Disjoint $s - t$: varianti del problema di flusso massimo

È un problema che consiste nel determinare quanti cammini diversi esistono tra una sorgente s e una destinazione t in un grafo. Il numero dei cammini che esistono in un grafo tra s e t è esponenziale, ma possiamo imporre delle regole su questi cammini; una è che questi cammini devono essere disgiunti. Questo concetto si può determinare in 2 modi:

- **Edge disjoint** : i cammini non devono condividere archi, ma possono attraversare gli stessi vertici
- **Vertex disjoint** : i cammini non devono condividere né archi né vertici intermedi

Immaginiamo di dover individuare un percorso e rendere la scelta del percorso che faremo il più possibile imprevedibile a persone diverse.

Esempio : supponiamo un caso di sicurezza, come il viaggio di una figura importante (Trump). Per garantire sicurezza, si vogliono pianificare percorsi diversi e imprevedibili da s (ambasciata americana) a t (Vaticano). Se ci sono molti percorsi alternativi, rendere imprevedibile la scelta del percorso rende impossibile pianificare gli attacchi.

Soluzione problema Edge disjoint

- Caso grafo diretto :
  - Gli archi del grafo hanno direzioni uniche *(come strade a senso unico)*
  - Obiettivo : trovare il numero massimo di cammini $s - t$ che non condividono archi
  - Metodo :
    - Assegniamo capacità unitarie a tutti gli archi $c(e) = 1$ perché vogliamo che ogni arco venga utilizzato 1 volta
    - Calcoliamo il flusso massimo usando il modello di **`MaxFlow`**
    - Il valore del flusso massimo, siccome il flusso è intero e le capacità sono unitarie, è 1
    - Per trovare i cammini, scegliamo solo gli archi su cui passa il flusso, li rimuoviamo dal grafo e ripetiamo questi passaggi fino ad esaurire le possibilità
  - Complessità : $O(nm)$

- Caso grafo indiretto :
  - Trasformiamo ogni arco indiretto in 2 archi diretti opposti, cioè in un arco in cui i nodi si collegano in entrambe le direzioni
  - Questo assicura che ogni arco venga utilizzato al massimo 1 volta per andare o tornare

Soluzione problema Vertex disjoint

Questo vincolo sui nodi è più difficile da trattare rispetto a quello sugli archi, ma c'è un trucchetto :

- Si trasforma ogni nodo in 2 nodi distinti :
  - Nodo 1 : riceve gli archi in ingresso
  - Nodo 2 : invio degli archi in uscita
- Si collegano poi i nodi con un arco di capacità unitaria
- In questo modo si garantisce che al massimo un cammino può essere attraversato da ogni nodo

CIRCULATIONS WITH DEMANDS

Nel problema del massimo flusso standard c'è 1 sola sorgente e 1 solo target. Nel problema delle *circulations* :

- Un insieme S di sorgenti
- Un insieme T di target

(I nodi interni non alterano il bilancio del flusso)

Definizione formale

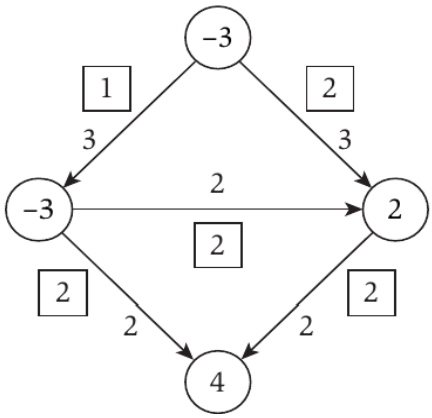
Sia G un grafo diretto con pesi sugli archi interi positivi e poi si ha una funzione d_v definita sui vertici del grafo che rappresenta la quantità di flusso che viene consumata su quello specifico nodo:

- $d_v > 0$: allora il nodo è una destinazione
  - Consumano flusso, quindi : $d_v > 0$
- $d_v < 0$: allora il nodo è una sorgente
  - Quello che entra è meno di quello che esce
  - Producono un flusso totale $>$ di quello che consumano : $d_v < 0$
- $d_v = 0$: allora il nodo è un nodo interno
  - Tutto il flusso entrante è uguale a quello uscente : $d_v = 0$

Applicazione pratica

Immaginiamo un'azienda con :

- Stabilimenti *(sorgenti)* che producono merci
- Destinazioni *(pozzi)* dove le merci vengono consegnate



Il problema consiste nello spostare il flusso di merci tra i nodi rispettando i vincoli imposti dalle capacità degli archi e dai demand d_v.

