.13 – 0.36 più vicino a 0.5 di 0.16)

**Esempio Branch & Bound:** Una copisteria prepara e vende appunti di due corsi, stampando le pagine e rilegandole. La copisteria possiede 240 g di colla, 1000 fogli di carta e 3 ore di tempo a disposizione.

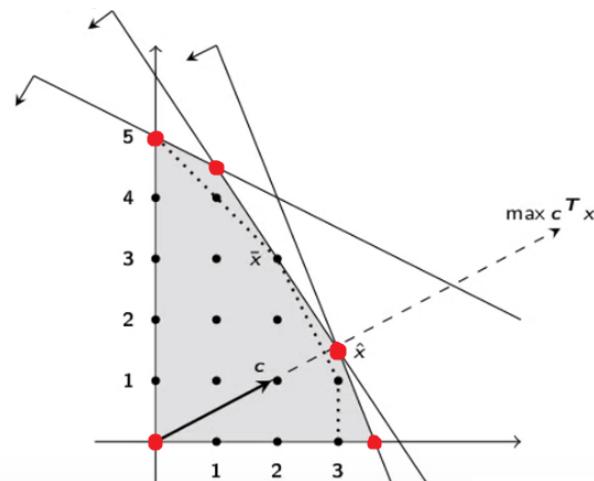
I primi appunti hanno 100 pagine e richiedono 60 g di colla e 50 minuti di preparazione. I secondi appunti hanno 200 pagine e richiedono 40 g di colla e 20 minuti di preparazione. Ciascuna rilegatura per il primo corso porta a un profitto di \$1.9, mentre quelle dei secondi appunti di \$1 ciascuno. L'obiettivo è di massimizzare il profitto totale.

Denotiamo con  $x_1$  e  $x_2$  il numero delle rilegature dei due corsi. Il problema diventa:

$$\max[1.9x_1 + 1x_2]$$

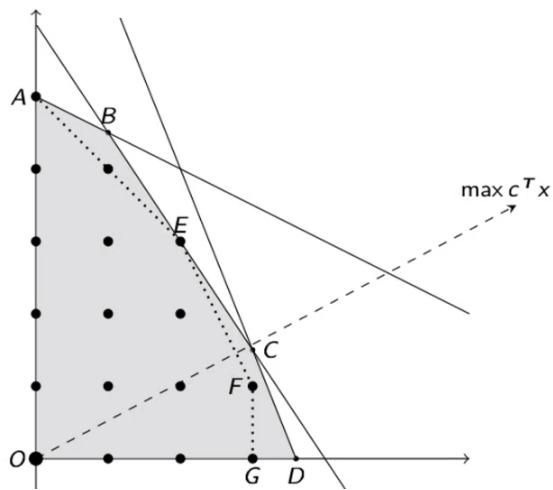
t.c.

$$\begin{aligned} 60x_1 + 40x_2 &\leq 240 \\ 100x_1 + 200x_2 &\leq 1000 \\ 50x_1 + 20x_2 &\leq 180 \end{aligned}$$

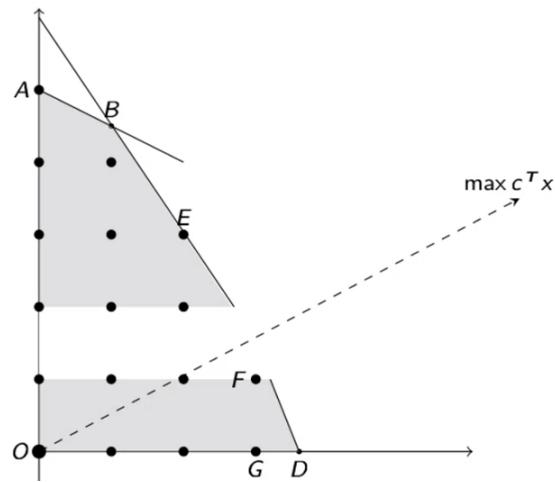


I vertici di  $P^0$  sono  $O := (0,0)$ ,  $A := (0,5)$ ,  $B := (1, \frac{9}{2})$ ,  $C := (3, \frac{3}{2})$  e  $D = (\frac{18}{5}, 0)$ . I vertici di  $P(X)$  sono  $O, A, E := (2, 3), F := (3, 1)$  e  $G := (3, 0)$ .

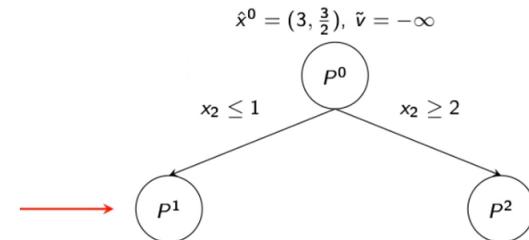
Calcolando la funzione obiettivo su questi punti mostra che  $C$ , per cui  $c^T C = 7.20$  è la sol. ottimale di  $P$ , mentre  $E$ , per cui  $c^T E = 6.80$  è l'ottimale intero per cui stiamo guardando.

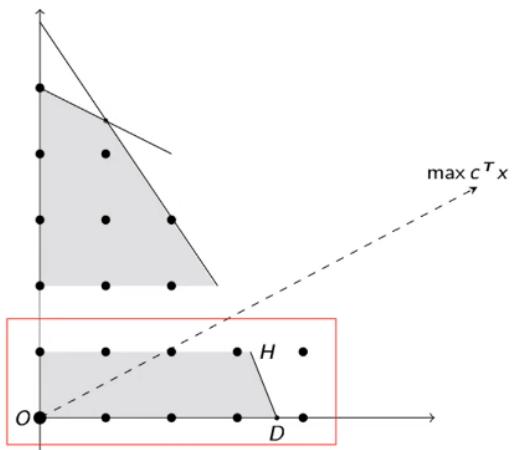


Applichiamo il metodo Branch and Bound, con una visita depth-first. Iniziamo risolvendo  $P^0$  che porta a  $\hat{x}^0 = C = \left(3, \frac{3}{2}\right)$ . Essendo  $\hat{x}_2^0$  frazionale, creiamo due nuovi problemi  $P^1 = P^0 \cap \{x : x_2 \leq 1\}$  e  $P^2 = P^0 \cap \{x : x_2 \geq 2\}$ .

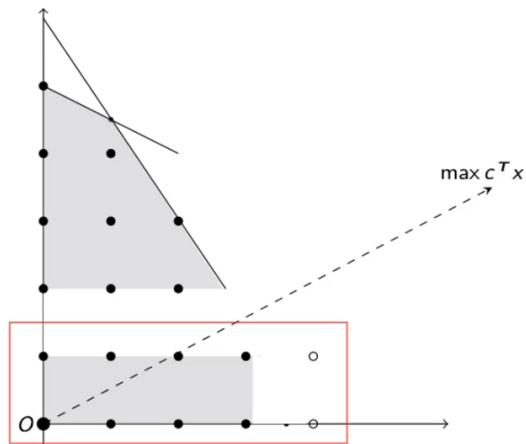


Il relativo albero di ricerca:

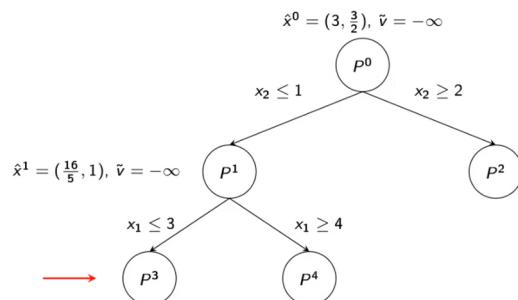


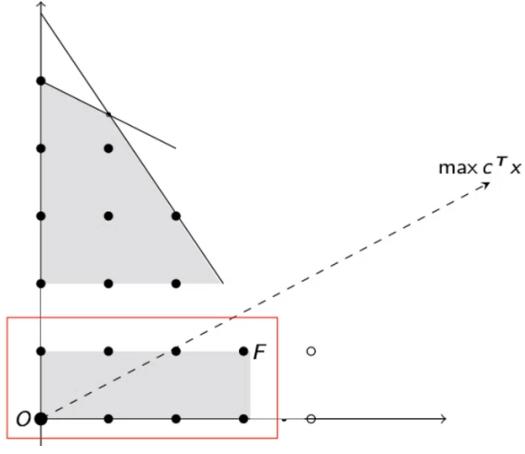


Trasformiamo la soluzione di  $P^1$ . Appaiono nuovi vertici  $H = \left(\frac{16}{5}, 1\right)$ , con  $c^T H = 7.08$ . Il vertice  $\hat{x}^1 = H$  è ottimale per  $P^1$ . Essendo  $\frac{16}{5}$  frazionale, aggiugniamo a  $P^1$  in un caso il vincolo  $x_1 \leq 3$ , ottenendo  $P^3$ , e nell'altro il vincolo  $x_1 \geq 4$ , ottenendo  $P^4$ .

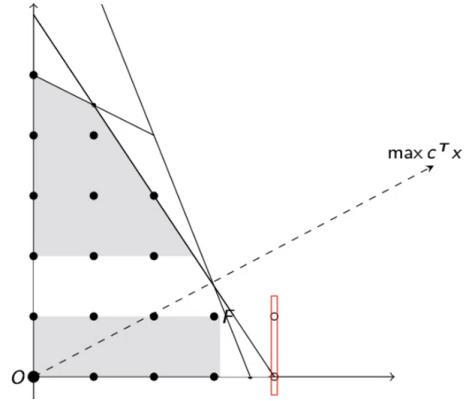
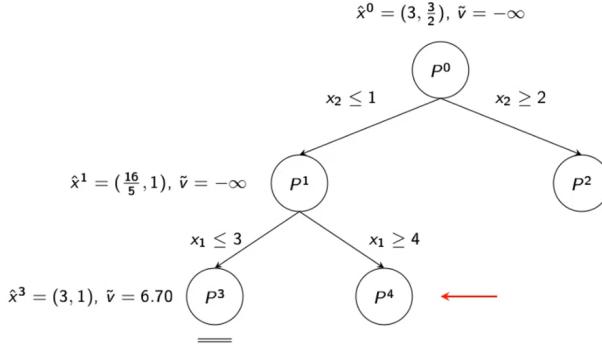


La nuova situazione dell'albero di ricerca:



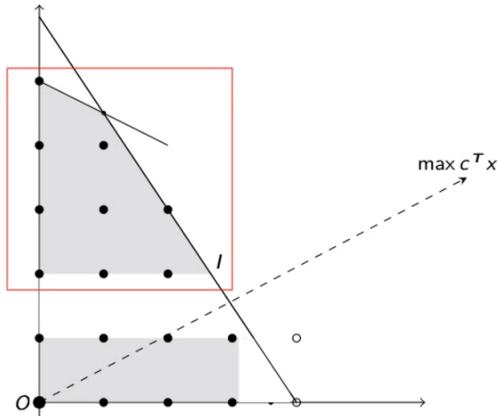
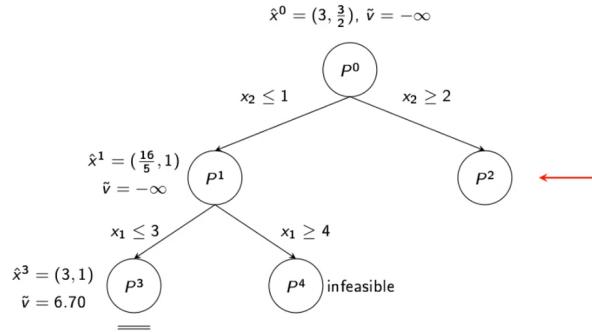


È il turno di  $P^3$ . Tutti i vertici di  $P^3$  sono interi! Per un poliedro limitato di partenza  $P$  questo deve accadere, prima o poi. La soluzione ottimale di  $P^3$  è  $\hat{x}^3 = F$ , con valore  $c^T F = 6.70$ . Essendo  $\hat{x}^3$  intero, non c'è branching da  $P^3$ . Inoltre, abbiamo trovato la soluzione del primo branch, la salviamo in  $\tilde{x} \leftarrow (3, 1)$  e assegnamo  $\tilde{v} \leftarrow 6.70$ .

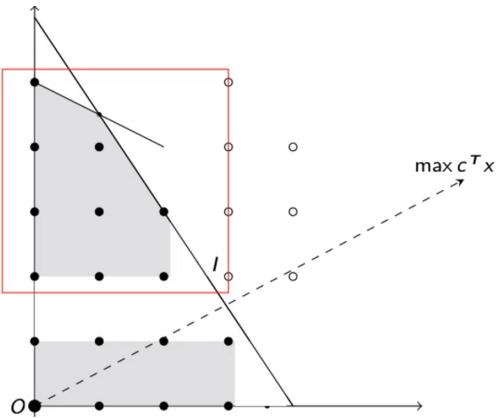


È ora il turno di  $P^4$ , che è vuoto (cioè,  $P^4$  è infattibile), quindi viene scartato (che conclude l'elaborazione di  $P^4$  e anche di  $P^1$ ). Ci muoviamo ora su  $P^2$ .

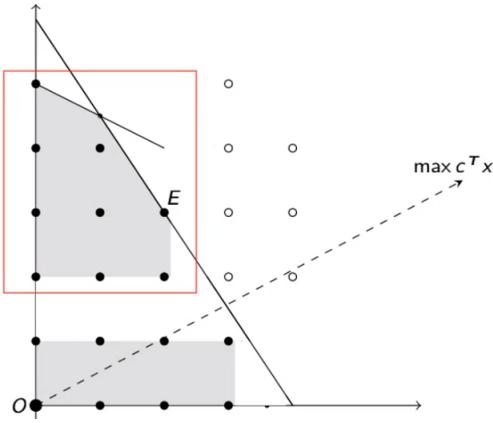
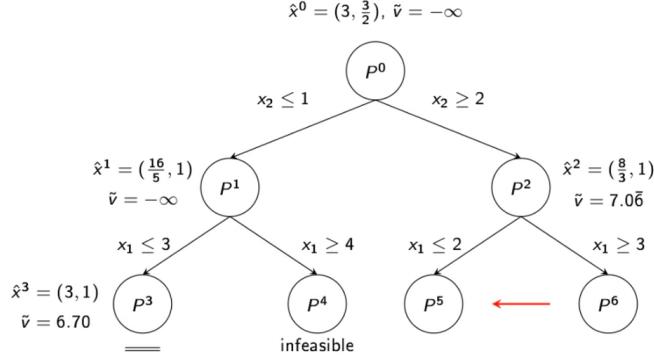
La nuova situazione dell'albero:



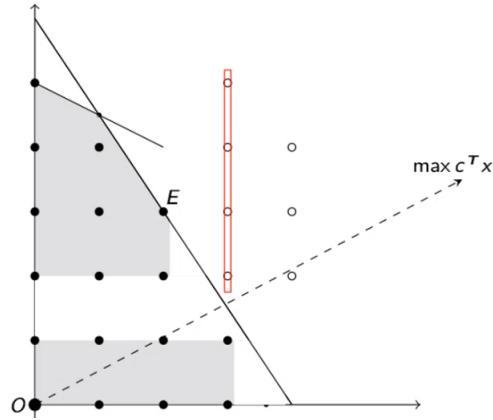
In  $P^2$  appare un nuovo vertice  $I = \left(\frac{8}{3}, 2\right)$ , con valore  $c^T I = 7.0\bar{6}$ . Questa è la soluzione ottimale di  $P^2$ , ovvero,  $\hat{x}^2 = 1$ . Essendo  $7.0\bar{6} = c^T \hat{x}^2 > c^T \tilde{x} = 6.70$ , non c'è phatoming e dobbiamo dividere. Da  $P^2$  generiamo due nuovi problemi,  $P^5 = P^2 \cap \{x : x_1 \leq 2\}$  e  $P^6 = P^2 \cap \{x : x_1 \geq 3\}$ .



La nuova situazione dell'albero:

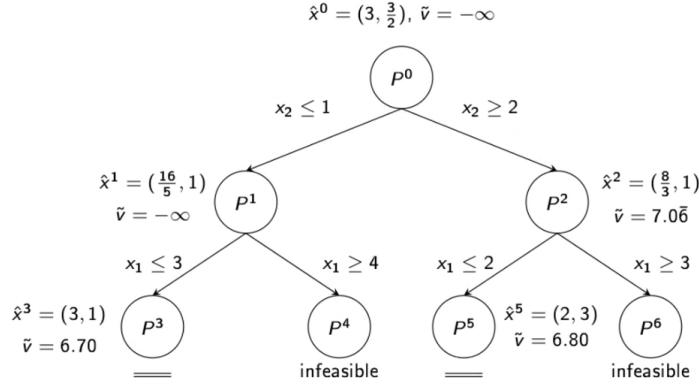


Consideriamo ora  $P^5$  che ha un nuovo vertice intero  $E = (2, 3)$ , come soluzione ottimale. Essendo  $\hat{x}^5$  un intero, non c'è branching. Confrontiamo  $\hat{x}^5$  a  $\tilde{x}$ . Dato  $6.80 = c^T \hat{x}^5 > c^T \tilde{x} = 6.70$  aggiorniamo la soluzione ottima corrente, impostando  $\tilde{x} \leftarrow (2, 3)$  e  $\tilde{v} \leftarrow 6.80$



Arriviamo infine a  $P^6$ , che risulta infattibile. Dato che la lista di problemi aperti è ora vuota, la ricerca si ferma con ottimo globale pari a  $E = (2, 3)$ .

L'albero finale risulta essere:



## 6.7 Metodo dei piani di taglio

Alternativa al Branch & Bound. Consideriamo un problema ILP in forma standard.

$$\begin{aligned} \max c^T x \\ Ax = b \\ x \geq 0, x \in \mathbb{Z}^n \end{aligned}$$

Chiamiamo  $P := \{x : Ax = b, x \geq 0\}$  il rilassamento di LP e sia  $X := P \cap \mathbb{Z}^n$ , e denotiamo con  $P(X) = \text{conv}(X)$ .

Risolviamo il rilassamento LP e sia  $\hat{x}$  la sua soluzione ottimale. Se  $\hat{x}$  è intero, allora è anche ottimale per il problema ILP originale. Altrimenti, ci dev'essere una diseguaglianza valida per  $P(X)$  violata da  $\hat{x}$ , cioè, una diseguaglianza:

$$\alpha_1 x_1 + \cdots + \alpha_n x_n \leq \beta$$

t.c.  $\alpha_1 \bar{x}_1 + \cdots + \alpha_n \bar{x}_n \leq \beta$  per ogni  $\bar{x} \in P(X)$ , ma  $\alpha_1 \hat{x}_1 + \cdots + \alpha_n \hat{x}_n > \beta$  (per contraddizione: se tutte le diseguaglianze valide per  $X$  erano soddisfatte da  $\hat{x}$  avrebbe implicato che  $\hat{x} \in P(X)$ ). In particolare  $\hat{x}$  sarebbe una combinazione convessa di punti di  $X \subseteq P$  e quindi non può essere vertice di  $P$ ).

Se aggiungiamo la diseguaglianza  $\alpha^T x \leq \beta$  al sistema definente  $P$ , otteniamo un nuovo rilassamento (cioè, un poliedro inferiore)  $P^1$  tale che  $P^1 \supseteq P(X)$  e  $\hat{x} \notin P^1$ , cioè:

$$P \supset P^1 \supseteq P(X)$$

L'aggiunta di qualsiasi diseguaglianza valida per  $P(X)$  ma violata da  $\hat{x}$ , si dice che  $\hat{x}$  è "tagliata fuori" (cut-off), e quindi queste diseguaglianze sono chiamate "tagli".

Quando i tagli sono stati aggiunti al modello, risolviamo il problema LP su  $P^1$ , ottenendo una soluzione ottimale LP  $\hat{x}^1$ . Se  $\hat{x}^1$  è intero allora è ottimale per l'ILP originale. Altrimenti dovremmo trovare una (o più) diseguaglianze:

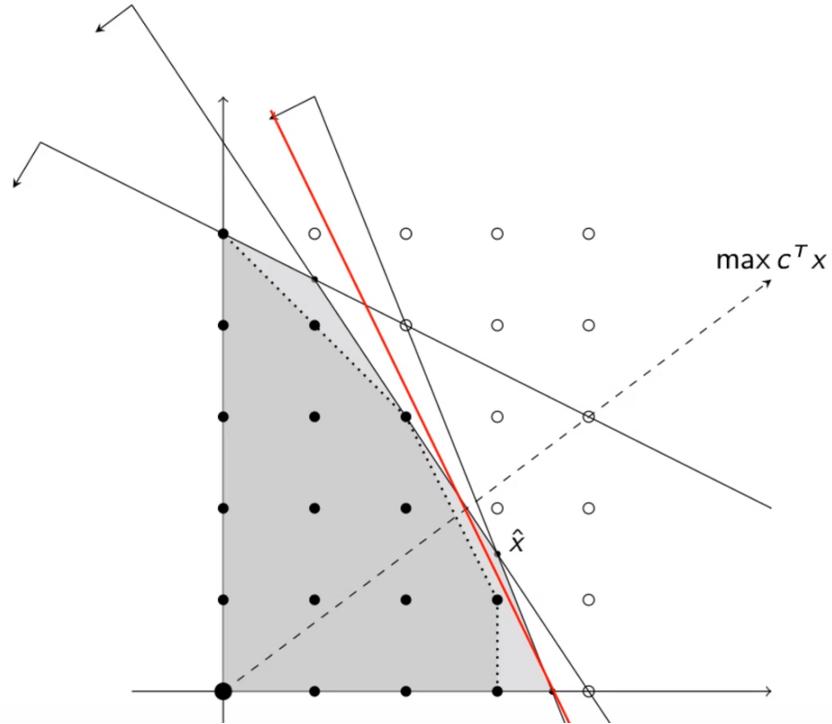
$$\alpha_1 x_1 + \cdots + \alpha_n x_n \leq \beta$$

valide per  $P(X)$  ma violate da  $\hat{x}^1$ . Lo aggiungiamo a  $P^1$  e otteniamo  $P^2$ . Possiamo ripetere questo processo ottenendo  $P^3, \dots$  t.c.

$$P \supset P^1 \supset P^2 \supset \dots \supseteq P(X)$$

finché eventualmente l'ultima soluzione LP  $\hat{x}^k$  è intera e d'ora in poi è la soluzione ottimale ILP.

Trovare una diseguaglianza violata per il vertice frazionario corrente  $\hat{x}^i$  è chiamato problema di separazione (separation problem).



### Diseguaglianze di Chvatal

Vedremo come tutte le diseguaglianze valide per  $P(X)$  siano nella forma:

$$\lfloor uA \rfloor x_\Delta \leq \lfloor ub \rfloor$$

dove  $u = (u_1, \dots, u_m)$  sono moltiplicatori non negativi per le righe del sistema.

Queste sono dette diseguaglianze di Chvatal per il problema originale. Se aggiungiamo tutti i tagli di Chvatal a  $P$  e chiamiamo  $C(P)$  l'insieme ottenuto, può essere mostrato (dimostrazione molto difficile!) che  $C(P)$  è di fatto un poliedro (chiamato prima chiusa di Chvatal di  $P$ ). Sia  $Q := C(P)$ . Possiamo allora aggiungere tutti i tagli di Chvatal derivabili da  $Q$  e ottenere  $S = C(Q)$ , chiamata seconda chiusura di Chvatal di  $P$  e così via.

Procedendo in questo modo, eventualmente otteniamo esattamente  $P(X)$ .

Perciò, l'aggiunta di un numero finito di tagli di Chvatal è sufficiente ad ottenere  $P(X)$  ma non sappiamo in anticipo quali tagli, e possiamo finire ad aggiungere più di quello che è necessario.

Sia assuma che l'ILP sia in forma standard. Sia  $u \in \mathbb{R}^m$ . Se  $Ax = b$  allora è anche  $u^T Ax = u^T b$ , cioè:

$$\sum_{j=1}^n \left( \sum_{i=1}^m u_i a_{ij} \right) x_j = \sum_{i=1}^m u_i b_i$$

Se riduciamo i coefficienti sulla sinistra, data la non negatività di  $x$  otteniamo:

$$\sum_{j=1}^n \left\lfloor \sum_{i=1}^m u_i a_{ij} \right\rfloor x_j \leq \sum_{i=1}^m u_i b_i$$

Quando  $x \in X$ , la parte sinistra della diseguaglianza deve essere un numero intero, e quindi non possiamo approssimare la parte destra a un valore intero, quindi ottenendo una diseguaglianza valida per  $X$ :

$$\sum_{j=1}^n \left\lfloor \sum_{i=1}^m u_i a_{ij} \right\rfloor x_j \leq \left\lfloor \sum_{i=1}^m u_i b_i \right\rfloor$$

Chiamiamo questa diseguaglianza una diseguaglianza di rango 1 di Chvatal, generata da  $u$ .

### Esempio:

Siano  $X$  le soluzioni al sistema:

$$2x_1 + 5x_2 + x_3 = 30$$

$$4x_1 - 3x_2 + x_4 = 6$$

Sia  $u = \left(\frac{5}{6}, \frac{11}{12}\right)$ . Moltiplicando le due equazioni dai coefficienti  $u$  e aggiungendoli otteniamo:

$$\begin{aligned} \left(\frac{5}{6} \cdot 2 + \frac{11}{12} \cdot 4\right) x_1 + \left(\frac{5}{6} \cdot 5 - \frac{11}{12} \cdot 3\right) x_2 + \frac{5}{6} x_3 + \frac{11}{12} x_4 &= \left(\frac{5}{6} \cdot 30 + \frac{11}{12} \cdot 6\right) \\ \text{i.e. } \frac{16}{3} x_1 + \frac{17}{12} x_2 + \frac{5}{6} x_3 + \frac{11}{12} x_4 &= 30.5 \end{aligned}$$

Arrotondando a sinistra, otteniamo:

$$5x_1 + x_2 \leq 30.5$$

Finalmente, arrotondando a destra otteniamo la diseguaglianza di Chvatal:

$$5x_1 + x_2 \leq 30$$

## Tagli di Gomory

Le diseguaglianze di Chvatal sono sempre valide, ma non tutte tagliano un dato ottimo LP  $\hat{x}$ .

Un metodo per ottenere diseguaglianze di Chvatal che sono anche tagli è l'utilizzo dei tagli di Gomory.