Perché esista una *circulation*, la somma dei flussi prodotti dalle sorgenti deve essere uguale alla somma delle domande dei pozzi:

$$\sum_{v\,:\,d_v<0} d_v = \sum_{v\,:\,d_v>0} d_v$$

Se questa condizione non è rispettata, non è possibile soddisfare le richieste di flusso.

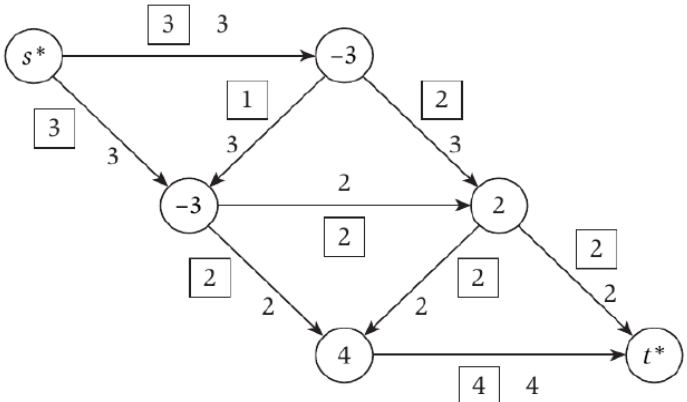
Questo problema è un problema decisionale, non di ottimizzazione. Si chiede semplicemente se è possibile soddisfare i requisiti, senza ottimizzare alcun valore.

L'obiettivo consiste nel capire se con i supply delle sorgenti posso soddisfare la domanda dei target.

Grafo → Rete di flusso

Per risolvere il problema delle *circulations* trasformiamo il grafo originale G in una rete di flusso G':

- Aggiungiamo una *super-sorgente* s^* collegata a tutte le sorgenti
- Aggiungiamo un *super-pozzo* t^* collegato a tutti i pozzi
- Poi dobbiamo assegnare una capacità agli archi da s^* e verso t^*
  - Agli archi che collegano i nodi alle sorgenti mettiamo i demand di quei nodi **con segno positivo**
  - Agli archi che collegano i nodi ai pozzi mettiamo i demand di quei nodi **con segno positivo**



In questa configurazione, tutto il flusso viene prodotto da s^* e distribuito ai nodi sorgenti, tutto il flusso richiesto dai pozzi viene raccolto in t^*.

Si deve risolvere il problema di massimo flusso tra s^* e t^*. Il risultato D rappresenta il valore del flusso che soddisfa tutte le richieste della rete:

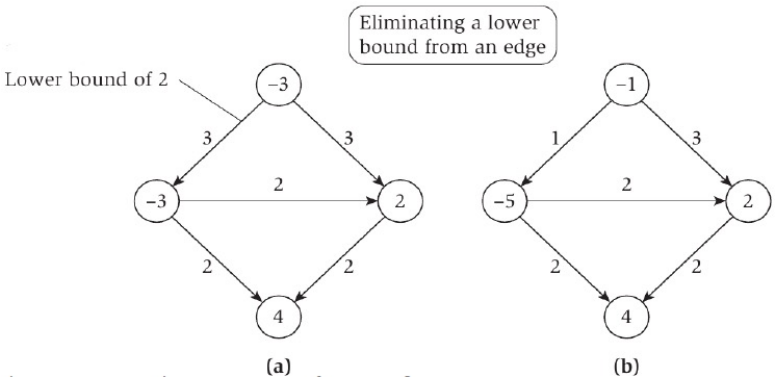
- $D = \sum_{v\,:\,d_v<0} |d_v|$ → Esiste una *circulation* che soddisfa i vincoli
- $D > \sum_{v\,:\,d_v<0} |d_v|$ → C'è troppa domanda e la rete è satura
- $D < \sum_{v\,:\,d_v<0} |d_v|$ → Condizione impossibile, perché il flusso massimo non può superare le capacità della rete

L'unico modo per costruire un flusso da s^* a t^* di valore D è saturare tutti gli archi che collegano la *super-sorgente* s^* alle sorgenti e la *super-destinazione* t^* ai pozzi. Questo è possibile poiché il taglio $(s^*, V + t^*)$ ha una capacità esattamente pari a D. Pertanto, l'unica soluzione è quella di utilizzare al massimo la capacità di questi archi, garantendo che il flusso soddisfi le domande e le offerte. Non esiste un altro modo per ottenere una soluzione valida!

Lower bound

Questo problema è una generalizzazione del problema di *circulation* che include un particolare vincolo di *lower bound* l_e su ogni arco e.

Il problema senza *lower bound* è un caso specifico di questo problema, in cui $l_e = 0$ per tutti gli archi.



Condizioni

Ogni arco e ha un *lower bound* l_e ed una capacità c_e:

$$l_e \leq f(e) \leq c_e$$

Strategia per risolvere

La soluzione è ricondurre un problema con *lower bounds* ad un problema senza *lower bounds*:

- Si inviano l_e unità di flusso su ciascun arco e
  - Questo permette di rispettare il *lower bound* inviando flusso minimo su ciascun arco, eliminandolo di fatto come vincolo
- Si modificano i parametri
  - Demand di ogni nodo :
    - Si toglie dal nodo il flusso minimo l_e che arriva sugli archi in ingresso
    - Si aggiunge al nodo il flusso minimo l_e che parte dagli archi in uscita

$$d_v = d_v - \sum_{\text{e in ingresso}} l_e + \sum_{\text{e in uscita}} l_e$$

    - Se un nodo ha domanda iniziale di 3 e ha un arco in ingresso con $l_e = 2$ e un arco in uscita con $l_e = 1$, la nuova domanda è $d_v = 3 - 2 + 1 = 2$
  - Capacità degli archi : è necessario ridurre la capacità dell'arco del valore del suo *lower bound*

$$c_e = c_e - l_e$$

    - Se un arco ha capacità iniziale di 5 e un *lower bound* di 2, $c_e = 5 - 2 = 3$
  - *Lower Bounds* : vengono impostati tutti a 0

A questo punto, il problema è equivalente a un problema di *circulations* standard senza *lower bounds*, che può essere risolto con gli algoritmi già noti *(ad esempio con quello di Ford-Fulkerson)*.

Osservazioni

- Conservazione del flusso : sottraendo i *lower bounds* dai nodi e dagli archi, la somma complessiva del flusso nel grafo resta invariata
- Condizione necessaria : per garantire l'esistenza di una *circulation*, la somma dei flussi richiesti dalle sorgenti ($\sum_{v \in S} d_v < 0$) deve essere uguale alla somma dei flussi richiesti dai pozzi ($\sum_{v \in S} d_v > 0$)

IMAGE SEGMENTATION

La segmentazione di un'immagine consiste nel dividere l'immagine in varie regioni coerenti → Si ha un'immagine, si vogliono distinguere gli oggetti all'interno di essa.

Caso base

Supponiamo di voler distinguere solo il *foreground* dal *background*. L'immagine digitale è rappresentata come una griglia di pixel e ogni pixel deve essere classificato come appartenente a una delle 2 categorie.

Problema di classificazione ☹

Per ogni pixel i, dobbiamo determinare la probabilità che appartenga al primo piano (a_i) o allo sfondo (b_i). Se un pixel viene classificato in modo errato:

- Errore di classificazione di un pixel nel *foreground* : probabilità b_i
- Errore di classificazione di un pixel nel *background* : probabilità a_i

L'obiettivo è ridurre al minimo la probabilità complessiva di errore.

Funzione di classificazione

$$q(A,B) = \sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{(i,j)\ discordanti} p_{ij}$$

- A : insieme di pixel classificati come *foreground*
- B : insieme di pixel classificati come *background*
- $\sum_{i \in A} a_i$: probabilità complessiva che i pixel classificati nel *foreground* siano effettivamente corretti
- $\sum_{i \in B} b_i$: probabilità complessiva che i pixel classificati nel *background* siano effettivamente corretti
- p_{ij} : penalità per separare 2 pixel i e j che sono adiacenti e dovrebbero appartenere allo stesso insieme
    - Pixel vicini con valori simili sono più inclini ad appartenere allo stesso insieme

L'obiettivo è massimizzare $q(A, B)$ minimizzando la penalità di separazione.

Problema di Min Cut

La segmentazione può essere vista come un problema di "taglio" in un grafo, con alcune differenze:

- Stiamo cercando di massimizzare un obiettivo
- Non ci sono sorgenti né pozzi nella definizione di base
- I nodi hanno valori di probabilità che indicano quanto è probabile che appartengano al *foreground* o al *background*
- Il grafo non è orientato, quindi i collegamenti tra i nodi non hanno direzione

Il problema di segmentazione può essere quindi visto come un problema di Min Cut perché, pur avendo come obiettivo la massimizzazione della funzione di segmentazione $q(A, B)$, questa può essere riformulata come un problema di minimizzazione del costo del taglio nel grafo associato.