Sia  $B$  la base ottimale associata all'ottimo LP  $\hat{x}$ , e siano  $N$  le variabili non basiche. Sia  $\bar{A} = A_B^{-1}A$  e  $\bar{b} = A_B^{-1}\bar{b} = \hat{x}_B$ . Si assuma che le colonne del sistema siano state permutate così da  $\bar{A} = (I \mid A_B^{-1}A_N)$ . Sia  $\hat{x}_1$  un componente frazionale di  $\hat{x}$  e si consideri la  $I$ -esima riga del sistema  $\bar{A}\bar{x} = \bar{b}$ :

$$x_l + \sum_{j \in N} \bar{a}_{lj} x_j = \bar{b}_I$$

La parte destra  $\bar{b}_I$  (che è  $= \hat{x}_l$ ) è frazionale. Allora possiamo prima arrotondare la parte di sinistra, ottenendo:

$$x_I + \sum_{j \in N} \lfloor \bar{a}_{lj} \rfloor x_j \leq \bar{b}_I$$

e la parte destra, così da ottenere la seguente diseguagliaza valida:

$$x_1 + \sum_{j \in N} \lfloor \bar{a}_{lj} \rfloor x_j \leq \lfloor \bar{b}_I \rfloor$$

La diseguagliaza:

$$x_l + \sum_{j \in N} \lfloor \bar{a}_{lj} \rfloor x_j \leq \lfloor \bar{b}_I \rfloor$$

è detta taglio di Gomory nella sua forma intera. Non è difficile riconoscere che (1) non è altro che una diseguagliaza di Chvatal generata dal particolare  $u$  dato dalla  $I$ -esima riga di  $A_B^{-1}$ .

Si noti che:

1. è una diseguagliaza valida per  $X$  (una diseguagliaza di Chvatal);
2. o è violata da  $\hat{x}$ . Infatti, si ha che  $\hat{x}_1 = \bar{b}_I$  e  $\hat{x}_j = 0$  per  $j \in N$ . Rimpiazzando in (1), otteniamo  $\bar{b}_I \leq \lfloor \bar{b}_I \rfloor$ , che è chiaramente falso dato che  $\bar{b}_I \notin \mathbb{Z}$ .

Perciò,  $\hat{x}$  viene tagliata da (1), che può essere aggiunta ai vincoli di  $P$  così da ottenere  $P'$ .

## Tagli di Gomory frazionali

Si denoti la parte frazionale di un reale  $z$  con  $[z]_f = z - \lfloor z \rfloor$  (chiaramente è  $0 \leq [z]_f < 1$  per ogni  $z \in \mathbb{R}$ ). La diseguagliaza  $x_I + \sum_{j \in N} \bar{a}_{lj} x_j = \bar{b}_I$  diventa

$$x_I + \sum_{j \in N} \left( \lfloor \bar{a}_{lj} \rfloor + [\bar{a}_{lj}]_f \right) x_j = \lfloor \bar{b}_I \rfloor + [\bar{b}_I]_f$$

che può essere riscritta come:

$$\sum_{j \in N} [\bar{a}_{lj}]_f x_j = [\bar{b}_I]_f + \left( \lfloor \bar{b}_I \rfloor - \sum_{j \in N} \lfloor \bar{a}_{lj} \rfloor x_j - x_l \right) \quad (2)$$

Il termine nelle parentesi nella parte destra deve essere non negativo. Inoltre, dalla forma intera dei tagli di Gomory abbiamo che  $\lfloor \bar{b}_1 \rfloor \geq x_1 + \sum_{j \in N} \lfloor \bar{a}_{lj} \rfloor x_j$ , cioè,

$$\lfloor \bar{b}_I \rfloor - \sum_{j \in N} \lfloor \bar{a}_{lj} \rfloor x_j - x_I \geq 0 \quad (3)$$

Usando (3) in (2) deriviamo la seguente diseguaglianza:

$$\sum_{j \in N} [\bar{a}_{lj}]_f x_j \geq [\bar{b}_l]_f$$

La diseguaglianza:

$$\sum_{j \in N} [\bar{a}_{lj}]_f x_j \geq [\bar{b}_l]_f \quad (4)$$

Questa è la forma frazionale dei tagli di Gomory. Si noti come la soluzione ottimale basica corrente non soddisfa (4), dato che  $\hat{x}_j = 0$  per ogni  $j \in N$ , ma  $[\bar{b}_l]_f$  è una frazione positiva.

### Metodo dei piani di tagli (di Gomory)

Iniziamo risolvendo  $P^i$  per  $i := 0$ , e poi iterando la seguente procedura:

1. while  $\hat{x}^i$  is not integer do
2. Find a Gomory cut corresponding to a fractional component of  $\hat{x}^i$
3. Add the cut to the constraints of  $P^i$  thus obtaining  $P^{i+1}$
4. Set  $i \leftarrow i + 1$
5. Solve  $P^i$  thus obtaining  $\hat{x}^i$
6. end while

Si noti che un taglio è aggiunto a un modello in forma standard, per mantenere il modello in forma standard dobbiamo trasformare il taglio in un'equazione, introducendo o una variabile di slack o una variabile di surplus.

**Esempio:** Gestiamo una piccola compagnia che produce prodotti  $A$  e  $B$ . Ciascuna unità di  $A$  ha bisogno di 2 ore di lavoro mentre ciascuna unità di  $B$  necessita di 5 ore. Abbiamo 30 ore per la produzione. Ciascuna unità di  $A$  crea 4 unità di un sottoprodotto  $C$  e ogni unità di  $B$  consuma 3 unità di  $C$  per essere prodotto. Se  $C$  è prodotto in eccesso, possiamo immagazzinare le unità in un magazzino, che ha una capacità di al massimo 6 unità. Il profitto dalla vendita delle unità di  $A$  o  $B$  è lo stesso. Vogliamo decidere la produzione al fine di massimizzare il profitto.

Si denoti con  $x_1$  e  $x_2$  il numero di unità del prodotto  $A$  e del prodotto  $B$  che verranno prodotti, e si denoti con  $x_3$  il profitto. Il modello ILP è:

$$\max x_3$$

soggetto a:

$$\begin{array}{lll}
2x_1 & +5x_2 & \leq 30 \\
4x_1 & -3x_2 & \leq 6 \\
x_1 & +x_2 & -x_3 = 0 \\
x_1, x_2 & \geq 0, \text{ integer.}
\end{array}$$

Trasformando il modello in forma standard e rilassando l'integralità otteniamo il seguente problema LP:

$$\max x_3$$

soggetto a:

$$\begin{array}{lll}
2x_1 & +5x_2 & +x_4 = 30 \\
4x_1 & -3x_2 & +x_5 = 6 \\
x_1 & +x_2 & -x_3 = 0 \\
x_1, \dots, x_5 & \geq 0
\end{array}$$

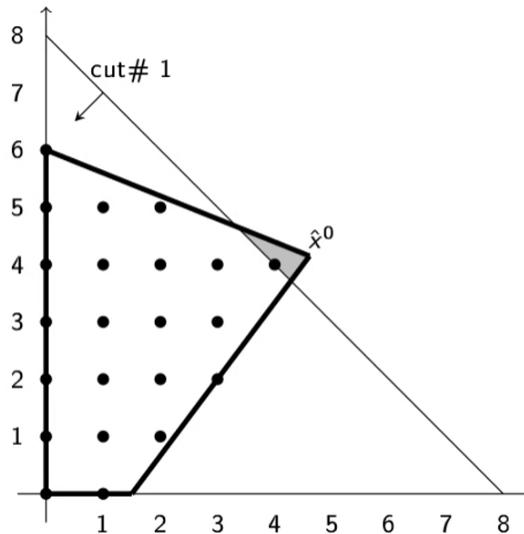
La soluzione ottimale  $\hat{x}^0 = (\frac{60}{13}, \frac{54}{13}, \frac{114}{13}, 0, 0)$  è frazionale, generiamo così un taglio di Gomory. Ci sono tre possibili tagli di Gomory. Tipicamente generiamo il taglio per cui la parte frazionale destra è la più grande.

In questo caso, corrispondere alla variabile frazionale  $x_3$ . Le variabili basiche sono  $x_1, x_2, x_3$  e le non basiche sono  $x_4, x_5$ . Se moltiplichiamo la matrice dei vincoli per  $A_B^{-1}$  la terza riga diventa:

$$x_3 + \frac{7}{26}x_4 + \frac{3}{26}x_5 = \frac{114}{13}$$

da cui deriviamo il taglio:

$$\frac{7}{26}x_4 + \frac{3}{26}x_5 \geq \frac{10}{13}$$



Risulta utile guardare al taglio graficamente, nello spazio di  $x_1$  e  $x_2$ . Dato che  $x_4 = 30 - 2x_1 - 5x_2$  e  $x_5 = 6 - 4x_1 + 3x_2$ , il taglio può essere espresso nello spazio di  $(x_1, x_2)$  –

come:

$$7(30 - 2x_1 - 5x_2) + 3(6 - 4x_1 + 3x_2) \geq 20$$

cioè:

$$x_1 + x_2 \leq 8$$

Aggiungiamo il taglio e trasformiamo il problema in forma standard con l'introduzione di una variabile di slack  $x_6$ . Il modello diventa:

$$\max x_3$$

soggetto a:

$$\begin{array}{rclcl} 2x_1 & +5x_2 & +x_4 & = 30 \\ 4x_1 & -3x_2 & +x_5 & = 6 \\ x_1 & +x_2 & -x_3 & & = 0 \\ & & \frac{7}{26}x_4 & +\frac{3}{26}x_5 & -x_6 = \frac{10}{13} \\ x_1, \dots, x_6 & \geq 0 & & & \end{array}$$

La soluzione ottimale è  $\hat{x}^1 = (\frac{30}{7}, \frac{26}{7}, 8, \frac{20}{7}, 0, 0)$  con variabili basiche  $x_1, x_2, x_3, x_4$ . Generiamo un nuovo taglio di Gomory corrispondente al componente  $\hat{x}_4^1$ . Quando moltiplichiamo il sistema per  $A_B^{-1}$ , il vincolo per le variabili non basiche è:

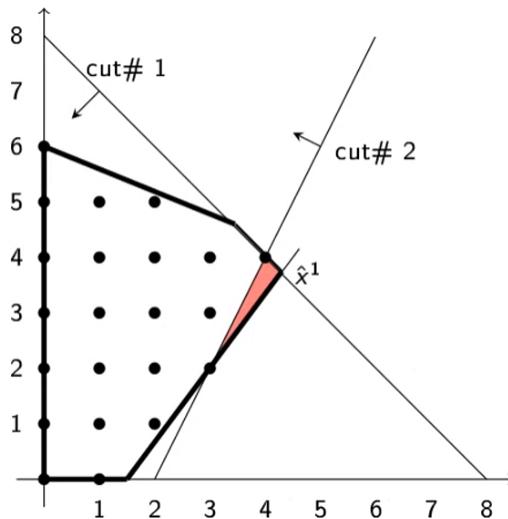
$$x_4 + \frac{3}{7}x_5 - \frac{26}{7}x_6 = \frac{20}{7}$$

dato che  $-26/7 = -4 + 2/7$  e  $20/7 = 2 + 6/7$ , otteniamo il seguente nuovo taglio di Gomory:

$$\frac{3}{7}x_5 + \frac{2}{7}x_6 \geq \frac{6}{7}$$

che nello spazio di  $(x_1, x_2)$  corrisponde al vincolo:

$$2x_1 - x_2 \leq 4$$



Aggiungiamo il taglio, insieme al nuovo surplus  $x_7$  al vincolo che diventa:

$$\begin{array}{rclcl}
 2x_1 & +5x_2 & +x_4 & & = 30 \\
 4x_1 & -3x_2 & & +x_5 & = 6 \\
 x_1 & +x_2 & -x_3 & & = 0 \\
 & \frac{7}{26}x_4 + \frac{3}{26}x_5 & -x_6 & = \frac{10}{13} \\
 & \frac{3}{7}x_5 & +\frac{2}{7}x_6 & -x_7 & = \frac{6}{7}
 \end{array}$$

La nuova soluzione ottimale è  $\hat{x}^2 = (4, 4, 8, 2, 2, 0, 0)$  corrispondente al vertice (4,4) nello spazio  $(x_1, x_2)$ . Dato che la soluzione è intera, è la soluzione ottimale al problema originale ILP, e il metodo si ferma.

## 7 Problemi di Ottimizzazione combinatoria e problemi sui grafi

I grafi sono molto comuni nella definizione e risoluzione di problemi di ricerca operativa.

Un importante numero di problemi di ottimizzazione richiedono di trovare il sottoinsieme ottimo  $X$  di vertici e/o nodi in un dato grafo pesato.

Gli elementi di  $X$  definiscono un particolare sottografo, che può essere un cammino ottimale, una clique, uno spanning tree, etc.

Durante la fase di modellazione di un problema di ottimizzazione, potremmo individuare strutture a grafo, e quindi la soluzione ottimale può essere ridotta alla risoluzione di un problema classico nell'ambito della teoria dei grafi.

È importante conoscere questi problemi classici, così da poterli riconoscere. Molti di questi problemi sono NP-HARD, ma esistono anche molti problemi sui grafi che possono essere risolti in tempo polinomiale.

Problemi di ottimizzazione combinatoria sui grafi sono generalmente visti come esempi di problemi di ottimizzazione combinatoria.

### 7.1 Problemi di ottimizzazione combinatoria

Determinare un sottoinsieme fattibile  $S$  di elementi, dato un universo  $E$ . Ciascun elemento  $e \in E$  possiede un costo associato  $c(e)$ , e il costo di un insieme  $S \subseteq E$  è definito come  $c(S) := \sum_{e \in S} c(e)$ .

Questi problemi sono chiamati combinatorici perché tipicamente un sottoinsieme è fattibile se identifica un oggetto matematico combinatorio e discreto, come una permutazione o un sottoinsieme di indici in  $[n]$ , un ciclo o un cammino in un grafo, un mapping tra due insiemi finiti, etc.

Dato che l'ILP è una tecnica di risoluzione "general-purpose", potremmo usarla anche per questi problemi di ottimizzazione combinatoria, ma molte volte algoritmi alternativi che sfruttano la particolare struttura di questi problemi sono possibili.

Molti di questi problemi sui grafi possono essere modellati come problemi di programmazione lineare intera (ILP) e risolti come problemi di programmazione lineare (LP) in tempo polinomiale perchè la loro matrice dei vincoli risulta essere TUM. Inoltre, esistono algoritmi veloci che non utilizzano tecniche di LP, ma piuttosto determinano la soluzione ottimale costruendola in modo incrementale, o iniziando con una soluzione fattibile e applicando qualche cambiamento finché si può provare che risulta essere la sol. ottimale.

### 7.2 Rappresentazione dei grafi e strutture dati

Senza perdita di generalità, assumiamo che l'insieme dei vertici sia un sottoinsieme di  $\mathbb{N}$ , tipicamente  $V = [n]$ . Denotiamo il numero di archi come  $m$ .

I grafi possono essere espressi sia in forma **densa** che **sparsa**, e rispettivamente possono essere chiamati densi o sparsi al variare del numero di archi rispetto i nodi.

- Per i grafi **densi** il numero di archi è almeno un fattore costante del massimo possibile, ad esempio:

$$m \geq \alpha n^2$$

per qualche costante  $\alpha$ , e quindi  $m = \Theta(n^2)$ .

- Per i grafi **sparsi** il numero di archi è molto piccolo (limitato da un fattore lineare del numero di nodi) ad esempio:  $m = O(n)$ .

Perciò, un grafo può essere memorizzato sia in forma densa che sparsa.

---

**Forma densa:** tipicamente si utilizza una matrice di adiacenza  $n \times n$   $M$  tale che  $M[i, j] = \text{TRUE}$ , sse c'è un vertice tra  $i$  e  $j$ . È simmetrica per un grafo indiretto e asimmetrica per un grafo diretto. Il vantaggio di una matrice di adiacenza è che in tempo  $O(1)$  possiamo vedere se due nodi sono adiacenti o meno, o possiamo inserire/eliminare un arco. D'altro canto, accedere a tutti i vicini di un nodo  $v$  richiede tempo  $\Theta(n)$  piuttosto che  $\Theta(|\delta(v)|)$ . Se un grafo è denso, la forma densa non andrebbe evitata in quanto ottimale. La forma densa richiede una memoria  $m = \Theta(n^2)$ .

---

**Forma sparsa:** richiede memoria proporzionale a  $m$  dato che vale sempre  $m = O(n^2)$ . La forma sparsa richiede meno memoria fino a che  $m = \Theta(n^2)$ , caso in cui la forma sparsa e densa sono equivalenti (entrambi lineari nella dimensione del grafo).

Una possibile forma sparsa è una lista con tutti gli archi. Per ogni arco memorizziamo i suoi punti finali in un array, come  $[ \cdot, k ]$  per  $k = 1, \dots, m$ . Il  $k$ -esimo arco è  $(\text{arc}[1, k], \text{arc}[2, k])$ . Controllare se due nodi sono adiacenti, o accedere a tutti i vicini di un dato nodo  $v$  richiede tempo  $O(m)$ , dato che dovremmo passare per tutti gli archi per controllare se esiste o meno  $[1, k] = v$  o  $[2, k] = v$ .

Possiamo migliorare le performance con una lista di "stelle": una lista di archi, ma ciascun elemento è ripetuto due volte ed è anche accessibile direttamente tramite uno dei suoi punti finali (endpoints).

Per ciascun nodo  $v$  manteniamo un contatore del grado  $[v]$  per memorizzare il suo grado, e un'array  $[v]$  per memorizzare tutti i suoi vicini nelle celle:

$$\text{neigh}[v, 1], \dots, \text{neigh}[v, \text{degree}[v]].$$

Se  $G$  è diretto, abbiamo due liste, forward neighbors e backward neighbors ( $\text{neigh} = \text{neighbors} = \text{vicini}$ ). Accedere a tutti i vicini di  $v$  richiede tempo  $\Theta(d(v))$ , dove  $d(v)$  è il grado di  $v$ . Controllare se un certo arco  $(i, j)$  è nel grafo richiede tempo  $O(\min\{d(i), d(j)\})$  dato che scandiamo la lista più piccola tra le due liste di vicini e guardiamo per gli altri punti finali. Inserire un arco  $(i, j)$  richiede tempo  $O(1)$  estendendo la lista di  $i$  e  $j$  alla loro fine e incrementando entrambi i loro gradi.

L'espressione:

$$\text{forAllNeigh } (w \text{ IN } \delta(v))$$

significa che iteriamo sulla lista dei vicini di  $v$  restituendo, ogni volta, in tempo  $O(1)$ , in  $w$  il  $k$ -esimo vicino per  $k = 1, \dots, d(v)$ .

Se il grafo è diretto, possiamo guardare i vicini raggiungibili dagli archi in avanti o archi all'indietro come:

$$\text{for AllNeigh } (w \text{ IN } \delta^+(v)) \quad \text{e} \quad \text{for AllNeigh } (w \text{ IN } \delta^-(v))$$

### 7.3 Visite dei grafi

Iniziamo con un esempio, il problema della connessione:

**Problema:** dato un grafo indiretto  $G = ([n], E)$ , si trovi se  $G$  è connesso, e, in caso contrario, determinare tutte le sue componenti connesse. Questo è un problema polinomiale e l'algoritmo è una visita al grafo.

Si assegna ad ogni nodo  $v \in [n]$  un'etichetta  $l[v]$ . Alla fine,  $l[v]$  specifica l'indice della componente connessa a cui  $v$  appartiene.

Per trovare la  $k$ -esima componente, iniziamo in qualsiasi nodo  $v_0$  che non era in nessuna delle componenti  $1, \dots, k-1$ , e si prova a determinare tutti i nodi  $w$  che possono essere raggiunti da qualche cammino da  $v_0$ , etichettandoli con  $l[w] := k$ . Poi, se ci sono ancora nodi non etichettati, aumentiamo  $k$  e ripetiamo. Se l'etichetta più alta usata è  $k = 1$ , allora il grafo è connesso.

La visita usa una struttura dati  $S$  che può essere uno stack (DFS, Depth-First Search) o una coda (BFS, Breadth-First Search). Durante l'esecuzione dell'algoritmo, ogni nodo  $v$  è esattamente in uno dei seguenti tre stati:

1.  $v$  non è etichettato (dobbiamo ancora raggiungere  $v$ );
2.  $v$  è etichettato e  $v \notin S$ ;
3.  $v$  è etichettato e  $v \in S$ .

I nodi dell'ultimo tipo sono le "frontiere" del componente corrente, usate per propagare le etichette a nuovi nodi non etichettati.

Si supponga che si stiano etichettando i nodi del componente  $C_k$ . All'inizio,  $S$  contiene solo  $v_0$ , con  $l[v_0] := k$ , e tutti gli altri nodi di  $C_k$  sono non etichettati. L'algoritmo è un loop in cui estraiamo da  $S$  un vertice  $v$ , si prendono tutti i vicini non etichettati  $w$  di  $v$ , si etichettano con  $l[w] := k$  e li mettiamo in  $S$ . Quando  $S$  diventa vuoto, l'etichettamento del  $k$ -esimo componente è finito.

La complessità computazionale risulta  $O(n + |E|)$ . Il lavoro totale può essere suddiviso come il tempo di inserimento/estrazione di ciascun nodo da  $S$  (che è  $O(1)$  per nodo) più il lavoro per propagare le etichette, che è proporzionale alla somma dei gradi, cioè:  $O(|E|)$ .

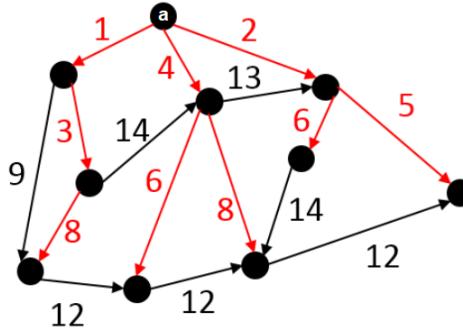
## Pseudocodice di Componenti Connesse:

```

1. for  $i = 1$  to  $n$  do  $I[i] := 0$  /*  $i$  is unlabeled */
2.  $k := 0$  /* Components found so far */
3. for  $i = 1$  to  $n$  if  $I[i] = 0$  then /*label component  $k$  */
4.    $k := k + 1$ ;
5.    $I[i] := k$ ; insert( $S, i$ )
6.   while  $S \neq \emptyset$  do
7.      $v = \text{extract}(S)$ 
8.     forAllNeigh( $w \in \delta(v)$ ) s.t.  $I[w] = 0$  do
9.        $I[w] = k$ ; insert( $S, v$ )
10.    end forAllNeigh
11.   end while
12. end for

```

Se  $G$  è diretto, l'algoritmo può determinare un insieme di arborescenze che coprono tutti i nodi. Ciascuna arborescenza è un albero con radice in qualche  $v$ . Per ricostruire l'albero associamo a ciascun nodo  $i$  un'etichetta  $P[i]$  che punta ai suoi genitori (si ha,  $P[i] = w$  se  $i$  è stato messo in  $S$  come vicino di  $w$  quando  $w$  è stato usato per propagare la sua tabella). Segue un esempio di arborescenza con root: a.



## 7.4 Grafi aciclici e ordine topologico

Un'importante classe di grafi sono i grafi diretti aciclici (DAG). Molti problemi difficili sui grafi generali diretti sono più semplici sui DAG.

Inoltre, per alcuni problemi polinomiali, l'algoritmo può essere molto più efficiente quando l'input è un DAG.

Un grafo aciclico diretto (DAG)  $G = (V, A)$  induce a un ordine parziale su  $V$ , cioè  $i \prec j$  se esiste un cammino in  $G$  da  $i$  a  $j$ . Se  $G$  contiene un cammino Hamiltoniano, allora  $G$  rappresenta un ordine totale.

Per un ordine totale esiste esattamente un vertice  $v \in V$  (il "minimo" – da  $v$  posso raggiungere ogni nodo) t.c.  $v \prec i$  per ogni  $i \neq v$  e un vertice  $w$  (il "massimo" -  $w$  è raggiungibile da ogni nodo) t.c.  $i \prec w$  per ogni  $i \neq w$ .

**Fonti e pozzi:** In un DAG esiste sempre uno o più nodi  $w$  t.c.  $d^-(w) = 0$  ("fonti"). Similmente, esiste sempre un nodo  $v$  t.c.  $d^+(v) = 0$  ("pozzi").

Può essere mostrato per contraddizione: si supponga che ciascun nodo abbia  $d^+(v) > 0$ . Iniziando al nodo  $v_0$ , si segua un cammino random  $v_0, v_1, v_2, \dots$  nel grafo. Ogni volta che entra un nodo  $v_k$  possiamo usare un arco  $(v_k, v_{k+1})$  per uscire da  $v_k$ , dato che  $d^+(v_k) > 0$ .

Per il principio della piccionaia, dopo  $|V| + 1$  passi almeno un nodo  $v$  nel cammino deve esser stato ripetuto (visitato 2 volte). Questo implica che esiste un ciclo da  $v$  a  $v$ , impossibile in un DAG.

Un **sorting topologico** di un DAG  $G = (V, A)$  è un etichettamento (labeling)  $I : V \mapsto \{1, \dots, |V|\}$  t.c.

$$I(i) < I(j) \text{ per ogni } (i, j) \in A$$

Un DAG  $G = (V, A)$  è già topologicamente ordinato se  $(i, j) \in A$  implica  $i < j$ . In un DAG ordinato topologicamente, molti problemi sui cammini (come il cammino minimo o massimo tra un nodo e gli altri) possono essere risolti da algoritmi molto veloci, con complessità computazione  $O(|A|)$ .

Un sorting topologico può essere ottenuto in due modi equivalenti, cioè procedendo in avanti o all'indietro (forwards or backwards).

Nel primo caso, etichettiamo prima le fonti e continuiamo finchè non abbiamo etichettato l'ultimo pozzo. Nel secondo caso iniziamo dal pozzo e continuiamo finchè non etichettiamo l'ultima fonte.

**L'idea alla base del backward labeling:** Sia  $v'$  un nodo (pozzo) con  $d^+(v') = 0$ . Si assegna l'etichetta  $n$  a  $v'$  (non può essere sbagliata, dato che  $n$  è l'etichetta più alta possibile e  $v'$  può solo essere in testa, e mai in coda di nessun arco). Poi, rimuoviamo  $v'$  e  $\text{arcs } \delta^-(v')$ , ottenendo un DAG  $G'$  di  $n - 1$  nodi. Dato che  $G'$  è un DAG, ripetiamo il processo su  $G'$  etichettando uno dei suoi pozzi con  $n - 1$  e così via.

**L'idea alla base del forward labeling:** è simile, ma iniziamo etichettando con un 1 una fonte  $v'$  di  $G$ , e rimuoviamo  $v'$  e  $\delta^+(v')$  per ottenere  $G'$ . Poi etichettiamo una fonte di  $G'$  con 2, e così via.

Infatti, il DAG non è aggiornato eliminando nodi e/o archi, ma è sufficiente mantenere i contatori del grado in entrata corrente dei nodi mentre l'algoritmo è in esecuzione. Ad esempio, nel forward labeling, quando il grado in entrata diventa 0, un nodo deve diventare la fonte e diventa pronto per essere etichettato.

L'algoritmo di etichettamento è una visita che usa una struttura dati  $S$  per memorizzare la frontiera corrente del sottografo visitato.

### Esempio: implementazione con stack

Descriviamo, a titolo esplicativo, il caso in cui  $S$  è uno stack: Associamo a ciascun nodo un contatore  $\deg[i]$ , inizializzato con  $d^-(i)$ . I nodi nello stack in ciascun istante sono nodi che sono fonti del DAG corrente  $G'$ , e già stati etichettati.

Si assume che le etichette siano  $l[v]$ , per ciascun  $v \in V$ . C'è un contatore  $cnt$  globale che conta i nodi già etichettati.

Alla fine, ci aspettiamo di avere  $cnt = n$ . Se  $cnt < n$  allora l'input  $G$  non era un DAG e viene riportato un errore.

## Chiusura transitiva

Un DAG  $G$  definisce un ordine parziale  $\prec$ , t.c. ” $i \prec j$  se esiste un cammino diretto in  $G$  da  $i$  a  $j$ .

Non diciamo che  $i \prec j$  se  $(i, j) \in A$  perché per un ordine abbiamo bisogno della proprietà transitiva, ma non è garantito che  $A$  sia transitivo (cioè, potrebbe essere  $(i, j) \in A, (j, k) \in A$  ma  $(i, k) \notin A$  per qualche  $i, j, k \in V$ ) (quindi non varrebbe la proprietà transitiva se la definizione chiedesse la presenza di un arco da  $i$  a  $j$  invece che da  $i$  raggiungiamo  $j$ , se invece questa proprietà è valida, otteniamo una chiusura transitiva).

Dato  $G$  (non necessariamente aciclico), il grafo  $c(G) = (V, c(A))$  con un arco  $(i, j) \in c(A)$  per ogni  $i, j \in V$  t.c. esiste un cammino da  $i$  a  $j$  in  $V$  è chiamato chiusura transitiva di  $G$ .

Un grafo è un DAG sse la sua chiusura transitiva è anche un DAG. Inoltre, se  $G = (V, A)$  è un DAG e definiamo  $\prec$  da  $i \prec j \iff (i, j) \in A$ , allora  $\prec$  è un ordine sse  $c(G) = G$ .

Possiamo calcolare la chiusura transitiva di un grafo  $G$  in tempo  $O(n^3)$  sfruttando la seguente proprietà ricorsiva, per  $k = 1, \dots, n$ :

1. Esiste un cammino da  $i$  a  $j$  usando solo nodi in  $[k]$  come nodi intermedi sse esiste un cammino da  $i$  a  $j$  usando solo nodi in  $[k - 1]$  come nodi intermedi,

OPPURE

2. Esistono sia un cammino da  $i$  a  $k$  e un cammino da  $k$  a  $j$  usando solo nodi in  $[k - 1]$  come nodi intermedi.

Possiamo quindi impostare un algoritmo in cui iteriamo su tutte le possibili  $k = 1, \dots, n$ .

## Algoritmo chiusura transitiva

Usiamo una matrice  $n \times n$  detta  $P$  di booleani inizializzati tali da essere la matrice di adiacenza di  $G$ . Alla fine della  $k$ -esima iterazione,  $P[i, j] = \text{TRUE}$  per ogni coppia  $i$  e  $j$  tale che esiste un cammino da  $i$  a  $j$  i cui nodi intermedi sono tutti contenuti in  $[k]$ :

```

1. for  $i \in [n]$  for  $j \in [n]$  do
2.   if  $(i, j) \in A$  then  $P[i, j] := \text{TRUE}$  else  $P[i, j] := \text{FALSE}$ 
3.   end for
4.   for  $k := 1, \dots, n$ 
5.     for  $i := 1, \dots, n$ 
6.       for  $j := 1, \dots, n$ 
7.          $P[i, j] := P[i, j] \vee (P[i, k] \wedge P[k, j])$ 
8.       end for
9.     end for
10.   end for

```

(Se nello step 2 inizializziamo la diagonale  $P[i, i] = \text{TRUE}$  per ogni  $i$ , allora consideriamo ciascun nodo raggiungibile da se stesso con un cammino di 0 archi. Se, contrariamente lo inizializziamo  $P[i, i] = \text{FALSE}$  per ogni  $i$ , allora all fine  $P[i, i] = \text{TRUE}$  solo per i nodi che appartengono a qualche ciclo di lunghezza  $\geq 2$ ).

## 8 Minimo Albero di Copertura – Minimum Spanning Tree (MST)

Il problema dell'albero minimo di copertura (MST) è un esempio di network design: vogliamo costruire una rete che connette un insieme di siti. Una rete ha due costi associati, un costo di costruzione (pagato una volta per tutte) e un costo di utilizzo. Ci concentriamo sul costo di costruzione, che sappiamo essere pari a  $c(i, j)$  per costruire una connessione tra  $i$  e  $j$ .

Il problema è costruire la rete più economica possibile ma connessa, dove il minimo grafo a garantire la connessione è uno spanning tree. Segue la formalizzazione del problema

**PROBLEMA:** Minimum Spanning Tree (MST).

**INPUT:** Sia  $G = (V, E)$  un grafo con pesi non negativi,  $c(e)$  o  $c(i, j)$ , negli archi.

**OBIETTIVO:** Determinare uno spanning tree di costo minimo dove il costo di  $T = (V, E_T)$  è:

$$c(T) := \sum_{e \in E_T} c(e)$$


---

Il problema MST può essere risolto con un algoritmo greedy. L'approccio greedy consiste nella scelta di archi che mettiamo nella soluzione ottimale (l'MST che stiamo costruendo) sempre provando ad aggiungere un arco di costo minimo tra un insieme di candidati disponibili.

**L'algoritmo greedy:** Si definisce  $A \subseteq E$  un **insieme estendibile** se esiste qualche spanning tree di costo minimo  $T^* = (V, E_{T^*})$  t.c.  $A \subseteq E_{T^*}$ . Una condizione necessaria per un insieme per essere estendibile è di essere aciclico. Ovviamente,  $\emptyset$  è estendibile. Se  $A$  è un insieme estendibile e  $e \in E \setminus A$  è t.c.  $A \cup \{e\}$  è anche un insieme estendibile, allora  $e$  è un **arco safe** per  $A$ . Un insieme estendibile è una soluzione potenzialmente ottima. Un insieme per essere estendibile non deve avere cicli. Segue una versione generica di algoritmo per trovare un MST:

**GenericMST – Pseudocodice:**

1.  $A := \emptyset$
2. for  $i := 1$  to  $n - 1$  do
3.     Trova un arco  $e \in E - A$  che sia safe per  $A$ ;
4.      $A := A \cup \{e\}$
5. end for

Alla fine,  $A$  è l'insieme di archi di uno spanning tree, dato che un albero ha esattamente  $n - 1$  archi. Per definizione di insieme estendibile, l'albero trovato ha costo minimo.

La procedura appena descritta non può ancora essere chiamata algoritmo, per via del fatto che lo step 3 non è stato descritto a sufficienza, e non è chiaro come determinare un arco safe per  $A$ . Ora descriviamo una condizione sufficiente (ma non necessaria) per un arco per essere safe per  $A$ .

**Teorema:** Sia  $A$  un insieme estendibile e sia  $V'$  l'insieme di vertici di ogni componente连通 in  $(V, A)$ . Allora, ogni arco  $e \in \delta(V')$  (taglio) t.c.  $c(e) = \min \{c(a) : a \in \delta(V')\}$  è un arco safe per  $A$ .

Concettualmente: prendiamo l'arco di costo minimo tra tutti gli archi del taglio (in uscita) di una componente连通.

**Dimostrazione:** Sia  $e = ij$ , dove  $i \in V'$  e  $j \notin V'$  e sia  $T^* = (V, E_{T^*})$  un MST t.c.  $A \subseteq E_{T^*}$ . Se  $e \in E_{T^*}$  allora  $e$  è safe. Altrimenti, c'è un cammino tra  $i$  e  $j$  in  $T^*$  che non contiene l'arco  $e$ . Questo cammino contiene necessariamente un arco  $f \in \delta(V')$ . Dato che  $c(e) \leq c(f)$ , rimpiazzando  $f$  con  $e$  otteniamo un nuovo spanning tree  $T$  il cui costo non può essere maggiore di quello di  $T^*$ . In particolare, dato che  $T^*$  aveva costo minimo, anche  $T$  deve avere costo minimo. Dato che gli archi di  $T$  includono  $A \cup \{e\}$ ,  $e$  è un arco sicuro per  $A$ . Si noti che la condizione non è necessaria. Si ha che se  $G$  è già un albero, allora ogni arco di  $\delta(V')$  è safe, non solo quello di costo minimo.

### Strategie di Prim e Kruskal

La scelta dell'arco safe può essere effettuata seguendo una delle due popolari strategie: l'algoritmo di Prim o l'algoritmo di Kruskal.

Entrambi partono da un insieme estendibile vuoto, che induce a una foresta di  $n$  componenti, ciascuna con un nodo singolo.

A ciascuna iterazione, due componenti connesse, diciamo  $a$  e  $b$  della foresta corrente vengono fusi in un'unica componente fino a che la foresta non diventa uno spanning tree. Le differenze tra i due metodi sono le seguenti:

**Strategia di Prim:** Nell'algoritmo di Prim, la componente  $a$  è sempre la componente che contiene un nodo particolare (ad esempio: 1). Questo nodo è la radice dell'MST creato dalla procedura. La componente  $b$  consiste sempre di un nodo singolo. Possiamo vedere l'esecuzione dell'algoritmo di Prim come la creascita di un sottoalbero  $\mathcal{T}$  (che inizialmente consiste solo del nodo 1) in cui a ciascun step aggiungiamo una nuova foglia, la più vicina a  $\mathcal{T}$ . L'arco safe che viene aggiunto a ciascuna iterazione è quello di costo minimo tra gli archi di  $\delta(a)$  (raggiungibili da  $a$ ).

**Strategia di Kruskal:** Nell'algoritmo di Kruskal  $a$  e  $b$  possono essere qualsiasi delle componenti della foresta corrente. All'inizio della  $k$ -esima iterazione, la foresta corrente consiste di  $n - k + 1$  sottoalberi, e ci possono essere più di un sottoalbero non-banale. Eventualmente, tutti gli alberi vengono fusi in un albero unico. Le componenti  $a$  e  $b$  da unire vengono scelte nel seguente modo: troviamo un arco di costo minimo tra tutti gli archi i cui punti finali appartengono a componenti diverse, e quindi uniamo le due componenti, contenenti i punti finali.

Collegamenti tra archi safe (in entrambi gli algoritmi), possono essere rotti arbitrariamente. Al seguente link è possibile trovare una spiegazione dettagliata dei due algoritmi, con due esempi di esecuzione: [link video](#).

## Implementazione "semplice" dell'algoritmo di Prim

Una semplice implementazione di complessità  $\Theta(n^2)$ . Questa è ottimale (lineare sulla dimensione dell'input) assumendo che l'input sia un grafo completo (o molto denso), rappresentato da una matrice simmetrica  $n \times n$  detta  $c$ .

Se il grafo  $G$  non fosse completo, potremmo aggiungere gli archi mancanti con costo  $c(i, j) := M$  ( $M$  un upper bound molto alto). Se  $G$  fosse di fatto denso, allora la sua dimensione sarebbe  $\Theta(n^2)$  e questo processamento non peggiorerebbe l'efficacia dell'algoritmo, che rimane ottimale. Però, se  $G$  fosse sparso, allora esistono implementazioni più efficienti dell'algoritmo di Prim.

Si denoti con  $\mathcal{T}$  l'MST parziale costruito dall'algoritmo, cioè: il sottoalbero contenente il nodo 1. Facciamo uso dei seguenti array:

- $flag[i]$ : booleano che è vero sse  $i$  è in  $\mathcal{T}$ ;
- $L[i]$  : per ciascun vertice  $i \notin \mathcal{T}$  il valore  $L[i]$  rappresenta il minimo costo di qualsiasi arco  $ji$  con  $j \in \mathcal{T}$ . Questo array è insignificante per  $i \in \mathcal{T}$ .
- $P[i]$  : per ogni vertice  $i \notin \mathcal{T}$  l'etichetta  $P[i]$  denota un vertice  $j \in \mathcal{T}$  t.c.  $L[i] = c(j, i)$ . Per i vertici  $i \in \mathcal{T}$ ,  $P[i]$  denota il padre di  $i$  nello spanning tree. Per convenzione, il padre della radice è la radice stessa.

```

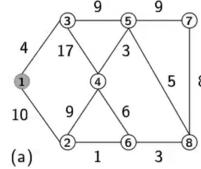
1.  flag[1] := 1; P[1] := 1; /* inserts the root in T */
2.  for k = 2, ..., n do
3.    L[k] := c(1, k); P[k] := 1; flag[k] := 0;
4.  end for
5.  for e = 1, ..., n - 1 do
6.    val := +∞; w := -1;
7.    for k = 2, ..., n do
8.      if (flag[k] = 0) ∧ (L[k] < val) then
9.        val := L[k]; w := k;
10.       end if
11.    end for
12.    flag[w] := 1; /* add w to T */
13.    for k = 2, ..., n do
14.      if (flag[k] = 0) ∧ (L[k] > c(w, k)) then
15.        L[k] := c(w, k); P[k] := w;
16.      end if
17.    end for
18.  end for

```

Gli step 1 – 4 sono inizializzazioni. Il loop 5 – 18 è eseguito  $n - 1$  volte e ogni volta un arco/nodo viene aggiunto a  $\mathcal{T}$ . Le istruzioni 6 – 11 determinano il punto finale  $w$  di un arco di costo minimo di  $\delta(\mathcal{T})$ . Ci sono potenziali  $\Theta(n^2)$  archi in  $\delta(\mathcal{T})$  ma troviamo quello di costo minimo in tempo  $\Theta(n)$  dato che conosciamo per ogni  $k \notin \mathcal{T}$  l'arco migliore che finisce in  $k$  (chiamarlo arco  $\{P[k], k\}$ ).

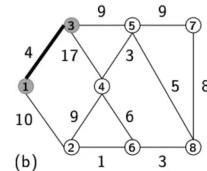
**Esempio esecuzione dell'algoritmo di Prim:** segue un esempio di esecuzione dell'algoritmo di Prim:

Passo 1:



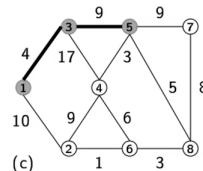
	1	2	3	4	5	6	7	8
flag:	1	0	0	0	0	0	0	0
$L$ :	0	10	4	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$P$ :	1	1	1	1	1	1	1	1

Passo 2:



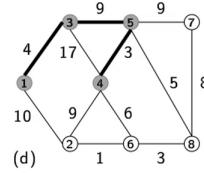
	1	2	3	4	5	6	7	8
flag:	1	0	1	0	0	0	0	0
$L$ :	0	10	4	17	9	$+\infty$	$+\infty$	$+\infty$
$P$ :	1	1	1	3	3	1	1	1

Passo 3:



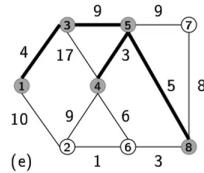
	1	2	3	4	5	6	7	8
flag:	1	0	1	0	1	0	0	0
$L$ :	0	10	4	3	9	$+\infty$	9	5
$P$ :	1	1	1	5	3	1	5	5

Passo 4:



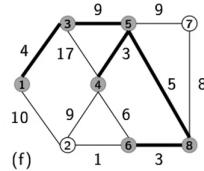
	1	2	3	4	5	6	7	8
flag:	1	0	1	1	1	0	0	0
L:	0	9	4	3	9	6	9	5
P:	1	4	1	5	3	4	5	5

Passo 5:



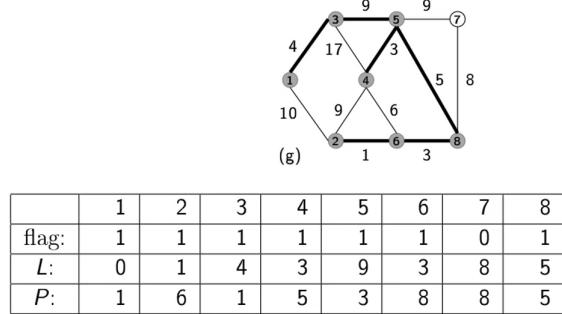
	1	2	3	4	5	6	7	8
flag:	1	0	1	1	1	0	0	1
L:	0	9	4	3	9	3	8	5
P:	1	4	1	5	3	8	8	5

Passo 6:



	1	2	3	4	5	6	7	8
flag:	1	0	1	1	1	1	0	1
L:	0	1	4	3	9	3	8	5
P:	1	6	1	5	3	8	8	5

Passo 7:



**Prim con heaps:** L'implementazione  $\Theta(n^2)$  non è adatta per grafi sparsi e l'aggiornamento delle etichette  $L$  (step 14 – 16) dovrebbe essere proporzionale a  $|\delta(w)|$  invece che  $\Theta(n)$ . Si assume che tutti i vertici siano posizionati in una min-heap (radice sempre di costo minimo, invariante delle sotto-heap), ordinata rispetto i loro valori in  $L[ ]$ .

- Per ogni nodo  $i$  teniamo traccia della sua posizione nella heap, così da essere in grado di accedere agli elementi della heap in tempo  $O(1)$ ;
- Costruire la heap ha costo  $O(n)$ ;
- Prendere il vertice  $w$  con minor valore  $L[ ]$  può essere fatto in tempo  $O(\log n)$  (cioè:  $O(1)$  per prendere  $w$  più  $O(\log n)$  per ripristinare la proprietà della heap);
- Quando  $w$  è stato determinato, in tempo  $\Theta(d(w))$ , usando la lista di stelle, accediamo a tutti i suoi vicini nella heap e, per ciascuno di essi per cui sono cambiati i valori in  $L$ , possiamo aggiornare la sua posizione nella heap in tempo  $O(\log n)$ . Questo step ha un costo  $O(d(w) \log n)$ . Aggiungendo tutto insieme su  $w$  porta a una complessità finale di  $O(m \log n)$  per questa implementazione dell'algoritmo di Prim.

### Implementazione algoritmo di Kruskal

La strategia può essere implementata efficacemente come segue. Inizialmente, ordiniamo gli archi in  $E$  in ordine non decrescente rispettivamente al loro costo, in tempo  $O(m \log m) = O(m \log n)$ , ovvero:

$$c(e_1) \leq c(e_2) \leq \dots \leq c(e_m)$$

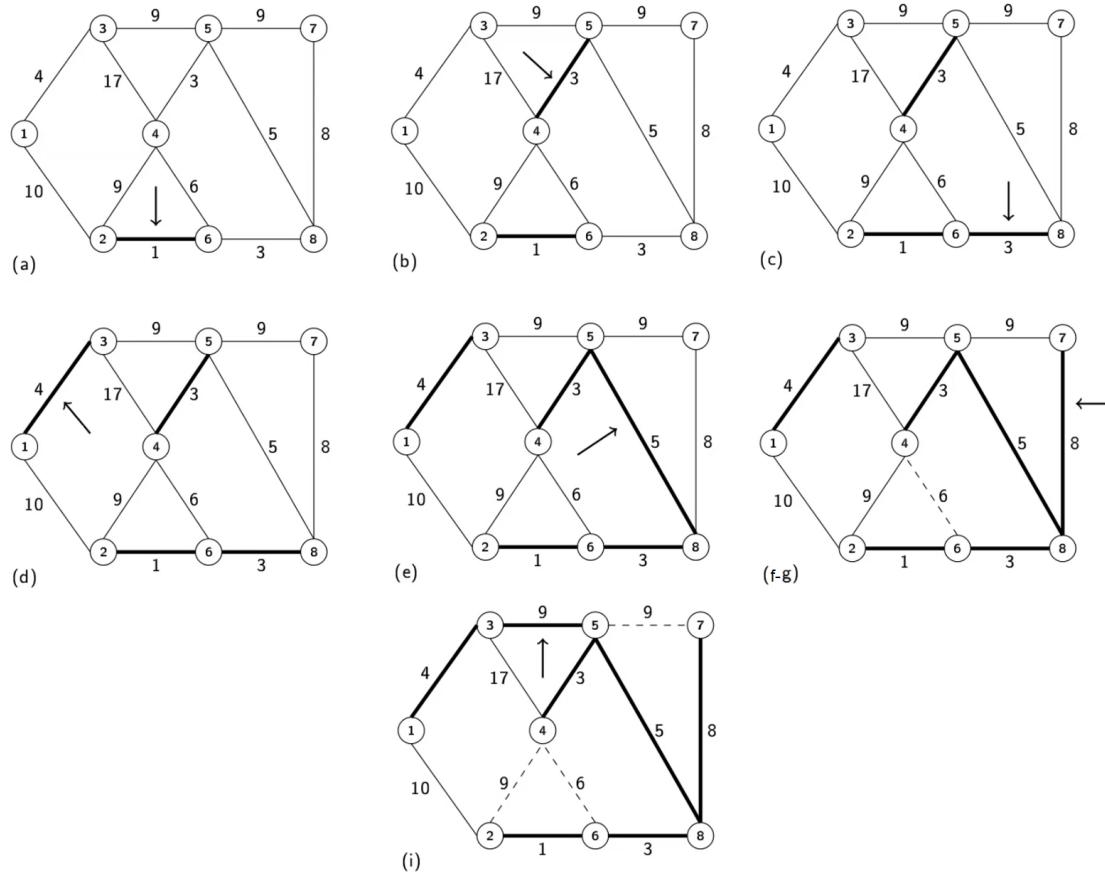
Iniziando con  $n$  componenti connesse, ciascuna consistente di un singolo nodo, analizziamo la lista ordinata per  $k = 1, \dots, m$ , fermandoci prima possibile dato che  $n - 1$  archi sono stati messi nell'albero.

Ad ogni passo, se i punti finali di un arco  $e_k$  appartengono allo stesso componente连通的，则我们跳过它。否则，我们将该边添加到我们的森林中，并将两个组件合并为一个。

Una implementazione efficace dell'idea usa strutture dati per fondere due componenti più velocemente possibile.

- **Implementazione naïve:** si usa un array  $\text{comp}[]$  t.c.  $\text{comp}[i] = \text{numero del componente a cui } i \text{ appartiene}$ . Controllare se due nodi sono nello stesso componente costa  $O(1)$ , ma fondendo due componenti richiede  $\Theta(n)$ . Dobbiamo analizzare  $O(m)$  archi, quindi un'analisi superficiale fa sì che l'algoritmo richieda in tutto tempo  $O(mn)$ . Solo  $O(n)$  volte paghiamo  $\Theta(n)$  per fondere due componenti, quindi la complessità è  $O(\max\{m \log n, n^2\})$  che è peggio che entrambe le implementazioni dell'algoritmo di Prim.
- **Implementazione buona:** Se potessimo fondere due componenti in tempo  $\log(n)$  allora l'algoritmo costerebbe  $O(m \log n)$  (dato che l'ordinamento iniziale racchiude la maggior parte del costo computazionale). In grafi sparsi, questa è un'implementazione efficace dell'algoritmo quanto Prim basato su heap. Inoltre, utilizzando strutture dati di tipo "Union-Find" per memorizzare i nodi di ciascuna componente connessa, permette di implementare l'algoritmo di Kruskal in tempo  $O(m \log n)$ .

**Esempio esecuzione dell'algoritmo di Kruskal:** segue un esempio di esecuzione dell'algoritmo di Kruskal appena descritto:



## 8.1 Casi speciali di Spanning Trees

### Spanning Tree con archi di costo negativo e Max Spanning Tree

Assumiamo che tutti i costi siano  $\geq 0$ . Mostriamo ora che la presenza di costi negativi non cambia la complessità del problema. Si assuma che i costi  $c$  siano sia positivi che negativi e sia  $C = \min_e c(e)$ . Se ridefiniamo i costi degli archi come:

$$c'(e) := c(e) - C$$

allora è  $c'(e) \geq 0$  per ogni  $e \in E$ . Inoltre, per ogni spanning tree  $T$  si ha:

$$c'(T) = c(T) - (n - 1)C$$

(ogni albero ha esattamente  $n - 1$  archi). Inoltre, per ogni albero  $T_1$  e  $T_2$  si ha:

$$c'(T_1) \leq c'(T_2) \iff c(T_1) \leq c(T_2)$$

quindi l'MST per i costi iniziale è lo stesso MST per i costi non negativi  $c'(\cdot)$ .

Trovare un **max spanning tree** può essere risolto come un MST, dato che se definiamo  $w(e) = -c(e)$  per ogni  $e \in E$ , allora  $T$  è di costo minimo per  $w()$  sse è il costo massimo per  $c()$ . Però non facciamo questo, ma invertiamo le diseguaglianze e eseguiamo l'algoritmo di Prim o Kruskal.

### Alcune considerazioni finali:

- I problemi di spanning tree in cui vogliamo minimizzare il massimo grado (o il massimo grado è vincolato ad essere  $\leq M$ ) sono NP-hard dato che modellano il problema del cammino hamiltoniano (un grafo con tutti i gradi  $\leq 2$ );
- Possiamo usare algoritmi MST per controllare se un grafo  $G = (V, E)$  è connesso: dato peso 1 agli archi in  $E$  e  $\geq 2$  agli archi  $\notin E$ . Allora  $G$  è connesso sse il costo di MST è  $n - 1$ . Se il costo è  $\geq n$  allora ha utilizzato almeno un arco di peso 2, quindi non è connesso;
- Il problema del minimo albero di collo di bottiglia (minimum bottleneck spanning tree (MBST)) richiede un spanning tree  $T = (V, F)$  t.c.  $\max_{e \in F} c(e)$  è minimo possibile (cioè l'arco di peso maggiore usato è di peso minore possibile). Perciò un algoritmo per MST risolve anche MBST (cioè ogni MST è un MBST – ma il contrario non è vero e potrebbero esserci algoritmi specifici più veloci di Prim e Kruskal).