Per massimizzare $q(A, B)$, possiamo riscriverla come:

$$q(A, B) = Q - q'(A, B)$$

Dove:

- Q è una costante *(somma totale delle probabilità)*

$$Q = \sum_i a_i + \sum_i b_i$$

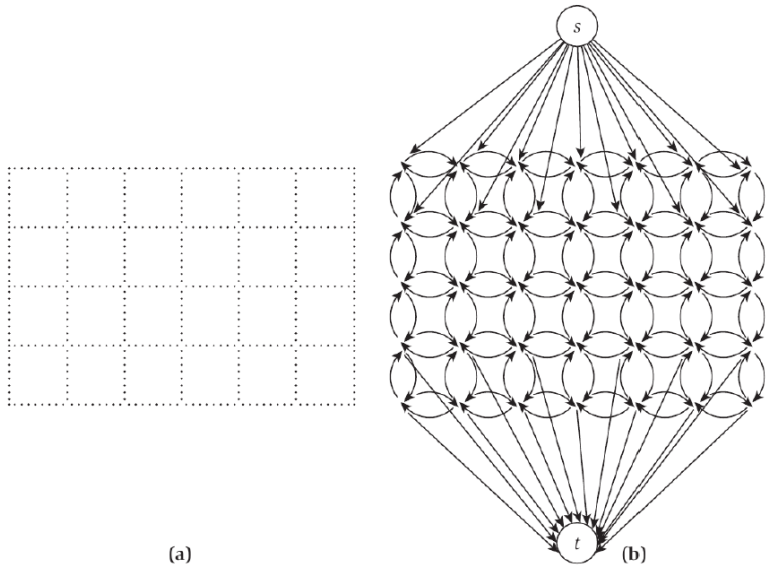
- $q'(A, B)$ è una funzione che rappresenta il costo del taglio:

$$q'(A, B) = \sum_{i \in A} b_i + \sum_{i \in B} a_i + \sum_{(i,j)\ \text{discordanti}} p_{ij}$$

Massimizzare $q(A, B)$ è equivalente a minimizzare $q'(A, B)$, che corrisponde al costo del taglio nel grafo.

Rete di flusso

- Nodi : ogni pixel dell'immagine è rappresentato come un nodo nel grafo
- Archi : gli archi li rendiamo diretti in entrambe le direzioni tra pixel adiacenti
  - Tra pixel : ogni coppia di pixel adiacenti i, j è collegata da un arco con capacità p_{ij}, che rappresenta la penalità di separazione
  - Verso sorgente e pozzo :
    - Dal nodo sorgente s (che rappresenta l'immagine in primo piano) ad ogni pixel i, c'è un arco con capacità a_i, rappresentando il "costo" di assegnare il pixel al *foreground*
    - Da ogni pixel i al nodo pozzo t (che rappresenta l'immagine in secondo piano), c'è un arco con capacità b_i, rappresentando il "costo" di assegnare il pixel al *background*
- Taglio :
  - Un taglio nel grafo separa i nodi in 2 insiemi S *(foreground)* e T *(background)*
  - Il costo del taglio è dato dalla somma delle capacità degli archi che vengono interrotti nel passaggi da S a T



Minimizzando il costo del taglio, si minimizza il costo totale associato agli errori di classificazione e alle penalità di separazione (si può usare come algoritmo del flusso massimo quello di *Ford-Fulkerson*). Il problema di segmentazione è un problema di Min Cut perché la configurazione ottimale del *foreground* e *background* si ottiene minimizzando il costo del taglio nel grafo associato. Questo approccio trasforma il problema della segmentazione da un problema di classificazione probabilistica a un problema di ottimizzazione su grafi.

ALGORITMI DI APPROSSIMAZIONE

La complessità computazionale dei problemi è un argomento di grande interesse nel campo degli algoritmi. Si cerca di rispondere alla domanda: "quanto è difficile risolvere un determinato problema?".

Un problema è calcolabile in modo efficiente se esiste un algoritmo con tempo di esecuzione polinomiale per risolverlo. Tempo polinomiale significa che il tempo necessario per risolvere il problema cresce come una potenza del numero di input: $n^2, n^3, \ldots$

Esistono problemi che non sono calcolabili in modo efficiente? È una domanda nota ma ancora irrisolta

---

Ci sono molti problemi per cui non conosciamo algoritmi con tempo polinomiale, e non possiamo nemmeno dimostrare che tali algoritmi non esistano.

P vs NP

- P : classe di problemi risolvibili in tempo polinomiale
  - Per ogni problema di questa classe conosciamo un algoritmo con tempo polinomiale
- NP : classe di problemi per cui una soluzione può essere verificata in tempo polinomiale
  - Se viene fornita una soluzione *(chiamata "witness")*, possiamo verificarla rapidamente
- $P = NP$?
  - È una domanda nota e ancora irrisolta
  - Fa parte dei problemi del millennio (chi risolve questa domanda guadagna 1 milione di dollari)

Problemi NP-completi

Sono problemi di decisione che appartengono a NP e sono i più difficili in questa classe.

Riduzione in tempo polinomiale

Una riduzione è un modo per trasformare un problema in un altro. In altre parole, possiamo prendere un problema X e "tradurre" ogni sua istanza in un'istanza equivalente di un altro problema Y, mantenendo intatta la difficoltà del problema.