## 9 Cammini ottimali

Un importante problema di ottimizzazione combinatoria consiste nel trovare un cammino ottimale tra due dati nodi, o tra tutte le coppie di nodi di un grafo in input  $G$ .

Questo problema corrisponde a ciò che devono risolvere i navigatori GPS ogni volta viene impostata una posizione  $d$  da una posizione corrente  $p$ .

Il cammino ideale è un cammino  $p \rightarrow d$  che minimizza il totale della distanza percorsa (shortest path) o un cammino  $p \rightarrow d$  che minimizza il totale del tempo di percorrenza.

Gli archi del grafo sono pesati dal costo che richiede la loro percorrenza: ad esempio,  $G$  potrebbe essere una rete di strade che connettono alcune città e  $c(i, j)$  è la lunghezza delle strade tra la città  $i$  e la città  $j$ , o il costo monetario per la percorrenza.

Ci concentriamo nel caso in cui  $G = (V, A)$  è un grafo diretto. Un grafo indiretto è un caso particolare di grafo diretto (usando lo stesso peso per entrambi gli archi indiretti, rappresentiamo un arco diretto). La maggior parte delle volte, i pesi dell'arco rappresentano distanze o tempi, e d'ora in poi saranno numeri non negativi.

In generale, tuttavia, i pesi non devono essere necessariamente non negativi. Un peso negativo potrebbe rappresentare un profitto che otteniamo utilizzando un arco  $(i, j)$  (ad esempio, un venditore realizza un profitto in alcune strade dove vivono i suoi clienti ma deve anche attraversare strade dove sa che non venderà nulla).

In generale, quindi, un percorso potrebbe essere costituito da archi di peso sia negativo che positivo.

Un cammino  $P$  è una sequenza di vertici  $P = (v_0, \dots, v_k)$  tali che  $(v_i, v_{i+1}) \in A$  per ogni  $i = 0, \dots, k - 1$ . Il peso (o lunghezza) di  $P$  è definito come:

$$c(P) := \sum_{i=0}^{k-1} c(v_i, v_{i+1})$$

Identifichiamo gli archi del cammino  $P$  come  $A(P) := \{(v_0, v_1), \dots, (v_{k-1}, v_k)\}$ , allora possiamo vedere  $c(P)$  anche come sommatoria dei costi degli archi di  $P$ .

Il problema base dei percorsi ottimali è il problema del percorso di peso minimo (in genere chiamato **problema del percorso più breve – shortest path problem**) tra una coppia specifica di nodi  $s, t$  o tra tutte le coppie di nodi del grafo.

Due tipologie particolari di percorsi sono i **percorsi semplici** (cioè, percorsi che non attraversano mai un arco più di una volta) e i **percorsi elementari** (cioè, percorsi che non visitano mai alcun nodo più di una volta).

Qualsiasi percorso elementare è anche semplice ma non il contrario. Si noti che esiste un numero finito di cammini elementari e semplici, e quindi esiste sempre un cammino elementare (semplice) ottimo. Tuttavia, esiste un numero infinito (a meno che  $G$  non sia aciclico) di percorsi generali e non è garantita l'esistenza di un percorso generale ottimale.

Ricapitolando:

- semplice  $\mapsto$  non ripete un arco;
- elementare  $\mapsto$  non ripete nodi;
- se non semplice  $\mapsto$  allora non elementare.

## 9.1 Cammino minimo – Shortest path

Possiamo limitare la nostra attenzione ai percorsi elementari, perché se esiste un percorso più breve, allora esiste anche un percorso più breve elementare.

Infatti, supponiamo che un percorso  $s \rightarrow t$   $P$  più breve includa un ciclo  $C$ . Allora deve essere  $c(C) = 0$ , poiché se  $c(C) > 0$  non avremmo attraversato il ciclo in un percorso ottimale. Quindi rimuoviamo  $C$  da  $P$  ottenendo un percorso dallo stesso costo  $P'$  ancora ottimale. Ripetiamo finché non otteniamo finalmente un percorso elementare ottimale. Si noti che non può essere  $c(C) < 0$ , poiché se potessimo attraversare un ciclo di peso negativo, continueremmo a percorrerlo indefinitamente, rendendo così il percorso sempre più breve. Allora il problema sarebbe illimitato e non esisterebbe un percorso più breve.

Definiamo il problema del cammino minimo come segue:

**PROBLEMA:** Shortest Path Problem (SPP)

**INPUT:** Un grafo diretto  $G = (V, A)$  con pesi  $c(i, j) \in \mathbb{R}$  per  $(i, j) \in A$ . Due nodi  $s, t \in V$ .

**OBIETTIVO:** Trovare un cammino elementare  $P$  da  $s$  a  $t$  t.c.  $c(P)$  è più piccolo possibile.

La possibile presenza di un ciclo di lunghezza negativa, che può avvenire se alcuni archi hanno peso negativo, è un fattore che complica il calcolo del cammino minimo in un grafo; e comporta che il problema diventi NP-HARD.

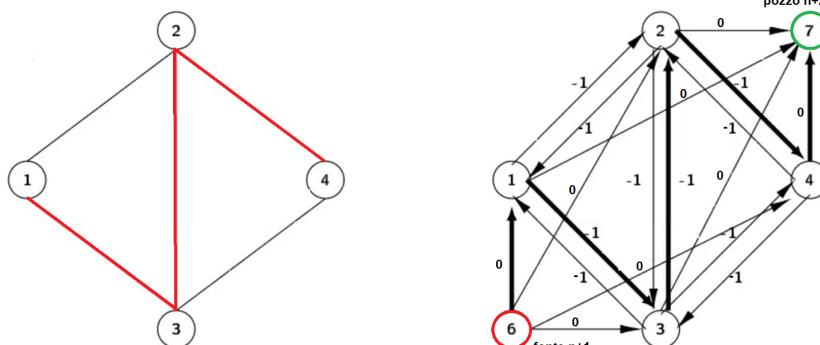
**Teorema:** Il problema del cammino minimo (SPP) è NP-hard.

**Dimostrazione:** Per dimostrare che il problema è NP-HARD, riduciamo il problema del cammino Hamiltoniano (NP-completo sui grafi con cicli) al problema del cammino minimo. Sia  $\hat{G} = ([n], E)$  una istanza del problema del cammino Hamiltoniano. Definiamo una istanza  $G = (V, A)$  dell'SSP come segue:

$$\begin{aligned} V &:= [n] \cup \{n+1, n+2\} \\ A &:= \{(i, j), (j, i) : ij \in E\} \cup \{(n+1, i), (i, n+2) : i \in [n]\} \\ c(h, k) &= -1 \text{ per ogni } (h, k) \in A \text{ con } h, k \in [n] \\ c(n+1, i) &= c(i, n+2) = 0 \text{ per ogni } i \in [n] \end{aligned}$$

Quindi, aggiungo 2 nodi fintizzi  $n+1$  e  $n+2$ , sdoppio gli archi non orientati in orientati e aggiungo archi dal nodo fintizzo  $n+1$  (fonte) ad ogni nodo  $i$  e archi per ogni nodo  $i$  al nodo  $n+2$  (pozzo). Tutti gli archi originali (frutto dello sdoppiamento) hanno costo  $-1$  e tutti gli archi in uscita dalla fonte e in entrata al pozzo gli do costo 0. Mi chiedo ora quale sia il cammino di costo minimo da  $s$  a  $t$  con  $s = n+1, t = n+2$ . Si noti che  $G$  contiene cicli negativi (uno per ogni arco e ogni ciclo di  $\hat{G}$ ). Segue un esempio.

Ciascun cammino elementare  $s \rightarrow t$  di lunghezza  $-k$  in  $G$  definisce un cammino elementare di lunghezza  $k$  in  $\hat{G}$  e viceversa. Perciò,  $\exists$  un cammino Hamiltoniano in  $\hat{G}$  sse il cammino minimo  $s \rightarrow t$  ha lunghezza  $-(n-1)$ .



(a)

(b)

Figura (a): il grafo  $\hat{G}$  con **cammino hamiltoniano**;

Figura (b): il grafo  $G$  con **cammino minimo** di valore  $-3$ .

Quando i pesi possono essere sia negativi che positivi, possiamo modellare il problema di trovare un percorso di peso massimo (solitamente chiamato anche percorso più lungo) come percorso più breve, cambiando il segno di ciascun peso dell'arco. Dalla suddetta riduzione segue che trovare il percorso elementare più lungo tra due nodi in un grafo è NP-HARD.

Vedremo che la presenza di cicli di peso negativo rende NP-HARD il problema del percorso più breve. Quando non ci sono tali cicli, il cammino elementare più breve può essere trovato in tempo polinomiale. Quindi distinguiamo alcune versioni dell'SPP senza cicli negativi nel grafico di input. Uno di questi è quando tutti i pesi non sono negativi. Un altro, è quando il grafico è aciclico, ecc.

## 9.2 Modelli di programmazione lineare intera

Si può notare che nessun cammino elementare da  $s$  a  $t$  può usare archi in entrata su  $s$ , neppure archi in uscita da  $t$ . Si assume, probabilmente dopo aver rimosso qualche arco, che  $\delta^-(s) = \delta^+(t) = \emptyset$ .

Un cammino è una sequenza di vertici, ma è chiaro che un cammino elementare è anche unicamente identificato dai suoi archi. Perciò, associamo variabili binarie  $x_{ij}$  agli archi  $(i,j) \in A$ , e le usiamo per selezionare gli archi del cammino.

Il cammino identifica un sottografo in cui:

1. il nodo  $s$  ha grado in uscita = 1;
2. il nodo  $t$  ha grado in entrata = 1;
3. ciascun nodo interno al cammino ha sia grado in entrata che uscita = 1;
4. tutti i nodi rimanenti hanno entrambi i gradi in uscita e entrata = 0.

Si considerino i seguenti vincoli:

$$\sum_{(s,j) \in \delta^+(s)} x_{sj} = 1 \quad (1)$$

$$\sum_{(j,t) \in \delta^-(t)} x_{jt} = 1 \quad (2)$$

$$\sum_{(v,j) \in \delta^+(v)} x_{vj} - \sum_{(i,v) \in \delta^-(v)} x_{iv} = 0 \quad \forall v \in V - \{s, t\} \quad (3)$$

$$\sum_{(i,j) \in A : i,j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subseteq V - \{s, t\} \quad (4)$$

Ovvvero:

1. un arco solo esce da  $s$ ;
2. un arco solo entra in  $t$ ;
3. la somma degli archi che escono da un certo nodo è uguale alla somma degli archi che entrano in quel nodo (esclusi  $s$  e  $t$ );
4. per ogni sottoinsieme di nodi che non contenga  $s$  e  $t$ , la somma di quei archi deve essere  $\leq$  alla cardinalità di  $S$  -1.

Per ogni soluzione fattibile  $x$ , sia  $X := \{(i, j) \in A : x_{ij} = 1\}$  (ovvero gli archi scelti). Diciamo che  $X$  è l'insieme degli archi di un cammino elementare  $s \rightarrow t$ .

Per dimostrarlo, usiamo il seguente risultato:

**Lemma:** Sia  $G' = (N, E)$  un grafo orientato t.c.  $d^-(v) = d^+(v)$  per ogni  $v \in N$ . Allora  $E$  è l'unione di  $k \geq 0$  cicli elementari di archi disgiunti.

**Dimostrazione:** Per induzione su  $|E|$ . Se  $|E| = 0$  allora  $E$  è l'unione di 0 cicli elementari. Si assuma vero (che si può partizionare) per  $|E| = 0, \dots, m-1$ . Si assuma vero per  $|E| = 0, \dots, m-1$ .

Dato  $G'$  con  $|E| = m$ , si prenda qualsiasi nodo  $v_0$  con  $d^+(v_0) > 0$  e si visiti il grafo, passando attraverso i nodi  $v_0, v_1, v_2, \dots$ . Ogni volta che entriamo in un nodo  $v_i$  usciamo con l'arco  $(v_i, v_{i+1})$ , dato che  $d^-(v_i) > 0$  implica  $d^+(v_i) > 0$ . Dal principio della piccionaia, dopo almeno  $n$  step, la lista  $v_0, \dots, v_n$  deve contenere qualche nodo ripetuto.

Sia  $I$  t.c.  $v_I = v_{l+t}$  ma  $v_I \neq v_{l+i}$  per  $i = 1, \dots, t-1$ . Allora  $C = (v_l, v_{l+1}, \dots, v_{l+t} = v_l)$  è un ciclo elementare  $C$ . Rimuovendo  $C$  da  $E$  otteniamo un grafo con  $m-t$  archi e  $d^-(i) = d^+(i)$  per ogni  $i \in N$ , che, per induzione, scomponiamo in un insieme di cicli. Questo insieme, insieme a  $C$ , fornisce la decomposizione di  $G'$ .

Si richiama che, per ogni soluzione binaria  $x$  a (1), (2), (3), (4), impostiamo  $X := \{(i, j) \in A : x_{ij} = 1\}$ . Dichiariamo che  $X$  è l'insieme di archi di un cammino elementare  $s \rightarrow t$ .

**Dimostrazione:** Si definisca  $G' = (V, X \cup \{(t, s)\})$ . In  $G'$  abbiamo che  $d^-(i) = d^+(i)$  per ogni  $i \in V$ . Perciò, dal lemma,  $X \cup \{(t, s)\}$  può essere decomposto in cicli elementari. Sia  $C$  il ciclo contenente l'arco  $(t, s)$ . Allora  $C \setminus (t, s)$  è un cammino elementare  $s \rightarrow t$  in  $X$ . Si assuma ora che non ci siano altri cicli  $C' \subseteq X$  nella decomposizione, detta  $C' =$

$(v_1, \dots, v_k, v_1)$ . Ma allora la diseguaglianza (9) corrispondente a  $S := \{v_1, \dots, v_k\}$  verrebbe violata da  $x$ , una contraddizione. Perciò,  $x$  consiste solo del cammino elementare  $s \rightarrow t$ .

Dopo aver aggiunto la funzione obiettivo, otteniamo il modello finale per il problema NP-HARD del cammino minimo (Shortest Path Problem – SPP):

$$\min \sum_{(i,j) \in A} c(i,j)x_{ij} \quad (5)$$

$$\sum_{(s,j) \in \delta^+(s)} x_{sj} = 1 \quad (6)$$

$$-\sum_{(j,t) \in \delta^-(t)} x_{jt} = -1 \quad (7)$$

$$\sum_{(v,j) \in \delta^+(v)} x_{vj} - \sum_{(i,v) \in \delta^-(v)} x_{iv} = 0 \quad \forall v \in V - \{s, t\} \quad (8)$$

$$\sum_{(i,j) \in A: i,j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subseteq V - \{s, t\} \quad (9)$$

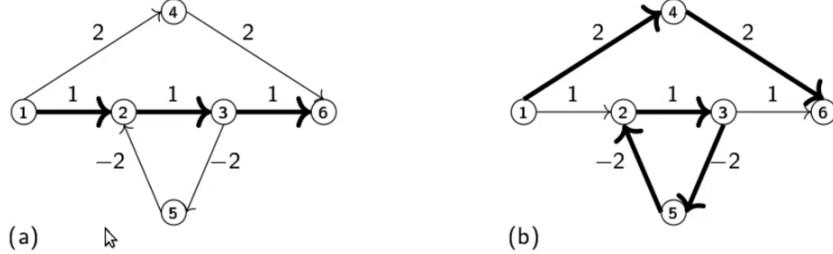
con variabili  $x_{ij} \in \{0, 1\}$  per ogni  $(i, j) \in A$ .

I vincoli (9):

$$\sum_{(i,j) \in A: i,j \in S} x_{ij} \leq |S| - 1 \quad \forall S \subseteq V - \{s, t\}$$

chiamati **”subtour-elimination constraints”**, sono necessari per evitare soluzioni in cui l’insieme di archi selezionati sono un cammino elementare  $s \rightarrow t$   $P$  più un ciclo di archi disgiunti  $C_1, \dots, C_k$  ricoprente altri nodi. In questo caso non sarebbe garantito che  $P$  sia un shortest path.

Si consideri la seguente situazione esplicativa in figura:



Per  $s = 1, t = 6$ , esistono solo due cammini elementari, cioè  $(1,2,3,6)$  di valore 3 e  $(1,4,6)$  di valore 4. In figura (a) raffiguriamo la soluzione ottimale (valore = 3, archi t.c.  $x_{ij} = 1$  sono in grassetto) del modello con i vincoli di eliminazione di subtour (che includono i vincoli  $x_{23} + x_{35} + x_{52} \leq 2$ ). In figura (b) il modello senza i vincoli di subtour, che prende anche il ciclo negativo per “guadagnare” -3. Ovviamamente non è corretto.

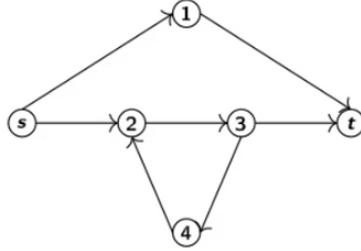
Il modello SPP senza le eliminazioni di subtour possiede la seguente proprietà positiva: la matrice dei vincoli è totalmente unimodulare (TUM).

Perciò, sia  $M$ , di dimensione  $n \times m$ , la **matrice di incidenza nodi-archi**. La  $i$ -esima riga  $m$  ha un +1 per ogni colonna  $(i, j) \in \delta^+(i)$  e un -1 per ogni colonna  $(j, i) \in \delta^-(i)$ . Moltiplicando la riga  $M_i$  per  $x$  otteniamo:

$$\sum_{(i,j) \in \delta^+(i)} x_{ij} - \sum_{(j,i) \in \delta^-(i)} x_{ji}$$

cioè, la differenza tra il numero di archi in entrata e uscita.

**Esempio:** esempio esplicativo della matrice TUM:



$$\begin{array}{ccccccccc}
 & s1 & s2 & 1t & 23 & 34 & 3t & 42 \\
 c & \begin{pmatrix} +1 & +1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 \\ -1 & 0 & +1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & +1 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & +1 & +1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & +1 \end{pmatrix} & \begin{pmatrix} x_{s1} \\ x_{s2} \\ x_{1t} \\ x_{23} \\ x_{34} \\ x_{3t} \\ x_{42} \end{pmatrix} & = & \begin{pmatrix} 1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\
 t & & & & & & & \\
 1 & & & & & & & \\
 2 & & & & & & & \\
 3 & & & & & & & \\
 4 & & & & & & & 
 \end{array}$$

$$\begin{aligned}
 x_{s1} + x_{s2} &= 1 \\
 -x_{1t} - x_{3t} &= -1 \\
 -x_{s1} + x_{1t} &= 0 \\
 -x_{s2} + x_{23} - x_{42} &= 0 \\
 -x_{23} + x_{34} + x_{3t} &= 0 \\
 -x_{34} + x_{42} &= 0
 \end{aligned}$$

Si definisca un vettore  $b$  di dimensione  $n \times 1$  t.c.  $b_s := -1, b_t := 1$  e  $b_i := 0$  per ogni  $i \neq s, t$  allora il modello senza eliminazioni di subtour, in forma matriciale è:

$$\min c^T x$$

t.c.

$$\begin{aligned}
 Mx &= b \\
 x &\in \{0, 1\}^m
 \end{aligned}$$

Sappiamo dalla teoria della programmazione intera che la matrice di incidenza di un grafo è una matrice TUM, quindi  $M$  è TUM  $\implies$  il problema ILP (senza eliminazioni di subtour) è naturalmente intero, e può essere risolto come un problema LP.

Questo prova che trovare il più corto cammino elementare sulla classe di tutti i grafi pesati senza cicli negativi è un problema sempre risolvibile in tempo polinomiale.

Si denota con **NNC** la classe di tutti i **grafi diretti pesati senza cicli negativi**. Possiamo suddividere e analizzare successivamente 3 sottoclassi della classe NNC:

Un grafo può essere NNC se:

1. tutti gli archi sono non-negativi, allora è NNC;
2. aciclici (indipendentemente dal fatto che abbiano o meno archi di peso negativo), allora è NCC;
3. con archi di peso negativo ma nessun ciclo negativo è NCC.

I problemi NNC si possono quindi risolvere con la programmazione lineare, però, esistono algoritmi combinatori alternativi specifici per i vari casi, spesso più performanti.

### 9.3 Algoritmo di Dijkstra – NNC con pesi non negativi (1)

Supponiamo che tutti i pesi degli archi non siano negativi. Quindi, non ci sono cicli di peso negativo in  $G$  e quindi i vincoli di eliminazione di subtour non sono necessari.

Poiché il modello senza eliminazione di subtour è naturalmente intero, possiamo trovare il percorso elementare più breve su pesi non negativi in tempo polinomiale tramite la soluzione di un LP.

Esistono, tuttavia, algoritmi combinatori più efficaci rispetto alla risoluzione di un LP, come l'approccio popolare noto come algoritmo del percorso più breve di Dijkstra.

L'algoritmo può essere visto come la computazione del cammino più breve da  $s$  alla sequenza di nodi  $v_0 = s, v_1, v_2, \dots$ , progressivamente più lontano da  $s$ , che finisce non appena  $v_i = t$ .

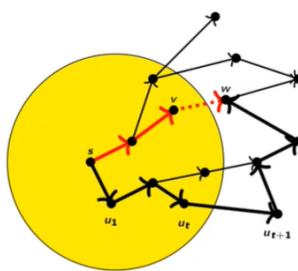
Il più corto cammino  $s \rightarrow s$  ha lunghezza 0 e consiste di 0 archi. Ciascun cammino più corto  $s \rightarrow v_i$  viene poi calcolato sfruttando la seguente proprietà:

**Lemma:** Sia  $\mathcal{T} = \{v_0, \dots, v_{i-1}\}$  un insieme di vertici tali che per ogni  $i \in \mathcal{T}$  un cammino minimo  $P^i$ , di lunghezza  $L[i]$  è noto. Sia  $(v, w) \in \delta^+(\mathcal{T})$  t.c.

$$L[v] + c(v, w) = \min \{ L[i] + c(i, j) : (i, j) \in \delta^+(\mathcal{T}) \}$$

Allora  $P^w := P^v \cup (v, w)$  è un cammino minimo  $s - w$ , di lunghezza  $L[w] := L[v] + c(v, w)$

**Dimostrazione:** Sia  $P = (s, u_1, \dots, u_k = w)$  ogni cammino  $s - w$ , e sia  $u_t$  l'ultimo nodo di  $\mathcal{T}$  attraversato da  $P$ . Si separi  $P$  in  $P' = (s, \dots, u_t)$  seguito da  $(u_t, u_{t+1}) \in \delta^+(\mathcal{T})$  e da  $P'' = (u_{t+1}, \dots, w)$ . Si noti che  $c(P'') \geq 0$ .



$$\begin{aligned}
c(P) &= c(\textcolor{red}{P'}) + c(u_t, u_{t+1}) + c(\textcolor{blue}{P''}) \\
&\geq L[u_t] + c(u_t, u_{t+1}) + c(\textcolor{blue}{P''}) \\
&\geq L[v] + c(v, w) + c(\textcolor{blue}{P''}) \\
&\geq L[v] + c(v, w) \\
&= c(P^w)
\end{aligned}$$

### Implementazione algoritmo di Dijkstra

Descriviamo una semplice implementazione dell'algoritmo di Dijkstra, che gira in tempo  $\Theta(n^2)$ . Assumiamo che l'input sia un grafo completo, dato dalla matrice  $n \times n$ ,  $c \geq 0$  t.c.  $c_{ij}$  sia il costo dell'arco  $(i, j)$  per  $i \neq j$ , mentre  $c_{i,i} = 0$  per ogni  $i$ . Sotto l'assunzione, l'algoritmo è ottimale (lineare sull'input).

Per risolvere SPP in un grafo  $G$  che non è completo, dovremmo aggiungere gli archi mancanti con costo  $c(i, j) := M$  ciascuno, dove  $M$  è un upper bound al costo del cammino minimo elementare  $s \rightarrow t$  (così, la soluzione ottimale non può usare nessuno di questi archi "dummy").

Se  $G$  fosse denso, allora la sua dimensione sarebbe già  $\Theta(n^2)$  e quindi questo processamento non peggiorerebbe l'efficacia dell'algoritmo, che rimane ottimale.

Inoltre, se  $G$  fosse sparso, allora sono possibili implementazioni migliori dell'algoritmo di Dijkstra rispetto a quella fornita.