Se questa trasformazione può essere fatta usando un algoritmo che richiede tempo polinomiale, si dice che X è riducibile a Y in tempo polinomiale. Questo implica che, se conosciamo un modo rapido *(polinomiale)* per risolvere Y, possiamo usare questa soluzione anche per risolvere X. Di conseguenza, X non è più difficile di Y.

Se un problema NP-completo *(cioè uno dei più difficili di NP)* può essere ridotto in tempo polinomiale ad un altro problema X, allora X è anche NP-completo. In altre parole, dimostrare che un problema è NP-completo vuol dire collegarlo, attraverso una riduzione, alla rete di difficoltà degli altri problemi NP-completi.

Applicazioni della NP-completezza

- Se $P = NP$
  - Se troviamo un algoritmo in tempo polinomiale per risolvere anche solo un problema NP-completo, significa che possiamo risolvere tutti i problemi in NP in tempo polinomiale. in questo caso la risposta alla famosa domanda "P = NP ?" sarebbe si
  - Questo cambierebbe il panorama dell'informatica
- Se $P \neq NP$
  - Allora non esiste alcun algoritmo in tempo polinomiale per risolvere i problemi NP-completi
  - In questo caso i problemi NP-completi diventano i migliori candidati per rappresentare problemi "intrattabili", cioè problemi che non possiamo risolvere rapidamente

Molti ricercatori ritengono che $P \neq NP$ perché finora non è stato trovato alcun algoritmo polinomiale per risolvere un problema NP-completo, nonostante decenni di studi.

NP-hardness

Un problema H è NP-hard se è almeno difficile quanto un problema NP-completo G. Questo significa che: se possiamo risolvere H, allora possiamo risolvere anche G, perché G può essere trasformato o ridotto in H in tempo polinomiale.

La riduzione in tempo polinomiale garantisce che la difficoltà di H sia almeno equivalente, se non maggiore, di quella di G.

La classe NP-hard include una gamma più ampia di problemi rispetto a NP-completi. Infatti, oltre ai problemi di decisione, include anche:

- Problemi di ricerca
- Problemi di ottimizzazione

Algoritmi

Come progettare algoritmi per problemi per cui un algoritmo in tempo polinomiale è probabilmente irraggiungibile?

- Considerare casi specifici in cui l'input ha una struttura particolare o istanze di piccole dimensioni
- Progettare euristiche
- Utilizzare algoritmi di approssimazione, utili per problemi di ottimizzazione

Gli algoritmi di approssimazione producono soluzioni sub-ottimali con una garanzia di performance: la soluzione prodotta può essere dimostrata vicina all'ottimo.

Garanzia di approssimazione

Per dimostrare una garanzia di approssimazione, dobbiamo confrontare la soluzione con una soluzione ottimale *(che può essere computazionalmente difficile da calcolare)*.

Tecniche comuni per progettare algoritmi di approssimazione:

- Algoritmi greedy
- Metodi di *pricing*
- Programmazione dinamica con input arrotondato

Per rendere significative le approssimazioni, dobbiamo avere una misura della qualità delle soluzioni:

- OPT : qualità della soluzione ottimale per il problema
- ALG : qualità (nel caso peggiore) della soluzione prodotta dall'algoritmo di approssimazione

Per ogni istanza i, vogliamo garantire che:

- Per problemi di massimo : $\mathrm{ALG}(i) \geq \frac{1}{\alpha}\mathrm{OPT}(i)$
- Per problemi di minimo : $\mathrm{ALG}(i) \leq \alpha \cdot \mathrm{OPT}(i)$

Dove α è il fattore di approssimazione, con $\alpha \geq 1$.

Algoritmi di approssimazione greedy

Gli algoritmi greedy sono semplici ma funzionano solo quando la scelta greedy porta ad un risultato ottimale globale. Negli algoritmi di approssimazione greedy, questa scelta è garantita per fornire una soluzione vicina all'ottimale. Tuttavia, per dimostrare che un algoritmo greedy garantisce un'approssimazione è complesso.

Esempio : load balancing

Abbiamo m macchine identiche e n lavori, ciascuno con un tempo di esecuzione t_j. L'obiettivo è assegnare i lavori alle macchine in modo che il carico sia il più bilanciato possibile.