L'implementazione (simile a quella di Prim per il problema del MST) fa uso dei seguenti array:

- $\text{flag}[\cdot]$ :  $\text{flag}[i]$  booleano (0/1) che è vero sse  $i$  è in  $\mathcal{T}$ ;
- $L[\cdot]$  : per ogni vertice  $i \notin \mathcal{T}$  il valore  $L[i]$  rappresenta la lunghezza del migliore cammino  $s \rightarrow t$  tra quei nodi (ad eccezione di  $i$ ) che sono in  $\mathcal{T}$ . Se  $i \in \mathcal{E}$  allora  $L[i]$  è la lunghezza del più corto cammino  $s \rightarrow i$ .
- $P[\cdot]$  : per ogni vertice  $i \notin \mathcal{T}$  l'etichetta  $P[i]$  denota un vertice  $j \in \mathcal{T}$  t.c.  $L[i] = L[j] + c(j, i)$ . Per ogni vertice  $i \in \mathcal{T}$ ,  $P[i]$  denota il predecessore di  $i$  nel cammino più corto. L'unione di tutti i cammini è un albero con radice in  $s$  chiamato l'albero dei cammini più corti (shortest path tree).  $P[i]$  è il padre di  $i$  in un albero. Per convenzione il padre della radice è la radice stessa.

```

1. for  $k = 1, \dots, n$  do
2.    $L[k] := c(s, k); P[k] := s; flag[k] := 0;$ 
3. end for
4.  $flag[s] := 1; /* inserts the root s in T */$ 
5. while  $flag[t] = 0$  do
6.    $val := +\infty;$ 
7.   for  $k = 1, \dots, n$  do
8.     if ( $flag[k] = 0$ )  $\wedge (L[k] < val)$  then
9.        $val := L[k]; w := k;$ 
10.      end if
11.    end for
12.    $flag[w] := 1; /* add w to T */$ 
13.   for  $k = 1, \dots, n$  do
14.     if ( $flag[k] = 0$ )  $\wedge (L[k] > L[w] + c(w, k))$  then
15.        $L[k] := L[w] + c(w, k); P[k] := w;$ 
16.     end if
17.   end for
18. end while

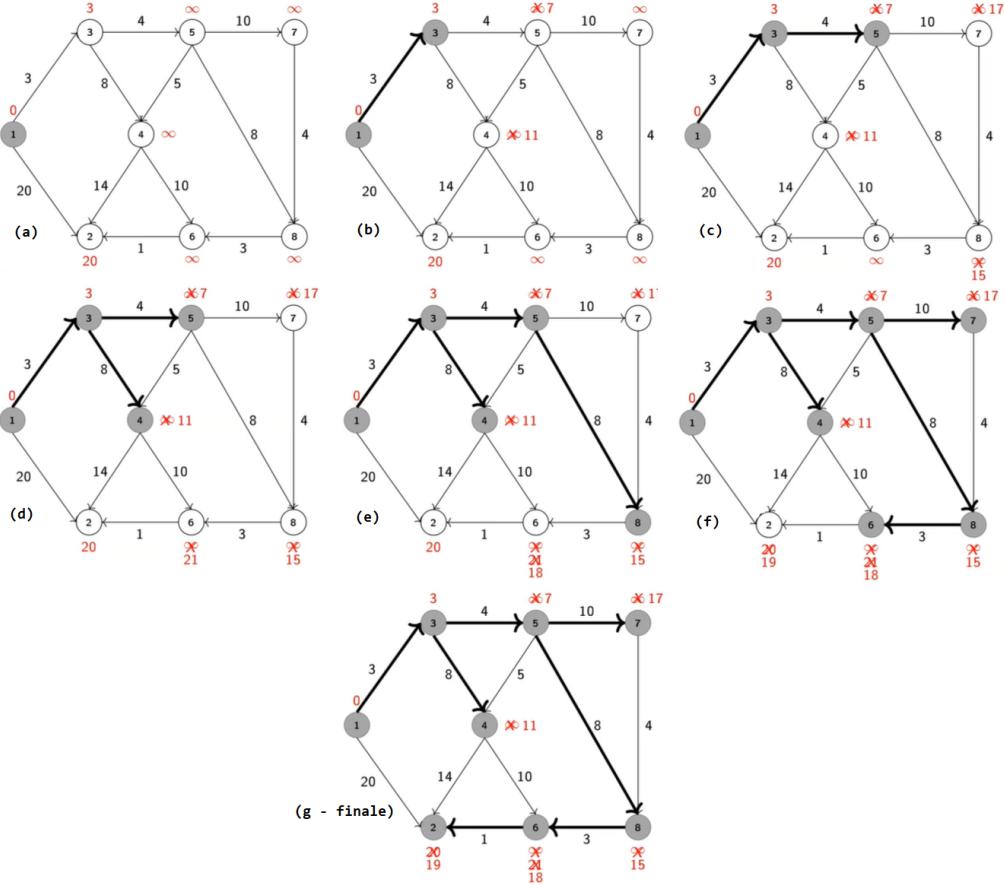
```

### Dijkstra implementato con le heap

Si assuma che tutti i vertici siano messi in una min-heap, ordinati secondo i valori di  $L[ ]$ .

- Per ogni nodo  $i$  teniamo traccia della sua posizione nella heap, così da poter accedere all'elemento della heap in tempo  $O(1)$ ;
- Costruire la heap costa  $O(n)$ ;
- Ottenerne il vertice  $w$  con minor valore  $L[ ]$  può essere fatto in tempo  $O(\log n)$  (cioè,  $O(1)$  per prendere  $w$  più  $O(\log n)$  per ripristinare la proprietà della heap);
- Dopo che è stato determinato  $w$ , in tempo  $\Theta(d(w))$ , usando la lista di stelle, accediamo a tutti i suoi vicini nella heap, e, per ognuno di essi il cui  $L$  valore è cambiato, aggiorniamo la sua posizione nella heap in tempo  $O(\log n)$ . Questo step ha un costo  $O(d(w) \log n)$ .
- Considerando tutto insieme,  $w$  porta a una complessità  $O(m \log n)$  per questa implementazione dell'algoritmo di Dijkstra. Se il grafo è sparso, questa complessità batte  $O(n^2)$ , ma se il grado è denso,  $O(n^2) = O(m)$  batte  $O(m \log n)$ .

**Esempio di esecuzione:** Si ha ( $s = 1, t = 2$ ):



### Minimizzare il numero di archi:

Dato  $G = (V, A)$  e  $s, t \in V$  vogliamo trovare un cammino  $s \rightarrow t$  con numero minore di archi.

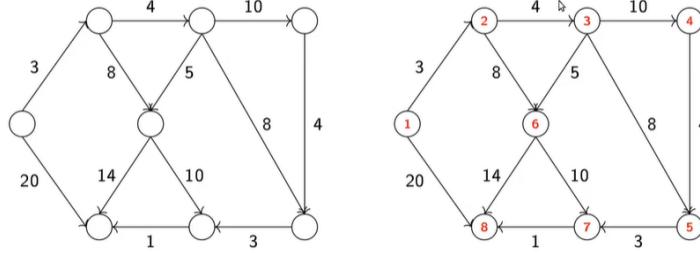
Potremmo dare peso  $c(i, j) = 1$  per ogni arco  $(i, j) \in A$  e usare l'algoritmo di Dijkstra, ma esiste un metodo migliore.

Facciamo una visita BFS (cioè utilizzando una coda) di  $G$  a partire da  $s$ . I primi nodi che entrano in coda sono quelli a distanza 1 da  $s$ . Dopo che tutti questi nodi sono stati visitati, si inseriscono in coda i nodi raggiungibili da 2 archi ma non da 1 (a distanza 2). Li visitiamo tutti, spingendo nuovi nodi raggiungibili di 3 archi ma non di 2 né di 1 (distanza 3), ecc.

Ogni volta che la coda contiene un blocco di nodi di distanza  $k$  seguito da nodi di distanza  $k+1$ , e queste sono sempre le distanze minime possibili. Pertanto, possiamo trovare i percorsi più brevi per tutti i nodi nel tempo  $O(m + n)$ , che è  $O(m)$  sui grafici connessi.

## 9.4 NNC: caso grafi aciclici (2)

**DAG e ordinamento topologico:** In un grafo diretto aciclico (DAG), non essendoci cicli, e quindi neanche cicli di lunghezza negativa, un DAG ammette un ordinamento topologico, quindi assumiamo che il grafo in input  $G = ([n], A)$  sia già ordinato topologicamente, cioè  $i < j$  per ogni  $(i, j) \in A$ :



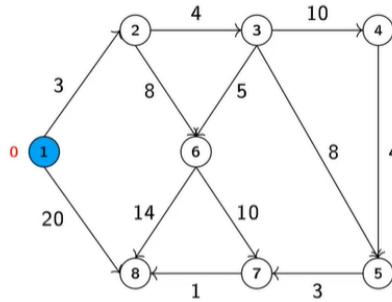
Usiamo un array  $\text{len}[i]$  per memorizzare la lunghezza di un cammino minimo da una fonte  $s$  verso qualsiasi  $i \in [n]$ . Riempiamo le entries di  $\text{len}[\cdot]$  da sinistra a destra. Chiaramente possiamo inizializzare  $\text{len}[i] := +\infty$  per ogni  $i = 1, \dots, s - 1$ , dato che nessun nodo è raggiungibile da  $s$  (tutte le etichette stanno aumentando lungo qualsiasi percorso in  $G$ ). Perciò, inizializziamo  $\text{len}[s] := 0$  per  $i = s + 1, \dots, t$ , ogni volta stiamo calcolando  $\text{len}[i]$ , i valori  $\text{len}[i]$  sono già conosciuti per tutti gli archi  $(j, i) \in \delta^-(i)$ . Dato che un cammino minimo da  $i$  deve usare uno di questi archi come arco finale, la lunghezza del cammino minimo  $s \rightarrow i$  è:

$$\text{len}[i] := \min_{(j,i) \in \delta^-(i)} \text{len}[j] + c(j, i)$$

Calcolare  $\text{len}[i]$  richiede tempo proporzionale a  $d^-(i)$ . Il ciclo for per  $i = s, \dots, t$  richiede almeno tempo  $\sum d^-(i) = O(m)$ . L'intero algoritmo richiede tempo  $O(n + m)$ .

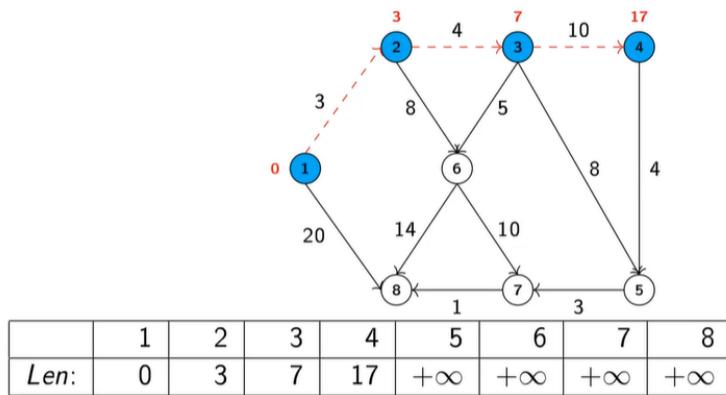
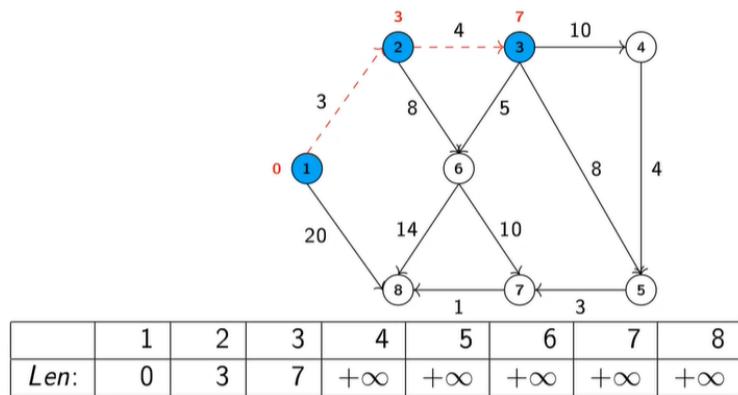
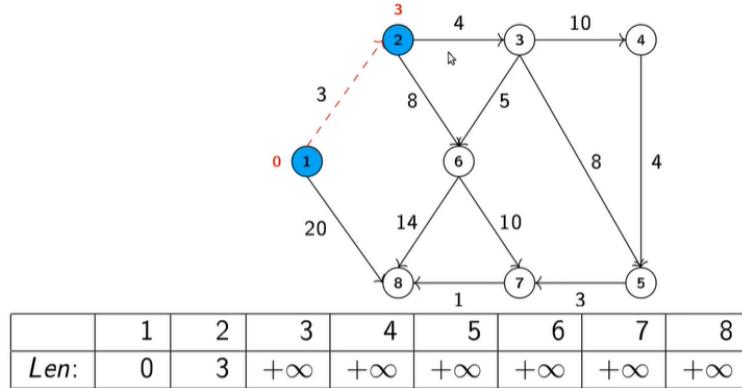
**Esempio:** Sia  $S = 1$  e  $t = 8$ :

Passo 1:

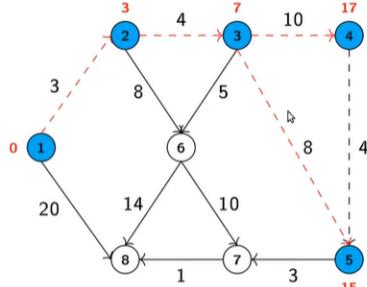


	1	2	3	4	5	6	7	8
Len:	0	$+\infty$						

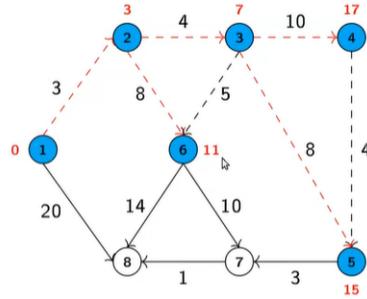
Passo 2,3,4:



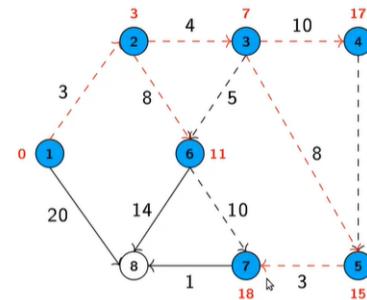
Passo 5,6,7:



	1	2	3	4	5	6	7	8
Len:	0	3	7	17	15	$+\infty$	$+\infty$	$+\infty$

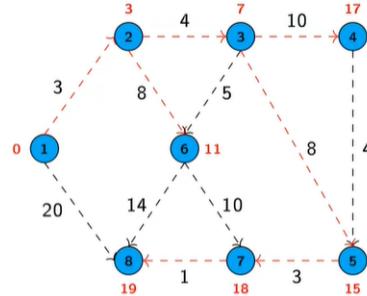


	1	2	3	4	5	6	7	8
Len:	0	3	7	17	15	11	$+\infty$	$+\infty$



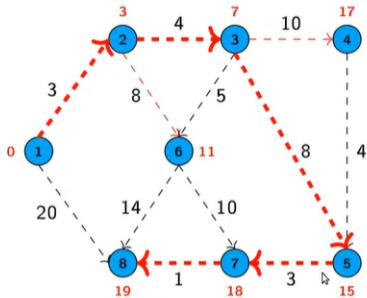
	1	2	3	4	5	6	7	8
Len:	0	3	7	17	15	11	18	$+\infty$

Passo 8:



	1	2	3	4	5	6	7	8
Len:	0	3	7	17	15	11	18	19

Passo finale: ripercorro all'indietro partendo dal nodo destinazione, guardando i nodi padre:



	1	2	3	4	5	6	7	8
Len:	0	3	7	17	15	11	18	19

## Il CPM (Critical Path Method)

Alcuni problemi di schedulazione hanno vincoli di precedenza tra i task  $J$ . Le precedenze definisco un digrafo (diretto = digrafo) aciclico  $G = (J, P)$ . Ciascun task  $j \in J$  ha tempo di processamento  $p(j)$

Uno piano, programma, scheduling ("schedule") è una sequenza di tempi di partenza  $t_j$  per  $j \in J$ , che deve soddisfare:

$$t_j \geq t_i + p_i \quad \forall (i, j) \in P$$

Possiamo aggiungere task "dummy" 0 e  $n+1$ , di tempo  $p(0) = p(n+1) = 0$  e  $\text{arcs}(0, i), (i, n+1)$  per ogni  $i \in J$ . Allora 0 è il primo task che viene schedulato ( $t_0 := 0$ ) e  $n+1$  è l'ultimo. Chiamiamo  $t_{n+1}$  il makespan ( $t_{\text{end}} - t_{\text{start}}$ ) che vogliamo minimizzare.

Pesiamo gli archi di  $P$ . Impostiamo  $w(i, j) := p(i)$  per ogni  $i$  e  $(i, j) \in \delta^+(i)$ . Si definisca  $ET[i] =$  il tempo di partenza più "presto" (earliest starting time) di  $i$ . Allora il minimo makespan è  $ET[n+1]$ . Ogni cammino  $0 \rightarrow i$  deve essere completato prima che  $i$  inizi, perciò  $ET[i] =$  è la lunghezza del cammino più lungo  $s \rightarrow i$ .

Possiamo calcolare  $ET[i]$  per un  $G$  ordinato topologicamente in tempo  $O(m)$  come:

$$ET[i] := \max_{(j, i) \in \delta^-(i)} ET[j] + p(j)$$

Sia  $M := ET[n+1]$  il makespan. If we use the schedule  $t_i := ET[i]$  for all  $i \in J$  we are sure we can achieve the makespan, but some jobs could be delayed with no harm while others are more critical. Let us define  $LT[i]$  to be the latest starting time, meaning that if a job starts after  $LT[i]$  then the makespan will be  $> M$ . Each job has a "slack interval" in which it could be started, i.e.  $[ET(i), LT(i)]$ . If  $ET[i] = LT[i]$  the job must start at time  $ET[i]$  or the makespan worsens. We call each such job a critical job. Every longest path from 0 to  $n+1$  is called a critical path. The critical path visits only critical jobs. Delaying any of them would worsen the makespan.

We can compute  $LT[i]$  similarly to  $ET[]$  but backwards. Set  $LT[n+1] = M$  and then, for  $i < n$

$$LT[i] := \min_{(i, j) \in \delta^+(i)} LT[j] - p(i)$$

If we denote by  $\lambda(i, j)$  the length of a longest  $i \rightarrow j$  path then we have, for each critical job  $i$

$$M = \lambda(0, i) + \lambda(i, n+1)$$

- Furthermore  $LT[i] = M - (\lambda(i, n+1) - p(i))$ . The Critical Path Method is the analysis of the jobs to identify critical jobs and devote resources so that they are not delayed

## 9.5 Flloyd-Warshall – NNC con archi negativi ma nessun ciclo negativo (3)

Quando  $G$  (con archi di lunghezza sia negativa che positiva) non ha un ciclo di lunghezza negativa, esiste un algoritmo efficace per calcolare i cammini più brevi tra tutte le coppie  $s, t$  di nodi.

L'algoritmo, conosciuto con il nome Flloyd-Warshall, richiede tempo  $O(n^3)$  ed è una generalizzazione dell'algoritmo per calcolare la chiusura transitiva di un grafo. Possiamo calcolare il percorso più breve tra due nodi  $i$  e  $j$  sfruttando la seguente proprietà ricorsiva, per  $k = 1, \dots, n$ :

Il percorso più breve tra  $i$  e  $j$ , che può utilizzare solo nodi  $\leq k$  come nodi intermedi, è il più breve tra:

1. il cammino più breve da  $i$  a  $j$  che usa solo nodi  $\leq k - 1$  come nodi intermedi, e
  2. il cammino  $P = P' \cdot P''$  composto dal cammino più breve  $P'$  da  $i$  a  $k$  seguito dal cammino più breve  $P''$  da  $k$  a  $j$ , dove  $P'$  e  $P''$  usano solo nodi  $\leq k - 1$  come nodi intermedi.
- 

Quando  $k = n$ , il cammino più breve tra  $i$  e  $j$  usando solo nodi  $\leq k$  come nodi intermedi è quindi il cammino più breve possibile.

Impostiamo un algoritmo che itera su tutti  $k = 1, \dots, n$ . Usiamo  $n + 1$  matrici quadrate  $C^k$ , per  $k \geq 0$  dove  $C^k[i, j]$  è la lunghezza del più corto cammino  $i - j$  che usa solo nodi in  $\{0, \dots, k\}$  come intermedi. Memorizziamo in  $p[i, j]$  il predecessore di ciascun nodo  $j$  nel cammino più breve tra  $i$  e  $j$ . L'inizializzazione è la seguente:

```
for  $i \in [n]$  for  $j \in [n]$  do
    if  $(i, j) \in A$  then  $C^0[i, j] := c(i, j); p[i, j] := i$ 
    else  $C^0[i, j] := +\infty; p[i, j] := -1$ 
```

Calcoliamo poi  $C^k$  da  $C^{k-1}$ , per  $k = 1, \dots, n$ , come:

```
for  $i \in [n]$  for  $j \in [n]$  do
    if  $C^{k-1}[i, k] + C^{k-1}[k, j] < C^{k-1}[i, j]$  then
         $C^k[i, j] := C^{k-1}[i, k] + C^{k-1}[k, j];$ 
         $p[i, j] := p[k, j]$ 
    else
         $C^k[i, j] := C^{k-1}[i, j]$ 
    end if
end for
```

Se non ci sono cicli di lunghezza negativa, la correttezza dell'algoritmo può essere dimostrata per induzione su  $k$ , cioè, se i valori in  $C^k$  sono corretti, allora i valori in  $C^{k+1}$  sono anche corretti.

Con cicli negativi, l'esistenza di percorsi più brevi non è garantita. Sebbene sappiamo che esisteranno percorsi semplici più brevi, l'algoritmo non sarebbe in grado di trovarli. Ad esempio, si consideri un grafo che è un ciclo di 3 nodi, con  $c(1, 2) = c(2, 3) = 1$  e  $c(3, 1) = -3$ . Quindi, nella avremmo  $C^3[1, 2] = C^2[1, 3] + C^2[3, 2] = 2 - 2 = 0$ , ma non esiste un percorso semplice di lunghezza 0 da 1 a 2 in  $G$ .

Possiamo verificare l'esistenza di cicli negativi in  $G$  osservando la diagonale in  $C^n$ . Se per ogni  $i$  si ha  $C^n[i, i] < 0$  allora esiste un ciclo di lunghezza negativa che passa attraverso  $i$ . In questo caso non è garantito che le entries rimanenti in  $C^n$  rappresentino i valori corretti dei percorsi più brevi tra le coppie di nodi.

Se usiamo matrici distinte  $C^k$ , la memoria è  $\Theta(n^3)$ . Osserviamo come sarebbero sufficienti due matrici ( $C^k$  dipende solo da  $C^{k-1}$  e quindi  $C^{k-2}$  non è più necessario quando stiamo calcolando  $C^k$ ). Potremmo usare solo  $C^0$  e  $C^1$ , sovrascrivendo  $C^k$  su  $C^0$  per  $k$  pari, e su  $C^1$  per  $k$  dispari. In questo modo la memoria richiesta dall'algoritmo è  $\Theta(n^2)$ . Ulteriori riflessioni dovrebbero convincerci che una matrice unica è sufficiente, dal momento che possiamo sovrascrivere  $C^{k+1}$  sopra  $C^k$ .

Quando l'input è un DAG, il calcolo dei percorsi più brevi di tutte le coppie richiederebbe  $O(n^3)$  da Floyd-Warshall e  $O(nm)$  da  $n$  applicazioni dell'algoritmo DAG. Se il grafico è denso le due complessità sono equivalenti, ma per un grafo sparso è meglio utilizzare l'algoritmo DAG.

### Recap dei 3 casi:

- NNC con pesi non negativi: Algoritmo di Dijkstra;
- NNC senza cicli (DAG): Ordinamento topologico;
- NNC con archi negativi (ma nessun ciclo negativo): Algoritmo di Floyd-Warshall.

## 10 Reti di flusso

Tipicamente, una merce (commodity) viaggia da produttori a consumatori attraverso una rete, cioè, un grafo diretto che connette i produttori e i consumatori attraverso dei nodi intermedi. Ciascun arco del grafo ha una certa capacità associata (massima quantità di quella merce che può passare attraverso un particolare arco verso la destinazione finale – ad esempio: bandwidth associata ad un link).

Esempi di reti di flusso:

- archi rappresentanti un ponte → capacità = massimo peso che il ponte regge;
- archi rappresentanti linee dei trasporti pubblici → capacità = massimo numero di passeggeri che possono viaggiare da  $i$  a  $j$  per unità di tempo;
- archi rappresentanti tubi → capacità = massimo volume di fluido che può fluire attraverso per unità di tempo.

Oltre alle capacità, gli archi della rete hanno normalmente anche un costo, che deve essere pagato per ogni unità di merce che viaggia lungo l'arco. Quando i costi sono presenti, un problema importante è determinare un modo a costo minimo per spedire una certa quantità di merce dai produttori ai consumatori. Se tutti i costi sono nulli, un problema importante è determinare la quantità massima di merce che può fluire lungo la rete date le capacità dell'arco.

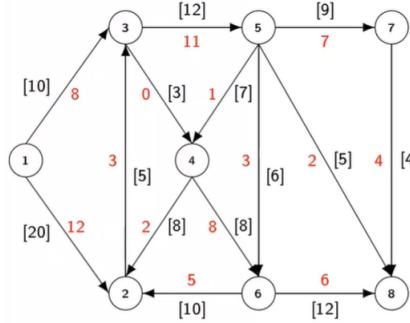
**Definizione:** Una **rete di flusso** (a volte chiamata anche rete di trasporto), formalmente, è una tripla  $(G, k, c)$ , dove:

- $G = (V, A)$  è un grafo diretto;
- $k : A \mapsto \mathbb{R}^+$  sono le **capacità** degli archi, dove  $k_{ij}$  è la capacità di  $(i, j) \in A$ ;
- $c : A \mapsto \mathbb{R}$  sono i **costi** degli archi, dove  $c_{ij}$  è il costo di  $(i, j) \in A$ .

**Definizione:** Un vettore  $x \in \mathbb{R}^A$  è chiamato un **flusso** (o scambio) se è possibile per le capacità, ovvero se:

$$0 \leq x_{ij} \leq k_{ij} \quad \forall (i, j) \in A$$

**Esempio di rete di flusso:** tutti i costi pari a 0.



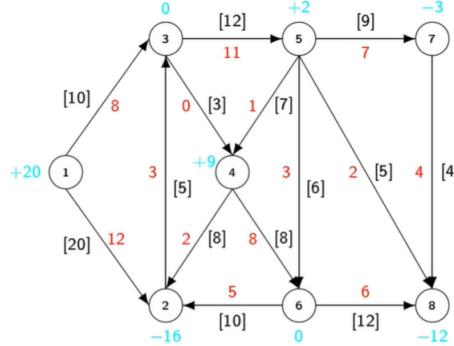
Le capacità vengono rappresentate tra parentesi come  $[k_{ij}]$ . Gli scambi in rosso  $x_{ij}$ . Archi saturi:  $(4,6), (7,8)$ . Archi vuoti:  $(3,4)$ .

**Definizione:** Sia  $G$  una rete di flusso e  $x$  un flusso in  $G$ . Dato  $v \in V$ , definiamo la **divergenza** di  $x$  in  $v$  come:

$$\begin{aligned}\Delta_x(v) &:= \sum_{(v,j) \in \delta^+(v)} x_{vj} - \sum_{(i,v) \in \delta^-(v)} x_{iv} \\ &= x(\delta^+(v)) - x(\delta^-(v)) \\ &= \text{merce in entrata} - \text{merce in uscita}\end{aligned}$$

Sia  $S := \{v : \Delta_x(v) > 0\}$  e  $T := \{v : \Delta_x(v) < 0\}$ . Allora possiamo interpretare lo scambio come la spedizione di una quantità  $\Phi := \sum_{v \in S} \Delta_x(v)$  dai nodi in  $S$  (fonti) ai nodi in  $T$  (pozzi). Infatti, vedremo che tutti i flussi che lasciano i nodi in  $S$  finiscono in nodi in  $T$ .

**Esempio:** Somma delle divergenze: Positive  $+20 + 9 + 2 = +31$ , Negative:  $-3 - 16 - 12 = -31$ . Divergenza totale = 0 (non è una coincidenza).