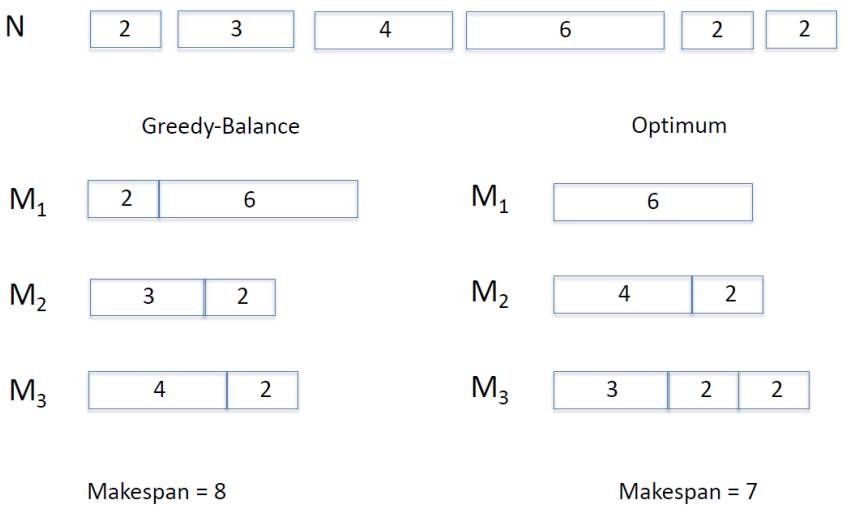
- $A(i)$: insieme dei lavori assegnati alla macchina i
- $T(i)$: carico totale sulla macchina i
- $T = \max_i T(i)$: il *makespan*, cioè il tempo massimo tra tutte le macchine

L'obiettivo è minimizzare il *makespan*.

Greedy-Balance : passi

- Ogni macchina inizia con $T(i) = 0$ e $A(i) = \emptyset$
- Per ciascun lavoro j, lo si assegna alla macchina con il carico minimo
- Si aggiorna il carico totale della macchina e l'insieme dei lavori assegnati

Questa implementazione mira a ridurre il *makespan* in ogni passo.



- Con Greedy-Balance, il *makespan = 8*
- Con la soluzione ottimale, il *makespan = 7*

Analisi

Per confrontare il *makespan* T dell'algoritmo greedy con quello ottimale T^*:

- Si calcola un limite inferiore LB per T^* :
  - $T^* \geq \frac{1}{m}\sum_j t_j$: ogni macchina deve eseguire in media $\frac{\text{somma dei tempi}}{\text{numero di macchine}}$
  - $T^* \geq \max_j t_j$: il lavoro più lungo determina il minimo tempo massimo
- Analisi del carico massimo :
  - Si assegna il lavoro j alla macchina M_i con il carico minimo
  - Prima dell'assegnazione, tutte le macchine avevamo almeno $T - t_j$ come carico
  - Dopo l'assegnazione, il carico massimo T è al massimo $2T^*$

Questo dimostra che l'algoritmo greedy bilancia bene il carico, anche se non raggiunge sempre l'ottimo.

Quanto è precisa l'analisi? – Dimostrazione che Greedy-Balance è vicino a 2 rispetto all'OPT

Consideriamo un insieme di macchine $M = \{M_1, M_2, \dots, M_m\}$ ed un insieme di lavori $N = \{1,2, \dots, n\}$ con $n = m(m-1) + 1$. Tutti i lavori richiedono tempo $t_j = 1$, tranne l'ultimo, che richiede $t_n = m$.

Greedy-Balance assegna uniformemente i primi $n - 1$ lavori tra tutte le macchine. L'ultimo lavoro viene assegnato a 1 delle macchine:

$$T = (m - 1) + m = 2m - 1$$

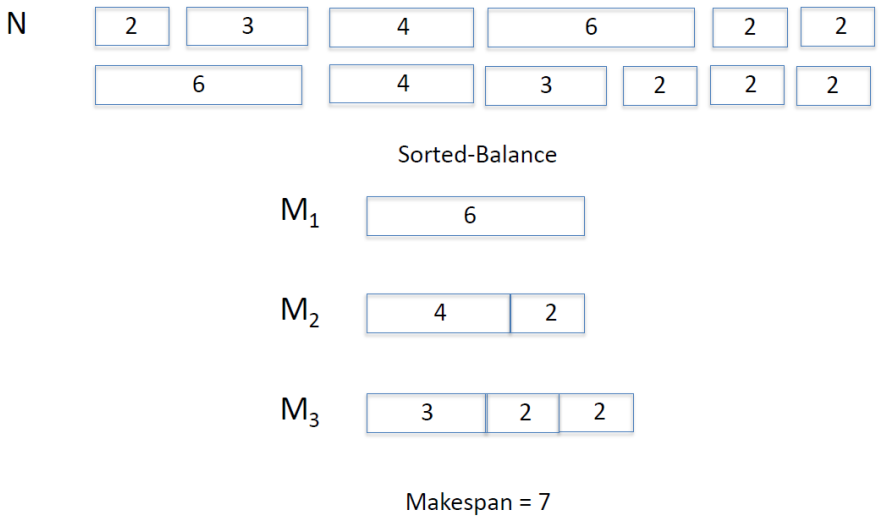
OPT, invece, assegna l'ultimo lavoro *(più lungo)* a una macchina e distribuisce uniformemente i lavori più piccoli tra le altre macchine:

$$T^* = m$$

Il fattore di approssimazione soddisfa : $\alpha \geq 2 - \frac{1}{m}$.