**Teorema:** Dato un flusso  $x$  in una rete  $G$ , sia  $S := \{v : \Delta_x(v) > 0\}$ ,  $T := \{v : \Delta_x(v) < 0\}$  e  $N := \{v : \Delta_x(v) = 0\}$ . Allora:

$$\sum_{v \in S} \Delta_x(v) = - \sum_{v \in T} \Delta_x(v)$$

**Dimostrazione:** Abbiamo:

$$0 = \sum_{v \in V} x(\delta^+(v)) - \sum_{v \in V} x(\delta^-(v))$$

(dato che ciascun  $(i, j)$  contribuisce  $x_{ij}$  a  $x(\delta^+(i))$  e  $-x_{ij}$  a  $x(\delta^-(j))$ );

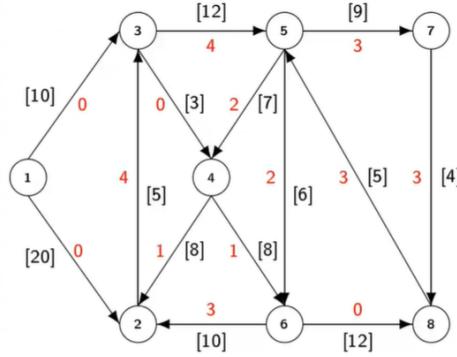
$$\begin{aligned} &= \left[ \sum_{v \in S} x(\delta^+(v)) + \sum_{v \in N} x(\delta^+(v)) + \sum_{v \in T} x(\delta^+(v)) \right] - \\ &\quad \left[ \sum_{v \in S} x(\delta^-(v)) - \sum_{v \in N} x(\delta^-(v)) - \sum_{v \in T} x(\delta^-(v)) \right] \\ &= \sum_{v \in S} \Delta_x(v) + \sum_{v \in N} \Delta_x(v) + \sum_{v \in T} \Delta_x(v) \\ &= \sum_{v \in S} \Delta_x(v) + \sum_{v \in T} \Delta_x(v) \end{aligned}$$

Dall'ultima equazione otteniamo la conclusione.

**Definizione:** Sia  $G$  una rete di flusso e sia  $x$  un flusso in  $G$ . Dato  $v \in V$ , diciamo che  $x$  è conservato in  $v$  (o che  $x$  soddisfa i vincoli di conservazione di flusso in  $v$ ) se  $\Delta_x(v) = 0$ . Un flusso che è conservato in ciascun nodo è detto **circolazione (circulation)**.

Una circolazione può essere interpretata come un scambio di una merce in cui nulla viene creato o distrutto, ma è semplicemente trasportata tra nodi.

**Esempio:** segue un esempio di circolazione:

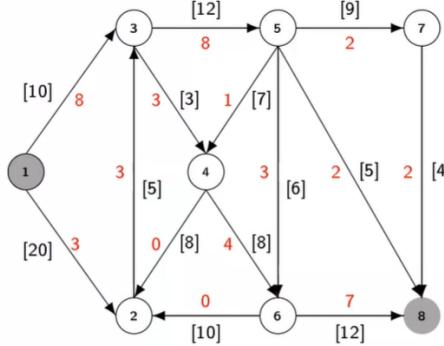


**Definizione:** Siano  $s$  (la fonte) e  $t$  (il pozzo) due nodi distinti in  $V$ . Ciascun flusso che è conservato in ogni nodo  $v \neq s, t$  è chiamato un **flusso s-t**. Il valore di un flusso s-t è definito come:

$$\phi_x := \Delta_x(s)$$

Un flusso s-t  $x$  può essere interpretato come la spedizione di  $\phi_x$  unità della merce da  $s$  a  $t$  lungo una o più rotte nella rete.

**Esempio:** segue un esempio di flusso s-t di valore 11, con  $s = 1$  e  $t = 8$ :



Capacità  $[k_{ij}]$  e flussi  $x_{ij}$ .

## 10.1 Problema del massimo e del minimo flusso

Il problema di determinare la massima quantità di merce che può essere inviata da un singolo produttore verso un singolo consumatore attraverso una rete senza costi può essere formalizzato come segue:

**PROBLEMA: MAX-Flow**

**INPUT:** Una rete di flusso  $(G = (V, A), k, 0)$ . Due nodi  $s, t \in V$ .

**OBIETTIVO:** Trovare un flusso s-t  $x$  che massimizza  $\phi_x$ .

Il problema di trovare il modo più economico di inviare una certa quantità fissa di merce da un produttore a un consumatore è un problema che può essere formalizzato come segue:

**PROBLEMA: MIN-COST Flow**

**INPUT:** Una rete di flusso  $(G = (V, A), k, c)$ . Due nodi  $s, t \in V$ . Un target  $B \in \mathbb{R}^+$ .

**OBIETTIVO:** Trovare un flusso s-t  $x$  di valore  $\phi_x = B$ , che minimizzi  $\sum_{(i,j) \in A} c_{ij}x_{ij}$ .

Vedremo come entrambi i problemi del min e del max siano risolvibili in tempo polinomiale.

**Definizione:** Dato un circuito elementare  $C = (v_1, \dots, v_k)$ , un **flusso circolare** attraverso  $C$ , di valore  $\tau$ , è una **circolazione**  $x^C$  definita da:

$$x_{ij}^C = \begin{cases} \tau & \text{se } (i, j) \in C \\ 0 & \text{altrimenti} \end{cases}$$

Ricordiamo che un circuito elementare è una sequenza di nodi tutti diversi tra loro e che portano da un nodo iniziale a se stesso (un ciclo che non ripete nodi se non l'inizio e la fine).

**Teorema della decomposizione di flusso:** Sia  $x \neq 0$  una circolazione non banale in  $G$ . Allora  $\exists$  cicli elementari distinti  $C_1, \dots, C_q$ , e numeri  $\tau_1, \dots, \tau_q > 0$ , tali che, denotando con  $x^{C_i}$  il flusso circolare attraverso  $C_i$  di valore  $\tau_i$ , si ha:

$$x = \sum_{i=1}^q x^{C_i}$$

In pratica il teorema dice che ogni circolazione è di fatto la somma di flussi circolari.

**Dimostrazione:** Si denoti con  $A^+(x) = \{(i, j) : x_{ij} > 0\}$ . Dalla conservazione di flusso, ogni vertice  $v$  in  $G[A^+(x)]$  ha sia  $d^-(v) > 0$  e  $d^+(v) > 0$  (ha sia archi in entrata che uscita – dato che flussi positivi in entrata di un nodo devono anche uscire) e questo implica che  $G[A^+(x)]$  non può essere aciclico (o ci sarebbero fonti/pozzi). Usiamo l'induzione su  $R(x) =$  numero totale di cicli elementari su  $G[A^+(x)]$ .

Il caso base è  $R(x) = 1$ , cioè,  $G[A^+(x)]$  consiste di un singolo ciclo  $C = (v, v_1, \dots, v_l = v)$ . Dalla legge di conservazione di flusso,  $x_{ij} = x_{vv_1}$  per ogni  $(i, j) \in C$  (su tutti gli archi viaggia lo stesso flusso). Allora, se definiamo questo flusso come  $\tau := x_{vv_1}$  si ha  $x = x^C$ .

Si assuma ora che  $R(x) > 1$  e che il risultato regga per le circolazioni  $x'$  con  $R(x') < R(x)$ . Sia  $C$  qualsiasi ciclo elementare in  $G[A^+(x)]$ , e sia  $\tau := \min\{x_{ij} : (i, j) \in C\}$ . Il flusso  $x' = x - x^C$  è una circolazione. Infatti, per ogni vertice  $v$  su  $C$  aumentiamo di  $\tau$  il flusso su archi entranti e uscenti, così che la conservazione di flusso valga ancora. I nodi appartenenti a  $C$  non sono affetti e così la conservazione di flusso è mantenuta anche per loro. Si noti che  $A^+(x') \not\subseteq A^+(x)$  e  $R(x') < R(x)$ , dato che tutti gli archi su  $C$  che avevano flusso  $x_{ij} = \tau$  hanno ora flusso  $x'_{ij} = 0$ . Per induzione, esistono cicli elementari distinti  $C_1, \dots, C_r$  e valori  $\tau_1, \dots, \tau_r > 0$  t.c.

$$x' = \sum_{i=1}^r x^{C_i}$$

Se impostiamo  $q = r + 1$ ,  $C_q := C$  e  $\tau_q := \tau$ , si ha:

$$x = \sum_{i=1}^q x^{C_i}$$

**Definizione:** Dato un cammino  $s-t$  elementare  $P = (s, v_1, \dots, v_k, t)$  in  $G$ , **un flusso cammino**, attraverso  $P$ , di valore  $\epsilon > 0$ , è un cammino  $x^P$  definito da:

$$x_{ij}^P = \begin{cases} \epsilon & \text{se } (i, j) \in P \\ 0 & \text{altrimenti} \end{cases}$$

**Corollario sul teorema della decomposizione di flusso:** Sia  $G$  una rete di flusso e sia  $x$  un flusso  $s-t$  di valore  $\phi_x > 0$ . Allora esistono cammini distinti elementari  $s-t$  detti  $P_1, \dots, P_p$ , con  $p \geq 1$ ; esistono cicli elementari distinti  $C_1, \dots, C_q$ , con  $q \geq 0$ , e numeri  $\epsilon_1, \dots, \epsilon_p, \tau_1, \dots, \tau_q > 0$ , t.c., denotando con  $x^{P_i}$  il flusso cammino  $P_i$  di valore  $\epsilon_i$ , e con  $x^{C_i}$  il flusso circolare attraverso  $C_i$  di valore  $\tau_i$ , è:

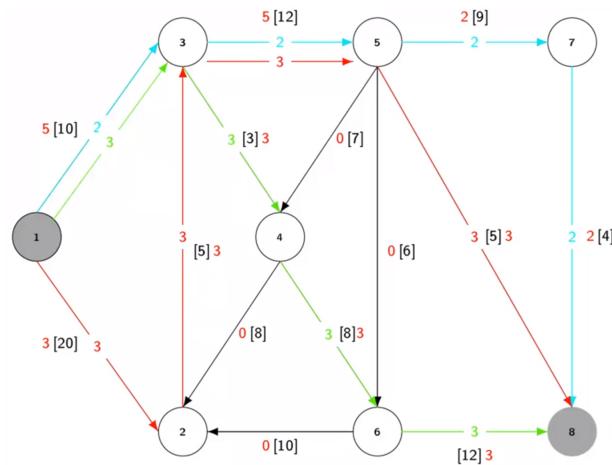
$$x = \sum_{i=1}^p x^{P_i} + \sum_{i=1}^q x^{C_i}$$

In pratica il teorema afferma che ogni un flusso  $s$ - $t$  elementare è scomponibile nella somma di flussi di cammini e flussi circolari

**Dimostrazione:** Aggiungiamo a  $G$  un arco  $(t, s)$  con flusso  $\phi_x$ , così  $x$  è trasformato in una circolazione  $\hat{x}$ . Dal teorema della decomposizione di flusso, è la somma di flussi circolari attraverso  $C^1, \dots, C^d$ . Ogni ciclo  $C_i$  che contiene l'arco  $(t, s)$  è di fatto un cammino  $s$ - $t$ ,  $P_i$  più l'arco  $(t, s)$ . Se in  $\hat{x}$  ignoriamo il componente  $(t, s)$ , quello che rimane è il flusso originale  $x$  che è tra l'altro la somma di flussi (corrispondente al ciclo contenente  $(t, s)$ ) e flussi circolari (corrispondente ai cicli rimanenti).

Questo teorema di decomposizione di flusso mostra come spedire  $\phi_x$  unità della merce da  $s$  a  $t$ . In particolare, dovremmo spedire  $\epsilon_i$  unità sulla rotta determinata dal percorso  $P_i$ , e ignorare i flussi circolari.

**Esempio:** segue un esempio di decomposizione di una rete di flusso (del corollario, in questo caso abbiamo solo cammini):



## 10.2 Restrizioni e generalizzazioni

Senza perdita di generalità, facciamo alcune assunzioni sulle reti di flusso. Questi accorgimenti ci fanno arrivare a reti di flusso che rispettano una sorta di "forma canonica".

(1) La rete originale può contenere archi non orientati.

In questo caso ciascuno di essi viene sostituito da una coppia di archi antiparalleli, entrambi con la stessa capacità e costo dell'originale.

Questi archi vengono poi ulteriormente sottoposti al punto (4) di seguito.

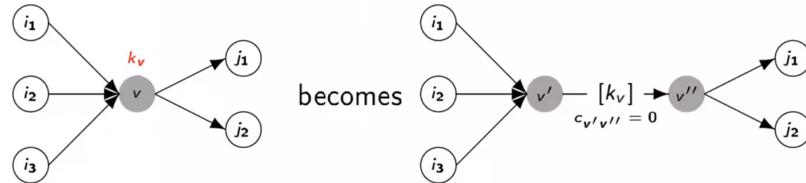


(2) Possiamo soddisfare i vincoli di capacità del nodo come segue.

Si assuma che il nodo  $v$  abbia una specifica capacità  $k_v$  che limita la quantità massima di flusso che può entrare nel nodo.

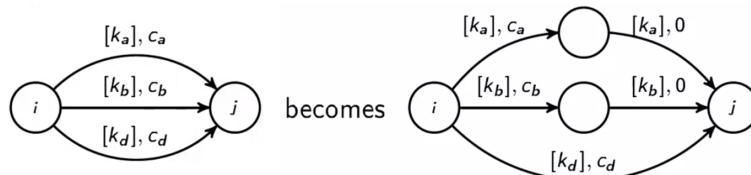
Rimpiazziamo tale nodo con una coppia di nuovi nodi  $v'$ ,  $v''$  e un arco  $(v', v'')$  di capacità  $k_{v',v''} := k_v$  e costo zero.

Ogni arco  $(i, v)$  diventa  $(i, v')$  e ciascun arco  $(v, j)$  diventa  $(v'', j)$ .

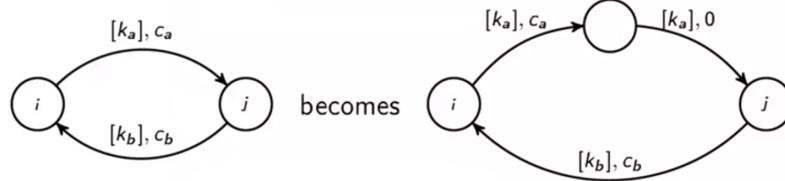


(3) Non ci sono archi paralleli. Infatti, si assume che  $a' = (i, j)$  e  $a'' = (i, j)$  siano archi paralleli con capacità  $k(a')$  e  $k(a'')$  e costi  $c(a')$  e  $c(a'')$ .

Allora possiamo suddividere uno dei due archi, ad esempio  $a'$ , inserendo un nodo  $v$  nel mezzo e impostando  $k(i, v) = k(v, j) = k(a')$  mentre  $c(i, v) = c(a')$  e  $c(v, j) = 0$ .



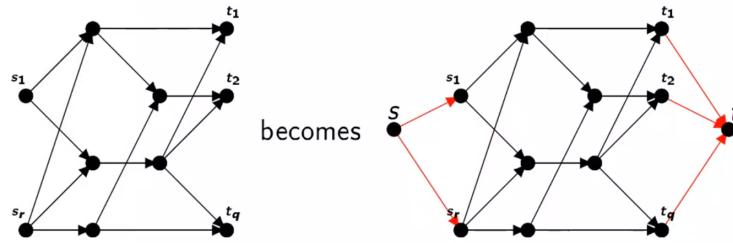
(4) Non ci sono archi antiparalleli. Infatti, si assuma  $(i, j)$  e  $(j, i)$  siano archi antiparalleli. Allora come nel caso (2) sopra, possiamo suddividere uno di essi in due archi paralleli.



(5) Ci sono  $\leq 1$  fonti e  $\leq 1$  pozzi. Si assuma che ci siano fonti  $s_1, \dots, s_r$  e pozzi  $t_1, \dots, t_q$ .

Aggiungiamo una fonte nuova  $s$  e archi  $(s, s_i)$  di capacità  $+\infty$  e costo 0. Aggiungiamo anche un pozzo  $t$  e  $k$  archi  $(t, t_i)$  di capacità  $+\infty$  e costo 0.

È chiaro come ciascun flusso  $s-t$  nella nuova rete corrisponde a un flusso da una fonte a un pozzo nella rete originale.



(6) Se c'è una fonte  $s$  e un pozzo  $t$ , allora  $\delta^-(s) = \delta^+(t) = \emptyset$ . Infatti, la trasformazione precedente porta a una rete con questa proprietà.

### 10.3 Insiemi di taglio e problema del minimo taglio

**Flussi attraverso insiemi di tagli:** Dato un insieme di vertici  $S \subset V$  e un flusso s-t  $x$ , definiamo un flusso attraverso  $S$  come:

$$\Delta_x(S) := \sum_{(i,j) \in \delta^+(S)} x_{ij} - \sum_{(i,j) \in \delta^-(S)} x_{ij} = x(\delta^+(S)) - x(\delta^-(S))$$

$\Delta_x(S) :=$  flusso degli archi che escono da  $S$  - flusso degli archi che entrano da  $S$

Questa definizione estende la nozione di divergenza agli insiemi generali. Infatti:

$$\Delta_x(v) = \Delta_x(\{v\})$$

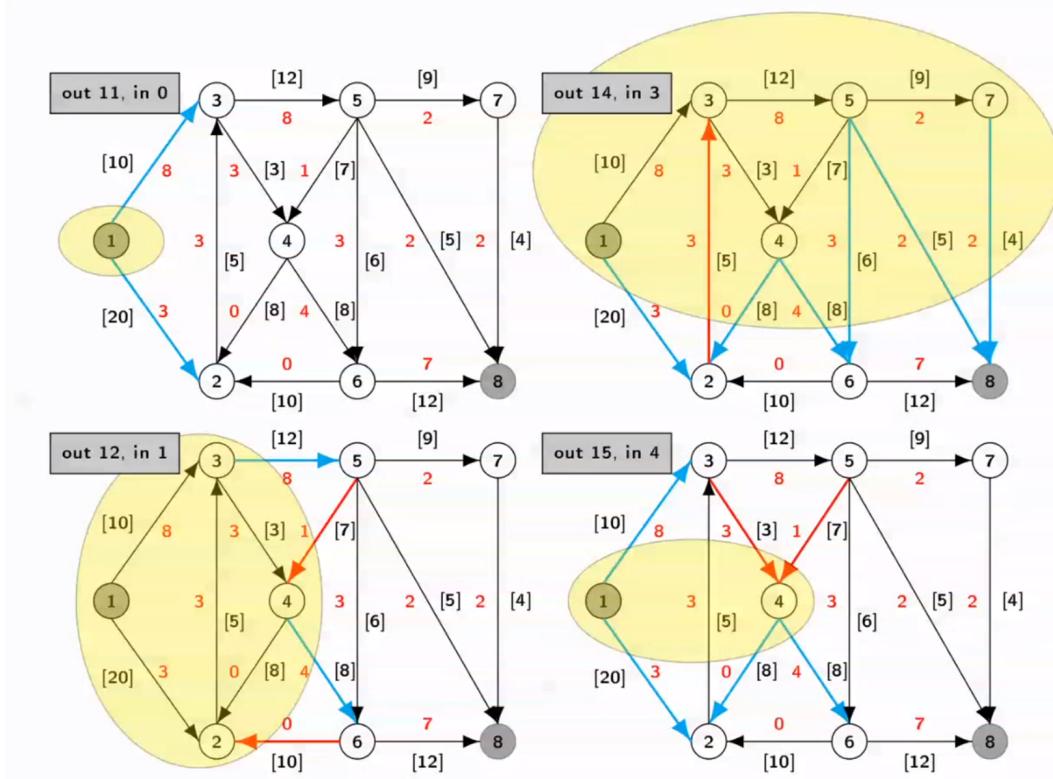
per ogni nodo  $v$ . Siamo particolarmente interessati agli insiemi  $S$  tali che  $s \in S$  e  $t \notin S$ . Chiamiamo ciascun  $S$  un insieme di taglio. Un **insieme di taglio** (o solo taglio) identifica una partizione di  $V$  come  $V = S \cup T$ , dove  $T := V - S$ . Ci sono  $2^{n-2}$  insiemi di taglio in totale.

Per definizione, il valore del flusso è un flusso attraverso un particolare insieme di taglio, ovvero  $S = s$ :

$$\phi_x = \Delta_x(\{s\})$$

Il prossimo teorema mostra come il flusso attraverso ciascun insieme di taglio è sempre lo stesso, ed è uguale al flusso della rete fuori da  $s$ .

**Alcuni esempi:** seguono 4 esempi.



**Teorema:** Sia  $x$  un flusso s-t e  $S$  un insieme di taglio. Allora  $\Delta_x(S) = \phi_x$ .

**Dimostrazione:** Abbiamo:

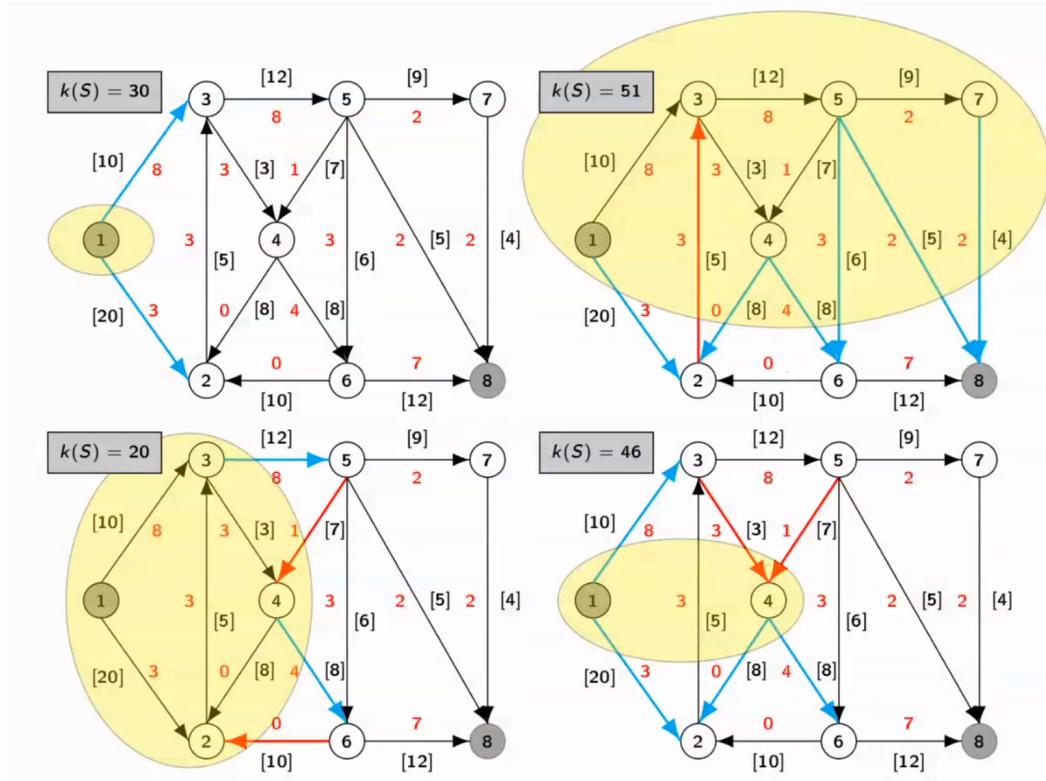
$$\begin{aligned}
 & \phi_x = \Delta_x(s) \\
 &= \sum_{v \in S} \Delta_x(v) \quad (\text{dato che } \Delta_x(v) = 0 \text{ per ogni } v \in S - \{s\}) \\
 &= \sum_{v \in S} x(\delta^+(v)) - \sum_{v \in S} x(\delta^-(v)) \\
 &= \sum_{(i,j):i \in S, j \in S} x_{ij} + \sum_{(i,j):i \in S, j \notin S} x_{ij} - \sum_{(j,i):i \in S, j \in S} x_{ji} - \sum_{(j,i):i \in S, j \notin S} x_{ji} \\
 &= x(\delta^+(S)) - x(\delta^-(S)) \\
 &= \Delta_x(S)
 \end{aligned}$$

### Capacità di un insieme di taglio

Dato un insieme di taglio  $S$ , chiamiamo  $\delta^+(S) \cup \delta^-(S)$  il taglio identificato da  $S$ , e definiamo la **capacità** del taglio e dell'insieme di taglio come:

$$k(S) := \sum_{(i,j) \in \delta^+(S)} k_{ij}$$

Solo degli archi in uscita. Si noti che gli archi all'indietro (cioè archi da  $V \setminus S$  in  $S$ ) non contribuiscono alla capacità del taglio. Seguono degli esempi.



Per ogni insieme di taglio  $S$ , la capacità di  $S$  limita il flusso massimo della rete che può passare da  $S$  a  $V \setminus S$ . Abbiamo il seguente teorema:

**Teorema:** Sia  $S$  un insieme di taglio e  $x$  un flusso s-t. Allora:

$$\Delta_x(S) \leq k(S)$$

**Dimostrazione:** Abbiamo:

$$\Delta_x(S) = x(\delta^+(S)) - x(\delta^-(S)) \leq x(\delta^+(S)) \leq \sum_{(i,j) \in \delta^+(S)} k_{ij} = k(S)$$

**Teorema del Flusso-Max/Taglio-Min forma debole (Max-flow/Min-cut Theorem):** Il massimo valore di un flusso s-t soddisfa:

$$\max_{s-t \text{ flow } x} \phi_x \leq \min_{\text{cutset } S} k(S)$$

Un flusso s-t di valore massimo è chiamato **massimo flusso**, mentre un insieme di taglio di capacità minima è detto **taglio minimo**. Ogni taglio è quindi un collo di bottiglia. È facile intuire che se il taglio minimo ha capacità  $x$ , non c'è motivo per non sfruttarla, e quindi il flusso max sarà = 10 (da lì forma debole del teorema).

## 10.4 Problema del taglio minimo

Trovare un taglio minimo è un problema importante in sé, noto come problema di taglio minimo s-t. Per un grafo non orientato, il problema può essere affermato come segue:

**PROBLEMA: s-t MIN-CUT (indiretto)**

**INPUT:** Un grafo indiretto  $G = (V, E)$ . Due vertici  $s, t \in V$ . Pesi  $w_{ij} \geq 0 \forall ij \in E$ .

**OBIETTIVO:** Trovare  $S \subset V$  t.c.  $s \in S, t \notin S$  e  $w(\delta(S))$  è minore possibile.

Si tratta di trovare quindi il taglio collo di bottiglia. Quando  $w_{ij} = 1$  per ogni  $ij$ , siamo interessati a trovare un taglio di cardinalità minima. Il problema di taglio minimo non orientato può essere ridotto a un problema di taglio di capacità minima su una rete in cui ogni arco  $ij \in E$  è sostituito da due archi  $(i, j)$  e  $(j, i)$  con capacità  $k_{ij} = k_{ji} = w_{ij}$ .

### Relazione tra taglio minimo e flusso massimo

Come vedremo, la soluzione del problema del massimo flusso avrà come sottoprodotto la soluzione del problema del minimo taglio, così che i due problemi siano equivalenti.

A volte ci interessa trovare un taglio minimo complessivo, cioè senza specificare i due nodi  $s$  e  $t$  separati dal taglio. Se  $C(n, m)$  è il tempo per trovare un taglio s-t minimo su un grafo con  $n$  nodi e  $m$  archi, potremmo trovare il taglio minimo complessivo in tempo  $\Theta(n^2)C(n, m)$  tentando ogni possibile coppia  $(s, t)$ .

Possiamo fare di meglio notando che il taglio complessivo minimo separerà un nodo fisso, diciamo il nodo 1, da qualche altro nodo  $t$ , e quindi possiamo trovarlo in tempo  $\Theta(n)C(n, m)$  tentando ogni possibile coppia  $(1, t)$ .

## 10.5 Formulazioni di programmazione lineare

Formulazione di programmazione lineare per il problema del flusso a costo minimo:

$$\min \sum_{(i,j) \in A} c_{ij}x_{ij} \quad (1)$$

soggetto a:

$$\sum_{(s,j) \in \delta^+(s)} x_{sj} - \sum_{(i,s) \in \delta^-(s)} x_{is} = B \quad (2)$$

$$\sum_{(v,j) \in \delta^+(v)} x_{vj} - \sum_{(i,v) \in \delta^-(v)} x_{iv} = 0 \quad \forall v \neq s, t \quad (3)$$

$$x_{ij} \leq k_{ij} \quad \forall (i, j) \in A \quad (4)$$

con variabili  $x_{ij} \geq 0$ .

Il vincolo 2 impone che la divergenza del nodo di partenza sia B, e il 3 è il vincolo di conservazione di flusso.

Indichiamo la matrice  $n \times m$  di incidenza nodo-arco di  $G$  con  $M'$  e sia  $M$  la  $(n-1) \times m$  sottomatrice di  $M'$  in cui manca la riga corrispondente a  $t$ . Inoltre, sia  $b := \text{col}(B, 0) \in \mathbb{R}^{n-1}$ . Quindi il problema di cui sopra può essere riscritto in forma di matrice come:

$$\min c^T x$$

soggetto a:

$$\begin{aligned} Mx &= b \\ Ix &\leq k \\ x &\geq 0 \end{aligned}$$


---

Il problema è stato descritto come un problema di LP supponendo che la merce possa essere frazionata a piacimento (come nel caso, ad esempio, delle spedizioni di petrolio, gas, elettricità, ecc.). Ma cosa succede quando la merce arriva in unità e possiamo spedire solo importi interi? (chiaramente anche in questo caso tutte le portate e i valori di flusso  $B$  sono interi.)

Fortunatamente, questo problema ILP può essere risolto anche come problema LP.

In effetti, la matrice dei vincoli  $\begin{pmatrix} M \\ I \end{pmatrix}$  è TUM. Pertanto, dato che  $b \in \mathbb{Z}^{n-1}$  e  $k \in \mathbb{Z}^m$ , la soluzione del programma lineare sarà naturalmente intera.

Si noti che il problema MAX-FLOW ha esattamente la stessa formulazione del problema MIN-COST FLOW, solo che la funzione obiettivo (1) deve essere sostituita da

$$\max B$$

quindi lo stesso ragionamento vale anche per il problema MAX-FLOW.

Abbiamo così dimostrato quanto segue:

**Teorema:** I problemi MIN-COST FLOW e MAX-FLOW sono polinomiali, anche quando il flusso deve essere intero su tutti gli archi.

Come è accaduto per altri problemi sui grafi, sebbene la programmazione lineare fornisca un algoritmo per risolvere questi problemi, esistono algoritmi combinatori più efficaci. Nel seguito descriveremo la teoria alla base di tali algoritmi combinatori. Storicamente, la prima procedura di questo tipo è l'algoritmo Ford e Fulkerson del 1956 per il problema MAX-FLOW.

## 10.6 Metodo di Ford e Fulkerson per il problema Max-Flow

Da adesso consideriamo una rete di flusso fissa  $G = (V, A)$  con fonte  $s$  e pozzo  $t$ , e con "flusso" implicitamente intendiamo flusso  $s-t$  (si conserva nei nodi diversi da  $s$  e  $t$ ).

**Definizione:** Dato un flusso  $s-t$   $x$ , definiamo la **capacità residua** dell'arco  $(i, j)$  come  $r_{ij} := k_{ij} - x_{ij}$ .

La capacità residua rappresenta la quantità massima di cui possiamo aumentare il flusso su un certo arco. Se la capacità residua è 0, l'arco è saturo.

**Definizione:** Sia  $x$  un flusso  $s-t$ . La rete residua  $R_x = (V, A_x)$  è un grafo diretto pesato sugli archi di lunghezza  $l_{ij}$ . C'è un arco  $(i, j) \in A_x$  se:

- $(i, j) \in A$  e  $x_{ij} < k_{ij}$  (allora  $l_{ij} := r_{ij}$ ); oppure:
- $(j, i) \in A$  e  $x_{ji} > 0$  (allora  $l_{ij} := x_{ji}$ ) (c'era l'opposto).

Nella Figura (a) possiamo vedere una rete di flusso con un flusso  $s-t$   $x$ . Le capacità degli archi sono scritte tra parentesi vicino a ciascun arco. La figura (b) descrive la rete residua  $R_x$  e le lunghezze dei suoi archi.

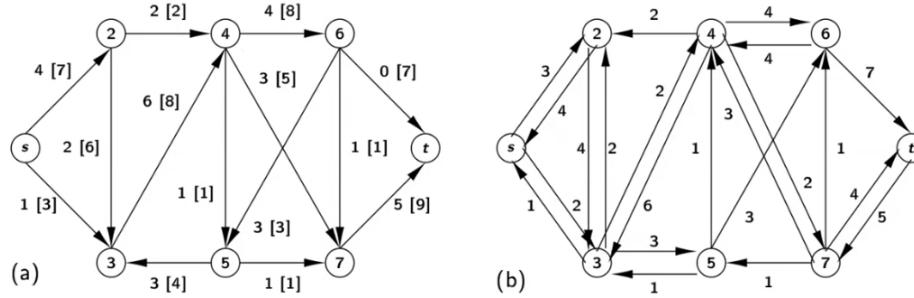


Figura: (a) Una rete di flusso con flusso  $x$  di valore 5; (b) La rete residua del flusso  $x$ ,  $R_x$ .

**Definizione:** Ogni cammino  $P$  da  $s$  a  $t$  in  $R_x$  è chiamato un cammino **aumentante** (ne esistono diversi). Ciascun arco  $(i, j) \in A$  t.c.  $(i, j) \in P$  è chiamato arco in avanti, mentre ciascun arco  $(j, i) \in A$  t.c.  $(j, i) \in P$  è un arco all'indietro. Denotiamo con  $P^+$  gli archi in avanti e con  $P^-$  quelli all'indietro. La lunghezza collo di bottiglia di un cammino aumentante  $P$  è definita come  $\epsilon(P) := \min_{(u,v) \in P} I_{uv}$  (l'arco di lunghezza più piccola).

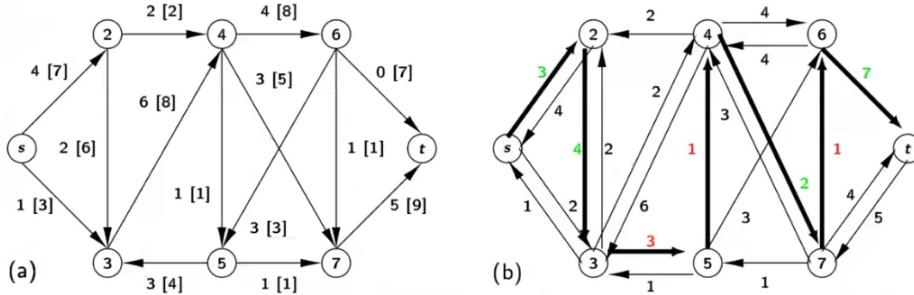


Figura: (a) Una rete di flusso con flusso  $x$ ; (b) La rete residua del flusso  $x$ ,  $R_x$  e un cammino aumentante  $P$ .

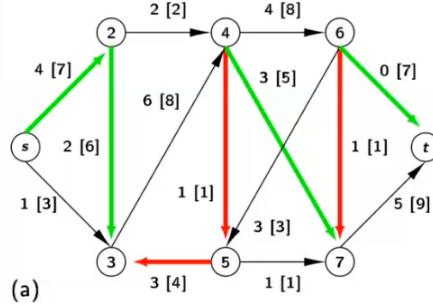


Figura: (a) Archi in avanti  $P^+$  sono verdi. Archi all'indietro  $P^-$  sono rossi.  $\epsilon(P) = 1$ .

Dato un cammino aumentante  $P = (s = v_0, v_1, \dots, v_q, v_{q+1} = t)$ , su tutti gli archi in avanti, il flusso può essere aumentato nella rete originale. Inoltre, il flusso di ciascun arco all'indietro dovrebbe venir decrementato. Sia  $\epsilon > 0$  (la variazione – collo di bottiglia al max) e si consideri una nuova soluzione  $x'$  definita come segue:

$$x'_{ij} = \begin{cases} x_{ij} + \epsilon & \text{se } (i, j) \in P^+ \\ x_{ij} - \epsilon & \text{se } (i, j) \in P^- \\ x_{ij} & \text{altrimenti} \end{cases}$$

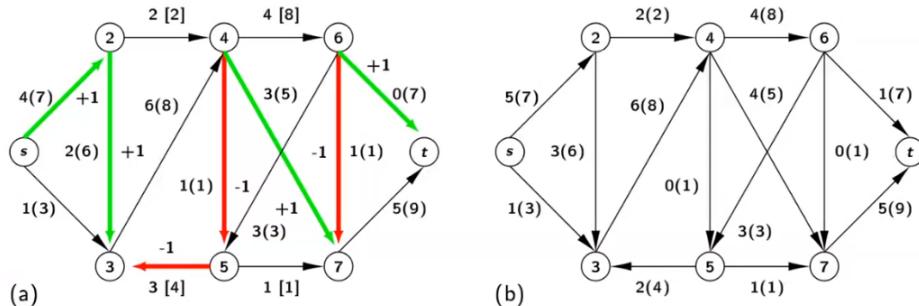


Figura: (a) Il flusso  $x$  di valore 5; (b): Il flusso  $x'$  dopo l'aumento, di valore 6.

Per quanto riguarda la conservazione di flusso, si noti che per ciascun nodo  $v \neq s, t$  si ha  $\Delta_{x'}(v) = \Delta_x(v) = 0$ . Infatti, se  $v$  non viene visitato da  $P$ , abbiamo  $\Delta_{x'}(v) = \Delta_x(v)$ . Altrimenti, sia  $v = v_i$  con  $1 \leq i \leq q$ . Abbiamo allora 4 casi:

(i)  $(v_{i-1}, v), (v, v_{i+1}) \in P^+$ . Allora  $x'(\delta^-(v)) = x(\delta^-(v)) + \epsilon$  e  $x'(\delta^+(v)) = x(\delta^+(v)) + \epsilon$  (si veda il nodo 2 in figura (a));

(ii)  $(v_{i-1}, v) \in P^+$  e  $(v, v_{i+1}) \in P^-$ . Allora  $x'(\delta^-(v)) = x(\delta^-(v)) + \epsilon - \epsilon$  and  $x'(\delta^+(v)) = x(\delta^+(v))$  ( nodi 3 e 7 (a));

(iii)  $(v, v_{i-1}) \in P^-$  e  $(v, v_{i+1}) \in P^+$ . Allora  $x'(\delta^-(v)) = x(\delta^-(v))$  e:  $x'(\delta^+(v)) = x(\delta^+(v)) + \epsilon - \epsilon$  ( nodi 4 e 7 in figura (a));

(iv)  $(v, v_{i-1}), (v_{i+1}, v) \in P^-$ . Allora  $x'(\delta^-(v)) = x(\delta^-(v)) - \epsilon$  e  $x'(\delta^+(v)) = x(\delta^+(v)) - \epsilon$  (nodo 5 in figura (a)).

Vediamo che in tutti i casi si ha:

$$\Delta_{x'}(v) = x'(\delta^+(v)) - x'(\delta^-(v)) = x(\delta^+(v)) - x(\delta^-(v)) = \Delta_x(v)$$

La rete residua può essere utilizzata per trovare un percorso aumentante o per dimostrare che un certo flusso è di fatto ottimale. Abbiamo il seguente teorema:

**Teorema:** Sia  $x$  un flusso in una rete di flusso  $(G, k, c)$ . Allora  $x$  è un flusso massimo sse non c'è un cammino aumentante nella rete residua  $R_x$ .

**Dimostrazione:** ( $\Rightarrow$ ) Chiaramente, se c'è un cammino aumentante  $x$  può essere migliorato e quindi non è massimo. ( $\Leftarrow$ ) Sia  $S := \{v : \exists \text{ cammino } s \rightarrow v \text{ in } R_x\}$ . Dalle nostre assunzioni,  $t \notin S$  cioè,  $S$  è un insieme di taglio. Si noti che devono essere saturi per ogni  $(i, j) \in A$  t.c.  $i \in S, j \notin S$ , deve essere  $x_{ij} = k_{ij}$  e devono essere scarichi per ogni  $(i, j) \in A$  t.c.  $i \notin S, j \in S$ , deve essere  $x_{ij} = 0$ . Queste condizioni implicano che:

$$\Delta_x(S) = x(\delta^+(S)) - x(\delta^-(S)) = k(\delta^+(S)) - 0 = k(S)$$

e dal corollario del MaxFlow-MinCut, concludiamo che  $x$  non può essere aumentato e quindi è un flusso massimo (e anche che  $S$  è un taglio minimo).

**Corollario:** (Teorema del Massimo Flusso - Minimo Taglio forte): Sia  $x^*$  un flusso massimo e sia  $S^*$  un insieme di taglio di capacità minima  $(G, k, c)$ . Allora il valore del flusso massimo coindice con la capacità del taglio minimo:

$$\phi_x^* = k(S^*)$$

**Dimostrazione:** Dal teorema precedente, sappiamo che esiste un insieme di taglio  $S$  (l'insieme dei nodi raggiungibili da  $s$  in  $R_{x^*}$ ) t.c.  $\phi_x^* = k(S)$ . Quindi  $S$  è un insieme di taglio di minima capacità, e dunque  $k(S) = k(S^*)$ .

In conclusione, un flusso è massimo se e solo se c'è un taglio  $S$  saturato da  $x$ , cioè un taglio per il quale tutti gli archi in uscita sono saturati e tutti gli archi in entrata sono vuoti.

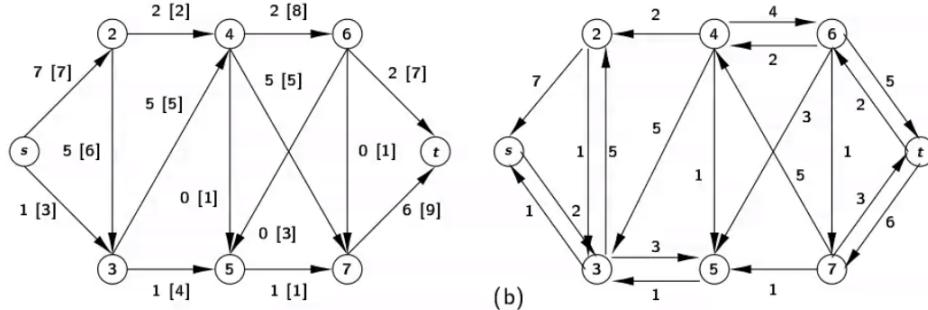
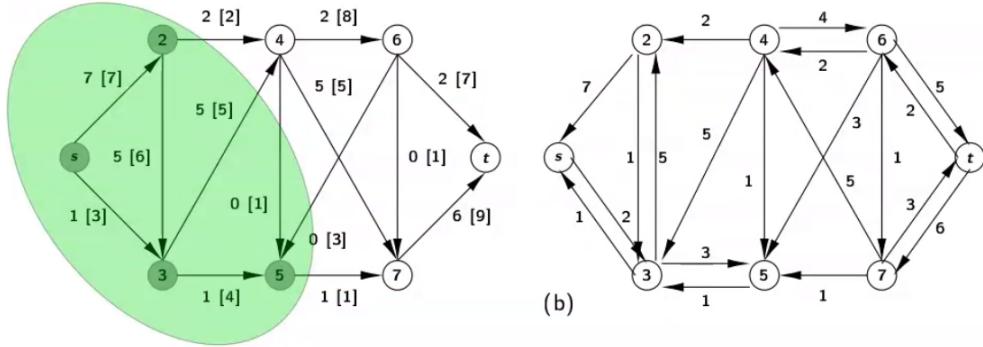


Figura: (a) Una rete con flusso  $x$ ; (b) La rete residua, che mostra come  $x$  sia max.

In figura vediamo  $S$ , tutti gli archi in uscita sono saturi e in entrata sono vuoti.  $S$  è anche il taglio minimo.



La rete residua insieme al teorema maxflow-mincut può essere utilizzata per definire uno schema generale per trovare un flusso  $s \rightarrow t$  massimo (Ford and Fulkerson, 1956).

```

1.  $x := 0; /* starts with zero flow */$ 
2. while there exist  $s \rightarrow t$  paths in  $R_x$  do
3.   let  $P$  be an  $s \rightarrow t$  path in  $R_x$ ; /* finds augmenting path */
4.   for each  $(i, j) \in P$  do
5.     if  $(i, j) \in A$  then /* forward arcs */
6.        $x_{ij} := x_{ij} + \epsilon(P);$ 
7.     else /* backward arcs */
8.        $x_{ji} := x_{ji} - \epsilon(P);$ 
9.     end if
10.   end for
11. end while
12. return  $x$ 

```

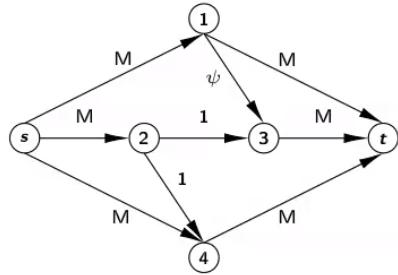
Chiamiamo questo uno "schema" piuttosto che un algoritmo poiché il passaggio 3 non è stato dettagliato abbastanza bene da rendere non ambiguo il modo in cui i percorsi aumentanti vengono trovati ad ogni passaggio. Infatti, se alcune capacità sono numeri irrazionali, una scelta sbagliata di cammini aumentanti può portare l'algoritmo a ripetersi all'infinito come mostrato nel Teorema seguente.

**Teorema:** Se esistono archi con capacità  $c_{ij} \notin \mathbb{Q}$  e ogni cammino aumentante può essere scelto nello step 3, allora per la procedura di Ford-Fulkerson's non è garantita la terminazione.

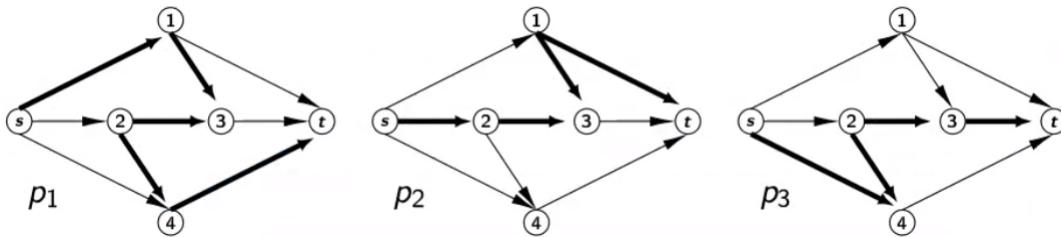
**Dimostrazione (sketch):** Si consideri la rete di flusso seguente. Sia  $M \geq 4$  un intero, mentre  $\psi := (\sqrt{5} - 1)/2$  è irrazionale.

Il flusso max è  $2M + 1 \geq 9$ , determinato dal taglio  $\{s, 2, 4\}$ . Aumentando attraverso una sequenza specifica di cammini aumentanti, dopo  $1 + 2k$  aumenti, il valore del flusso vale  $1 + 2 \sum_{i=1}^k \psi^i$  e dato che il suo valore è limitato da  $3 + 2\psi < 5$ , l'algoritmo non si fermerà mai.

Il primo cammino aumentante è  $(s, 2, 3, t)$  tale che il valore del flusso è aumentato di 1 e l'arco  $(2, 3)$  diventa saturo.



Allora il flusso sarà aumentato attraverso i cammini  $p_1 := (s, 1, 3, 2, 4, t)$   $p_2 := (s, 2, 3, 1, t)$  e  $p_3 := (s, 4, 2, 3, t)$ , seguendo la sequenza infinita  $p_1, p_2$ , poi  $p_1, p_3$ , poi  $p_1, p_2$ , poi  $p_1, p_3$ , etc. Può essere mostrato che l'incremento del fuso per la  $k$ -esima coppia della sequenza (i.e., per entrambi i cammini aumentanti) è  $\psi^k$ . Gli archi bottleneck della coppia  $p_1, p_2$  sono  $(1, 3)$  e  $(2, 3)$ , mentre per  $p_1, p_3$  sono  $(2, 4)$  e  $(2, 3)$ . In conclusione, non solo l'algoritmo fallisce nel terminale, ma il flusso converge a una soluzione sub-ottimale, di val  $1 + 2 \sum_{k=1}^{\infty} \psi^k = 1 + 2 \frac{\psi}{1-\psi} = 3 + 2\psi$ .



Se tutte le capacità sono numeri interi, è garantito che la procedura terminerà sempre, con la soluzione ottimale. L'idea alla base è che aumento sempre di 1, e quindi prima o poi arrivo alla sol. ottimale.

Tuttavia, a seconda del modo in cui vengono scelti i percorsi aumentanti, il tempo di esecuzione della procedura può variare da un polinomio di basso grado a una funzione esponenziale della dimensione dell'input.

**Teorema:** Se tutte le capacità dell'arco sono intere e qualsiasi percorso aumentante può essere scelto nel passaggio 3, la complessità temporale nel caso peggiore della procedura di Ford-Fulkerson è esponenziale.

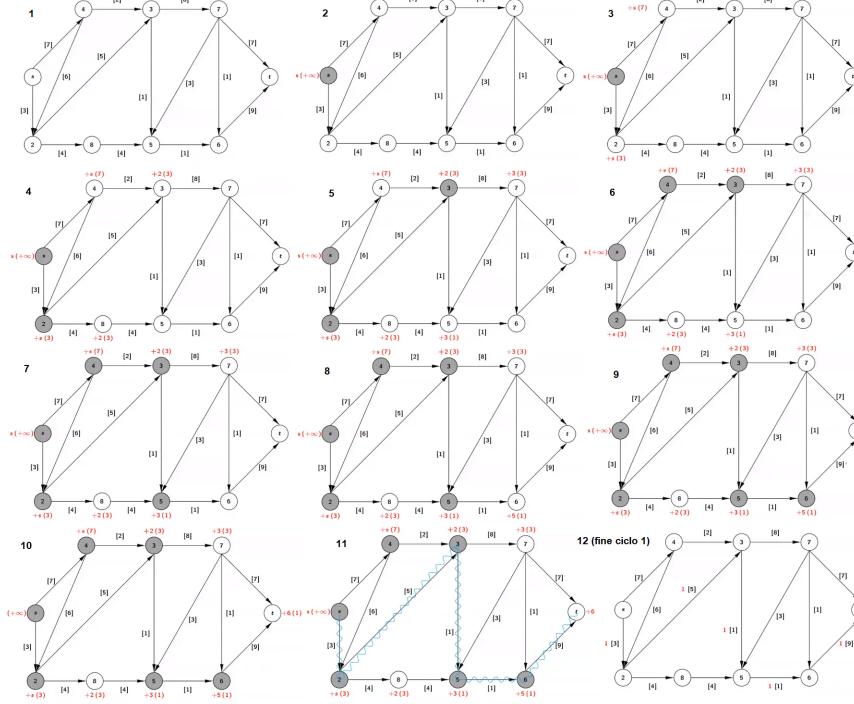
**Pseudocodice:** Descriviamo un'implementazione dell'algoritmo di Ford e Fulkerson con complessità temporale  $O(m^2 \max_{ij} k_{ij})$ . In questa implementazione, la rete residua non è costruita esplicitamente, ma è implicitamente rappresentato sulla rete originaria, associando opportune etichette ai nodi. Per ogni  $v \in V$  abbiamo due etichette, vale a dire  $\epsilon[v]$ , che rappresenta la quantità di "flusso incrementale" che può essere inviato al nodo  $v$  tramite un percorso aumentante.  $\text{pred}[v]$  identifica il padre del nodo  $v$  nel percorso aumentante da  $s$  a  $v$ . Usiamo anche un segno  $\pm$  per distinguere se l'ultimo arco del percorso fino a  $v$  nella rete residua era un arco avanti o indietro. Vale a dire, se  $\text{pred}[v] = +w$ , allora l'ultimo arco era l'arco in avanti  $(w, v)$ , mentre se  $\text{pred}[v] = -w$ , allora era l'arco all'indietro  $(v, w)$ .

```

1.  $\phi_x := 0$ ; forall  $(i,j) \in A$  do  $x_{ij} := 0$ ; /* starting zero flow */
repeat
2.   for  $j = 1, \dots, n$  do  $pred[j] := 0$ ;
3.    $e[s] := +\infty$ ;  $pred[s] := s$ ;  $push(Q, s)$ ;
4.   while ( $\neg empty(Q)$ )  $\wedge (pred[t] = 0)$  do
5.      $v := pop(Q)$ ;
6.     forAllNeigh( $j$  IN  $\delta^+(v)$ ) do /* forward arcs */
7.       if  $(x_{vj} < k_{vj}) \wedge (pred[j] = 0)$  then
8.          $e[j] := \min\{e[v], k_{vj} - x_{vj}\}$ ;  $pred[j] := v$ ;  $push(Q, j)$ ;
9.       end if
10.      forAllNeigh( $j$  IN  $\delta^-(v)$ ) do /* backward arcs */
11.        if  $(x_{jv} > 0) \wedge (pred[j] = 0)$  then
12.           $e[j] := \min\{e[v], x_{jv}\}$ ;  $pred[j] := -v$ ;  $push(Q, j)$ ;
13.        end if
14.      end while
15.    if ( $pred[t] \neq 0$ ) then
16.       $\epsilon := e[t]$ ;  $\phi_x := \phi_x + \epsilon$ ;
17.       $v := t$ ;
18.      while  $v \neq s$  do
19.         $u := |pred[v]|$ 
20.        if  $pred[v] > 0$  then  $x_{uv} := x_{uv} + \epsilon$  else  $x_{vu} := x_{vu} - \epsilon$ ;
21.         $v := u$ ;
22.      end while
23.    end if
24.  until  $pred[t] = 0$ ;
25. /* The set  $S := \{i : pred[i] \neq 0\}$  is a minimum-capacity cutset */
26.