Algoritmo Sorted-Balance

- Si ordinano i lavori in ordine decrescente di tempo
- Per ogni lavoro
  - Si seleziona la macchina con il carico minimo
  - Si assegna quel lavoro a quella macchina
  - Si aggiorna il carico T per tutte le macchine e l'insieme dei lavori assegnati



Analisi

Sia n il numero di lavori e m il numero di macchine:

- Caso $n \leq m$
  - L'algoritmo è ottimale, poiché assegna 1 lavoro a ogni macchina
- Caso $n > m$
  - $T^* \geq 2t_{m+1}$: l'ottimo deve gestire almeno 2 lavori su 1 macchina
  - Assunzione : la macchina i-esima riceve almeno 2 lavori
    - il 2° lavoro assegnato a quella macchina ha tempo $t_{m+1} \leq \frac{1}{2}T^*$
  - Conclusione : $T \leq \frac{3}{2}T^*$

Questo dimostra che Sorted-Balance ha un fattore di approssimazione $\frac{3}{2}$, migliore rispetto a Greedy-Balance.

Problema Vertex Cover

Dato un grafo non orientato G, si chiede di trovare un Vertex Cover ottimale *(con dimensione minima possibile)*.

Risolvere questo problema richiede un tempo computazionale esponenziale *(a meno che P=NP)*.

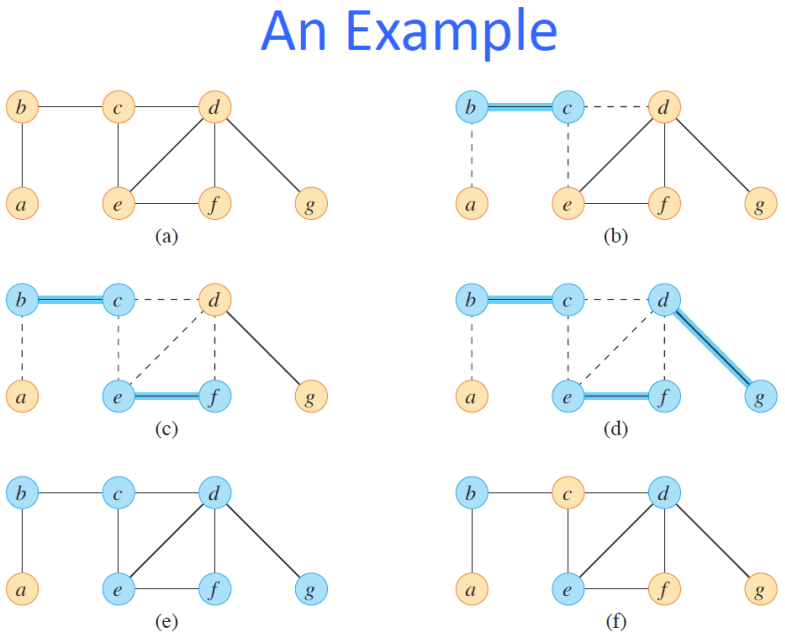
Algoritmo

Questo algoritmo fornisce un *2-apx* Vertex Cover, il che significa che la soluzione trovata ha al massimo il doppio del numero di nodi rispetto a quella ottimale.

Passaggi :

- Inizializzare l'insieme vuoto C
- Copiare tutti gli archi E in un nuovo insieme E'
- Finché E' non è vuoto :
    - Selezionare un arco qualsiasi (u, v) da E'
    - Aggiungere i nodi u e v a C
    - Rimuovere (u, v) e tutti gli archi incidenti su u o v da E'
- Restituire C

Complessità $O(n + m)$ con liste di adiacenza.



Dimostrazione del *2-apx*

Sia C^* il Vertex Cover ottimale e C quello restituito dall'algoritmo. Gli archi scelti nell'algoritmo formano un insieme A, che è un *matching* massimo, ovvero nessun arco in A condivide estremità.

Ogni arco in A richiede almeno 1 nodo nel Vertex Cover; perché ogni arco in A contribuisce con 2 nodi in C, abbiamo:

$$|C^*| \leq |A| \leq |C| = 2|A|$$

Ne consegue che $|C| \leq 2|C^*|$.