```

**Esempio:** solo il primo ciclo di esecuzione.

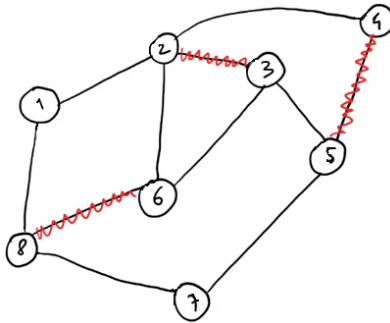


Affinché l’algoritmo di Ford-Fulkerson raggiunga la complessità del tempo polinomiale, è necessario prestare attenzione nella scelta dei percorsi aumentanti. In particolare, si è dimostrato (Edmonds e Karp) che se il percorso aumentante utilizzato nello step 3 è sempre il più breve possibile (cioè con il minor numero di archi), allora l’algoritmo di Ford-Fulkerson richiede al max  $nm$  aumenti e la complessità complessiva è  $O(m^2n)$ . Il problema del flusso max è polinomiale, ne segue che anche il taglio minimo lo è. Notiamo che, al contrario, il problema del max-cut che richiede di identificare un taglio di max capacità, è NP-hard.

## 11 Accoppiamenti (matchings)

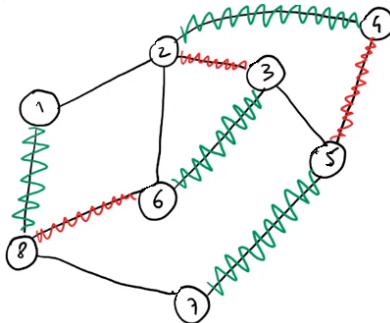
Dato un grafo non orientato  $G = (V, E)$ , un matching  $M$  è un sottoinsieme di archi se sono tutti a 2 a 2 non incidenti di  $E$  t.c.  $\forall i, j, u, v \in M$  si ha  $|i, j, u, v| = 4$  (tutti diversi, se ci fosse un nodo in comune sarebbe 3).

Esempio:  $M = 68, 23, 45$



I nodi non accoppiati si dicono **nodi esposti**. Un matching si dice **perfetto** se non ci sono nodi esposti.

Esempio matching perfetto:  $M = 18, 24, 36, 75$



Una condizione necessaria (non suff.) per l'esistenza di un matching perfetto è che  $n$  sia pari.

Nel caso che gli archi siano pesati, avremmo  $c_e = \text{costo arco } e$ . Se il grafo è pesato, si definisce **costo del matching** la seguente funzione:

$$c(M) := \sum_{e \in M} c_e$$

In generale i problemi di ottimizzazione richiedono di trovare matching di valore massimo.

**PROBLEMA:** MAX-Cost-Matching

**INPUT:** Un grafo  $G = (V, E)$  con pesi  $c_e$ .

**OBIETTIVO:** Trovare un matching  $M$  t.c.  $c(M) := \sum_{e \in M} c_e$  massimo possibile.

Abbiamo due casi particolari: caso in cui  $G$  è bipartito e il caso con i pesi tutti uguali a 1, in quel caso si parla di matching di cardinalità massima (accoppiare più vertici possibili).

**Esempio:** in un grafo bipartito abbiamo da un lato delle risorse, e dall'altro dei task, un esempio potrebbe essere massimizzare il numero massimo di task eseguibili dalle risorse.

I problemi di matching sono polinomiali (sia su grafi bipartiti che non). Per i grafi bipartiti però, trovare l'accoppiamento ottimo è più facile (entrambi polinomiali, ma la complessità tende ad essere minore di quella del caso generico).

È dimostrabile che il problema ha costo polinomiale tramite la programmazione lineare. Nel caso bipartito pesato abbiamo  $G = (V_1, V_2, E)$   $c : E \rightarrow \mathbb{R}$  con funzione obiettivo:

$$\max \sum_{ij \in E} C_{ij} x_{ij}$$

e i seguenti vincoli:

$$\begin{aligned} \sum_{i \in \delta(i)} x_{ij} &\leq 1 \quad \forall i \in V_1 \cup V_2 \\ 0 \leq x_{ij} &\leq 1 \in \mathbb{Z} \quad \forall ij \end{aligned}$$

Nel caso di un grafo bipartito, la matrice dei vincoli relativi alla sommatoria è TUM.

Se  $G$  è completo e la cardinalità di  $V_1$  è uguale a quella di  $V_2$  (problema dell'assegnamento), abbiamo:

variabili:

$$x_{ij} \quad \forall i = 1, \dots, n, \forall j = 1, \dots, n$$

funzione obiettivo:

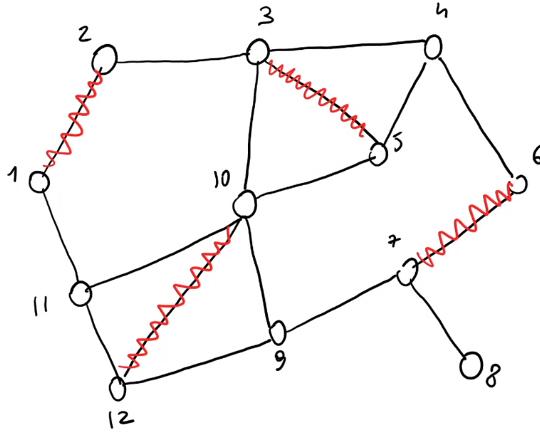
$$\max \sum_i \sum_j c_{ij} x_{ij}$$

e vincoli:

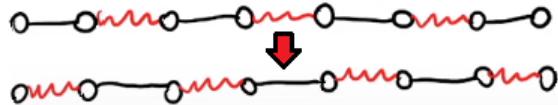
$$\begin{aligned} \sum_{j=1}^n x_{ij} &= 1 \quad \forall i = 1, \dots, n \\ \sum_{i=1}^n x_{ij} &= 1 \quad \forall j = 1, \dots, n \\ x_{ij} &\geq 0 \end{aligned}$$

Parliamo di cammino **alternante** se alternativamente un arco è nel matching e l'altro no.

Esempio: 1 - 2 - 3 - 5 - 4 oppure 9 - 12 - 10 - 5 - 3 - 2.



Un cammino è **aumentante** se collega due nodi esposti. Ad esempio, i nodi esposti sono 11,4,9. Un esempio è 9-12-10-5-3-2-1-11. Questo cammino è aumentante perché se si complementano gli archi che erano nel matching e quelli che non lo erano, si ottiene un nuovo matching di valore aumentato di 1.



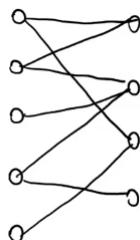
**Teorema:** Se esistono cammini aumentanti rispetto a un matching  $M$  allora  $M$  non è un matching di cardinalità massima.

Vale anche il viceversa ossia, se  $M$  non è massimo, devono esistere cammini aumentanti.

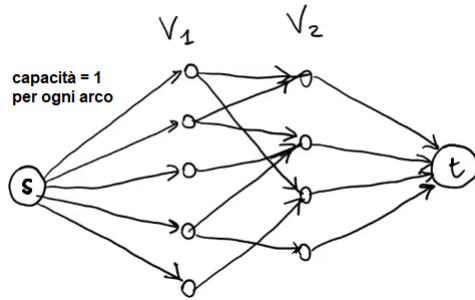
**Dimostrazione:** supponiamo che  $M$  non sia massimo, e sia  $M^*$  un matching massimo. Chiaramente la cardinalità di  $M^*$  è maggiore di quella di  $M$ . Prendiamo il grafo che ha tutti i nodi e come archi solo  $M \Delta M^*$  (differenza simmetrica: archi o solo in  $M$  o solo in  $M^*$ ).

**Teorema:** un matching  $M$  è massimo (cardinalità massima) sse non esistono cammini aumentanti.

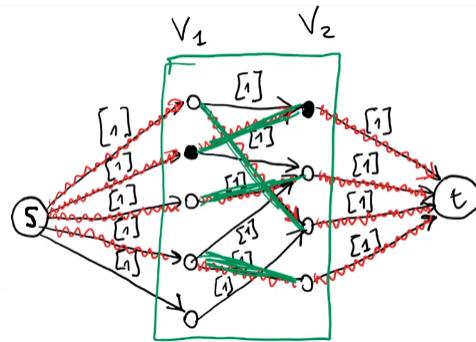
Si può vedere che il problema del massimo flusso, si può utilizzare per risolvere il problema del matching massimo nei grafici bipartiti. Supponiamo quindi di avere un grafo bipartito di cui vogliamo trovare un matching di massima cardinalità:



Dato  $G(V_1, V_2, E)$  si costruisce una rete  $\hat{G} = (\hat{V}, \hat{A})$  dove :  $\hat{V} = V \cup \{s, t\}$  (aggiungo nodo fonte e nodo pozzo);  $\hat{A} = \{(s, i) : i \in V_2\} \cup \{(i, t) : i \in V_2\} \cup \{(i, j) : ij \in E \text{ e con } i \in V_1, j \in V_2\}$  e poniamo la capacità di ogni arco a 1.



L'intuizione è che dal pozzo, per ogni cammino si saturerà un cammino da  $S$  a  $V_1$ , poi da  $V_1$  a  $V_2$  e poi da  $V_2$  a  $T$ . Questi cammini hanno 1 sola "merce" in viaggio quindi non si potrà mai sdoppiare il flusso. Se mi limito a guardare dalla parte del grafo iniziale vedrò che è un matching. Si vede che il massimo flusso è il massimo matching (anche come valore).



Si può quindi usare l'algoritmo di Ford e Fulkerson.

Sia  $G = (V, E)$  un grafo. Un insieme  $C \subseteq V$  è una **copertura** (un "vertex cover") se  $\forall ij \in E$  si ha  $i \in C$  oppure  $j \in C$ . Una copertura è quindi una selezione di nodi che "prendono" tutti gli archi.

### PROBLEMA: MIN-Vertex Cover

**INPUT:** Un grafo  $G = (V, E)$ .

**OBIETTIVO:** Trovare una copertura  $C^* \subseteq V$  t.c. la sua cardinalità sia  $\leq$  a quella di qualsiasi altra copertura  $C$ .

Questo problema è NP-HARD. Il problema del MIN-Vertex Cover e il problema del matching sono relazionati: ogni soluzione di ciascuno dei due problemi fornisce un bound alle soluzioni dell'altro.

**Teorema:** Sia  $G = (V, E)$  un grafo,  $C$  un cover e  $M$  un matching in  $G$ . Allora:

$$|M| \leq |C|$$

**Dimostrazione:** si consideri un matching (non per forza ottimo). Il cover, per ciascun arco del matching deve prendere uno dei due estremi. Indifferentemente da quali dei due nodi scelga, coprirà solo un arco del matching (non essendo adiacenti); quindi almeno un nodo per ogni arco del matching viene selezionato (da qui si deduce che il numero di nodi è almeno pari al numero di archi del matching); se il matching fosse massimo allora la cardinalità combacia, altrimenti dovrei prendere altri nodi per completare la cover.

**Corollario:** siano  $\hat{M}$  un matching e  $\hat{C}$  un cover tali che:

$$|\hat{M}| = |\hat{C}|$$

allora  $\hat{M}$  è un matching MAX e  $\hat{C}$  è un cover min.

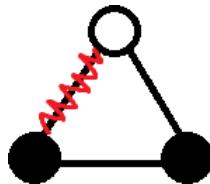
Si ha inoltre che:

$$\max |M| \leq \min |C|$$

quindi appunto

$$|\hat{M}| \leq |\hat{C}|$$

Perchè non possiamo avere  $=$  invece che  $\leq$ ? Non riusciamo a prendere esattamente un nodo da ogni arco del matching e coprire tutto il grafo perché con i grafi non bipartiti si presenta un problema, riscontrabile in questo esempio:



Nei grafi bipartiti, trovare un cover che usa esattamente un solo nodo (ed esclusivamente) del matching (massimo) è sempre possibile. Si ha che il problema del MIN-Vertex Cover sui grafi bipartiti non è NP-HARD ma polinomiale. Formalizziamo:

**Teorema:** Sia  $G = (V_1, V_2, E)$  un grafo bipartito ed  $M$  un suo matching massimo. Allora  $\exists$  una copertura  $C$  t.c.  $|M| = |C|$  (e quindi  $C$  è min.).

**Dimostrazione:** Per induzione su  $|E|$ : supponiamo vero il teorema ( $|E| = k > 1$ ) e sia vero l'asserto per  $|E| = 1, \dots, k-1$ ). Se  $\emptyset$  nodi esposti, allora necessariamente è un matching perfetto e  $C := V_1$  risulta un cover t.c.  $|C| = |M|$ . Nel caso in cui ci siano nodi esposti, invece, si ipotizzi  $uv \in E$  nodo esposto (ipoteticamente della parte di  $V_1$ ). Sia  $i$  il nodo della stessa parte di  $u$  matchato con  $v$ .  $v$  non è esposto e allora  $iv \in M$ . Ipotizziamo ora di rimuovere dal grafo il nodo  $i$  e tutti gli archi incidenti. Sia quindi  $M' = M - \{iv\}$  e otteniamo  $G' = (V', E')$  con  $V' = V - \{v\}$  e  $E' = E - \delta\{v\}$ . Per ipotesi induttiva ora le due cardinalità

sono inferiori ( $|E'| < |E|$ ) perchè ho tolto almeno  $iv$ , inoltre  $|M'| = |M| - 1$ . SE  $\exists$  un cammino aumentante  $x_1, x_2, \dots, x_t$  in  $G'$ , allora  $x_1$  oppure  $x_t$  deve essere in  $i$ . Questo implica che se riprendo lo stesso tratto di cammino (senza perdita di generalità supponiamo  $i = x_1$ ), siccome è aumentante deve finire nella parte opposta in un nodo esposto; a questo punto il cammino che va da  $i$  a  $x_t$  può essere esteso in cammino aumentante in  $G$ ;  $u, v, i, x_2, \dots, x_t$ . Quindi, se  $i, x_2, \dots, x_t$  è aumentante in  $G'$ , allora  $u, v, i, x_2, \dots, x_t$  è aumentante in  $G$ . Così facendo aumento un cammino aumentante in  $G$ , ma in  $G$  non ci sono, quindi è assurdo. Quindi  $M'$  è max in  $G'$ . Per induzione  $\exists$  cover  $C'$  in  $G'$  t.c.  $|C'| = |M'|$ . Allora  $C := C'$  unito  $\{v\}$  è una cover in  $G$  con  $|C| = |C'| + 1 = |M'| + 1 + |M|$ . [DA RIVEDERE TUTTO...]

Sia  $G = (V, E)$  e siano dati pesi  $w_i \forall i \in V$ . Definiamo il peso di un cover  $C$  come:

$$w(C) := \sum_{i \in C} w_i$$

$C^*$  è ottimo se  $w(C^*) \leq w(C) \forall C$ . Il modello di programmazione lineare intera (ILP) è:

$$\min \sum_{i \in V} w_i x_i$$

t.c.  $x_i = 1$  se prendo il nodo  $i$  nel cover e  $= 0$  altrimenti.

I vincoli sono:  $x_i + x_j \geq 1 \forall ij \in E$  e  $x_i \in \{0, 1\} \forall i \in V$ .

## 12 Esercizi

**Consegna:** Esistono 2 compagnie telefoniche; ognuna di esse offre piani tariffari da 1 mese.

	Compagnia 1	Compagnia 2
costo mensile	20.00	15.00
chiamate gratis incluse	1000	800
costo per chiamata extra	0.10	0.07
traffico internet incluso	4 GB	3 GB
costo traffico dati extra	0.5/500MB	0.2/100MB

Le chiamate sono gratis solo se verso lo stesso gestore. Il traffico dati è venduto in pacchetti interi (rispettivamente 500MB e 100MB alla volta).

L'esigenza della compagnia che necessita dei servizi sono le seguenti:

- Chiamate verso numeri C1: 4100;
- Chiamate verso numeri C2: 2600;
- Chiamate verso fissi (non C1, non C2): 1200;
- Traffico dati di 30GB;

Per quanto riguarda le soluzioni ottime:

- Comprare solo sim C1: soluzione ottima è prenderne 4: costo della soluzione: 484;
- Comprare solo sim C2: soluzione ottima è prenderne 4: costo della soluzione: 467.

L'obiettivo è trovare la migliore soluzione (minor costo) in cui si comprano  $x_1$  sim di tipo C1 e  $x_2$  sim di tipo C2 con  $x_1, x_2 \geq 1$ .

### Definiamo un modello di programmazione lineare intera (ILP)

Definiamo  $y_{ij}$  il numero di chiamate extra fatte da una sim della compagnia  $C_i$  verso numeri della compagnia  $C_j$  con  $i \in \{1, 2\}$  e  $j \in \{1, 2, 3\}$  con 3 = numeri fissi.

Abbiamo inoltre variabili  $w_i$ ,  $i \in \{1, 2\}$  che rappresentano quanti pacchetti extra di traffico internet acquistiamo per la compagnia  $C_i$ .

La funzione obiettivo sarà la seguente:

$$\min\{20x_1 + 15x_2 + 0.1(y_{11} + y_{12} + y_{13}) + 0.08(y_{21} + y_{22} + y_{23}) + 0.5w_1 + 0.2w_2\}$$

I vincoli sono i seguenti:

$$x_1, x_2 \geq 1$$

Si ha  $1000x_1$  chiamate gratis verso numeri di  $C_1$ . Quelle extra verso  $C_1$  saranno quindi quelle non comprese (tot - incluse):  $y_{11} + y_{21} = 4100 - 1000x_1$  anche se con l'uguale abbiamo

un problema: ci vincola a prendere al max 4 sim di  $C_1$  in quanto valori maggiori causerebbero valori negativi. Il vincolo corretto risulta:

$$y_{11} + y_{21} = \max\{4100 - 1000x_1, 0\}$$

In realtà è sufficiente  $y_{11} + y_{21} \geq 4100 - 1000x_1$  perchè l'upper bound è garantito dalla funzione obiettivo, che non porrà mai valori maggiori di 0, quando 0 è la sol. ottimale.

I vincoli sono quindi:

$$y_{11} + y_{21} \geq 4100 - 1000x_1$$

$$y_{12} + y_{22} \geq 2600 - 800x_2$$

$$y_{13} + y_{23} \geq 1200$$

Per quanto riguarda il traffico extra: 1 blocco = 100MB, abbiamo:

Compagnia 1: 40 blocchi compresi + 5 blocchi per pacchetto extra;  
 Compagnia 2: 30 blocchi compresi + 1 blocco per pacchetto extra;

Servirebbero 300 blocchi in totale, quindi:

$$40x_1 + 5w_1 + 30x_2 + w_2 \geq 300$$

Potrei eccedere ma risparmiando, per quello c'è  $\geq$  e non  $=$ .

Note a posteriori: si poteva evitare di introdurre le variabili  $y_{ij}$  in quanto le chiamate extra sono più economiche con le sim di  $C_2$ . Si può dedurre durante la modellazione o dal fatto che il solver eviterà chiamate da extra da  $C_1$ , e le variabili varranno sempre 0 nella sol. ottimale.

**Consegna:** Si ha un grafo  $G = (V, A)$  orientato tale che:  $V = \{\text{tutti gli animali}\}$  e  $A = \{\text{coppie di animali incompatibili}\}$  (incompatibile:  $(i, j) \in A \rightarrow i \text{ mangerebbe } j \text{ se si trovasse nella stessa gabbia}$ ). Ogni gabbia ha una capacità  $C$  e ogni animale  $i$  occupa uno spazio  $S_i$ . Obiettivo: pianificare la trasferta, utilizzando il numero minimo di gabbie possibili.

**Definiamo un modello di programmazione lineare intera (ILP)**

Al massimo serviranno  $n$  ( $= |V|$ ) gabbie, siano esse  $1, 2, 3, \dots, n$ .  $y_i \in \{0, 1\}$  = "utilizzo o meno la gabbia  $i$ ".

La funzione obiettivo sarà:

$$\min \sum_{i=1}^n y_i$$

ulteriori variabili binarie sono  $x_{ij}$  con  $i \in V, j = 1, \dots, n$  che dicono se l'animale  $i$  viene messo nella gabbia  $j$ .

I vincoli sono i seguenti:

$$\sum_{j=1}^n x_{ij} = 1 \quad \forall i \in V$$

ovvero ogni animale deve essere messo in una gabbia. Inoltre:

$$\sum_{i=1}^n S_i x_{ij} \leq C \quad \forall j = 1, \dots, n$$

che garantisce che le gabbie non siano più cariche del consentito. Inoltre:

$$x_{ij} \leq y_j \quad \forall i \forall j$$

Se un animale è nella gabbia  $j$  allora la gabbia  $j$  deve essere impostata come utilizzata. Il vincolo finale modella l'incompatibilità tra gli animali:

$$x_{uj} + x_{vj} \leq 1 \quad \forall j = 1, \dots, n \quad \forall (u, v) \in A$$

(non possono entrambe valere 1).

## 13 Appendice: recap problemi e algoritmi

- **Ordine topologico sui DAG:** backward e forwards labeling;
- **Min Spanning Tree:** Prim e Kruskal;
- **Spanning Tree con pesi negativi:** correggo i pesi degli archi rendendoli positivi, aggiungendo il  $\min(c)$ ;
- **Max Spanning Tree:** inverto le diseguaglianze degli algoritmi di Prim o Kruskal;
- **Problema della connessione:** peso 1 a tutti gli archi, eseguo Prim o Kruskal, il grafo è connesso se il costo tot è  $n - 1$ ;
- **Min Bottleneck Spanning Tree:** MST risolve anche MBST (minimum bottleneck);
- **Shortest Path Problem:** ILP + vincoli di subtour elimination: NP-HARD
- **Shortest Path Problem:** senza vincoli di subtour: la matrice è TUM = naturalmente intero = LP: polinomiale, oppure uso algoritmi specifici (i 3 casi sotto);
- **S.P.P. su NNC + pesi non negativi:** Dijkstra;
- **S.P.P. su NNC + DAG:** topological sorting;
- **S.P.P. su NNC + archi negativi ma niente cicli negativi:** Floyd-Warshall;
- **Longest Path Problem:** inverto i pesi ed eseguo SPP;
- **Minimizzare numero di archi in un cammino:** imposto a 1 il peso degli archi, eseguo Dijkstra;
- **Max-Flow:** polinomiale = s-t Min-Cut (equivalenti): LP, la versione ILP è naturalmente intera: LP; oppure con il metodo di Ford e Fulkerson;
- **Min-Cost-Flow:** polinomiale,
- **Taglio di cardinalità minima:** s-t Min-Cut con archi tutti di peso 1, polinomiale;
- **Taglio di cardinalità massima:** NP-HARD;
- **Max-Cost Matching:** polinomiale (sia che il grafo sia bipartito che non – ma sui bipartiti è più veloce): se è bipartito = ILP naturalmente intero (A è TUM) = LP = polinomiale.
- **Matching cardinalità max:** Max-Cost Matching con tutti gli archi di peso 1; Si può risolvere anche (nei bipartiti) con il max-flow impostando tutti gli archi a peso 1 e aggiungendo s-t (usando ad esempio Ford e Fulkerson). Nei non bipartiti: Jack Edmonds, polinomiale;
- **Min-Vertex Cover:** polinomiale sui bipartiti, altrimenti NP-HARD;

## 14 Appendice: ripasso nozioni

### 14.1 Lineare indipendenza

Un insieme di vettori di uno spazio vettoriale è formato da vettori linearmente indipendenti se nessuno di essi può essere espresso come combinazione lineare degli altri vettori dell'insieme; se invece almeno un vettore si può esprimere come combinazione lineare degli altri, allora i vettori sono detti linearmente indipendenti.

Siano  $v_1, v_2, \dots, v_n$  n vettori di un campo  $\mathbb{K}$ . Questi vettori sono linearmente indipendenti se, prendendo n scalari  $a_1, a_2, \dots, a_n \in \mathbb{K}$  e imponendo:

$$a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \dots + a_n\mathbf{v}_n = \mathbf{0}$$

risulta che la precedente uguaglianza è soddisfatta se e solo se

$$a_1 = a_2 = \dots = a_n = 0$$

cioè se e solo se l'unica n-upla di scalari che annulla la combinazione lineare  $a_1\mathbf{v}_1 + a_2\mathbf{v}_2 + \dots + a_n\mathbf{v}_n$  è la n-upla di coefficienti nulli. Diciamo invece che i vettori  $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$  sono linearmente dipendenti se oltre alla n-upla di scalari tutti nulli esiste almeno una n-upla di scalari non tutti nulli che annulla la combinazione lineare.

Fonte: <https://www.youmath.it>

### 14.2 Matrice in forma di Echelon

Una matrice è in forma di Echelon è il frutto risultante dalla procedura di eliminazione Gaussiana. Una matrice è in forma di echelon per righe o per colonna a seconda che l'algoritmo utilizzato abbia operato per righe o per colonne.

Una matrice è in forma di Echelon (per righe) se:

- tutte le righe con soli zeri sono in fondo;
- tutti i pivot (primi coefficienti) di una riga non nulla (non tutti zeri) è strettamente a destra del pivot della riga precedente.

Esempio:

$$\begin{pmatrix} 1 & 4 & 5 & 3 \\ 0 & 6 & 7 & 9 \\ 0 & 0 & 6 & 7 \end{pmatrix}$$

Fonte: [https://en.wikipedia.org/wiki/Row\\_echelon\\_form](https://en.wikipedia.org/wiki/Row_echelon_form)