Non esistono algoritmi migliori di $2 - \varepsilon$ per approssimare il Vertex Cover a meno che $P = NP$.

Problema Set cover

Questo problema è un problema combinatorio generale. Siano dati in input:

- Un insieme $U = \{1,2, \dots, n\}$
- Una lista di sottoinsiemi $S = \{S_1, S_2, \dots, S_m\}$, dove ogni S_i è contenuto in U
- Una lista di pesi $W = \{w_1, w_2, \dots, w_m\}$, dove $w_i \geq 0$

Un Set Cover $C = \{S_{i1}, S_{i2}, \dots, S_{ik}\}$ è tale che l'unione dei suoi sottoinsiemi copre tutto U:

$$\bigcup_{S_i \in C} S_i = U$$

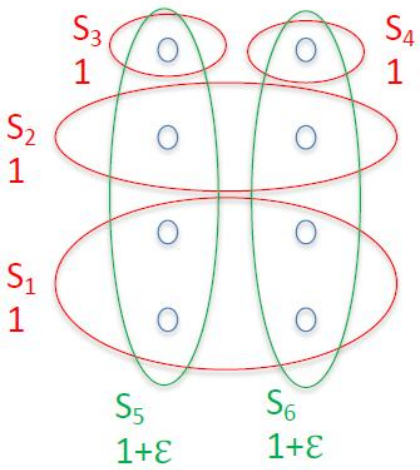
Il peso del Set Cover è la somma dei pesi dei sottoinsiemi in C:

$$w_C = \sum_{S_i \in C} w_i$$

L'obiettivo è trovare un Set Cover con peso minimo.

Esempio

Una squadra di calcio deve firmare contratti con un gruppo di osservatori per creare una rete di scouting mondiale. Ogni osservatore è specializzato in alcune regioni del mondo (che possono essere viste come sottoinsiemi di U) e ha un costo associato (pesi w_i).



L'obiettivo è assumere il numero minimo di osservatori che garantiscano una copertura completa, minimizzando il costo.

Soluzioni illustrate :

$$C' = \{S_1, S_2, S_3, S_4\}, \qquad \text{costo totale } w_{C'} = 4$$

$$C'' = \{S_5, S_6\}, \qquad \text{costo totale } w_{C''} = 2(1+\varepsilon)$$

Algoritmo greedy

- Inizializza l'insieme residuo $R = U$ e la copertura corrente $C = \emptyset$
- Finché $R \neq \emptyset$
  - Seleziona il sottoinsieme S_i che minimizza il rapporto $\frac{w_i}{|S_i \cap R|}$ *(cioè il costo per il nuovo elemento coperto)*
  - Aggiungi S_i a C
  - Rimuovi dal residuo R gli elementi coperti da S_i
- Restituisci C

Ogni iterazione cerca di coprire il maggior numero possibile di nuovi elementi al costo più basso.

Analisi

Per ogni elemento $s \in R \cap S_i$, paghiamo $c_S = \frac{w_i}{|R \cap S_i|}$. Il costo totale della copertura w_C è dato dalla somma dei pesi dei sottoinsiemi in C:

$$w_C = \sum_{S_i \in C} w_i = \sum_{s \in U} c_S$$

Confronto con la soluzione ottimale C^*: si assume che esista una costante $\alpha \geq 0$ tale che

$$\sum_{s \in S_k} c_S \leq \alpha \cdot w_k, \qquad \text{per ogni } S_k \in C^*$$

Analysis of Greedy-Set-Cover -- 2

- We can prove that $\mathrm{H}(|S_k|) \geq \frac{\sum_{s \in S_k} c_s}{w_k}$
  - where $\mathrm{H}(n) = \sum_{i=1}^{n} \frac{1}{n} = \Theta(\ln n)$
- If d^* is the size of the largest set in $\mathcal{C}^*$, then for each set $S_k \in \mathcal{C}^*$, we have

$$H(d^*) \geq \frac{\sum_{s \in S_k} c_s}{w_k}$$

$$w_{\mathcal{C}^*} \geq \frac{1}{H(d^*)} w_{\mathcal{C}}$$

$$w_{\mathcal{C}} \leq H(d^*) \cdot w_{\mathcal{C}^*}$$

- Observe that $d^* \leq n$ and $H(d^*) = \Theta(\ln n)$
  - If d^* is constant our algorithm is close to optimal

Can we do Better?

- Greedy-Set-Cover has a quite large approximation guarantee
- Can we do better?
  - Not much
- Generalizing our motivating example, we can construct an instance where
  - greedy returns a Set Cover with weight $\Omega(\log n)$
  - the best Set Cover has cost $2(1 + \varepsilon)$

LP – LINEAR PROGRAMMING

Le tecniche di programmazione lineare sono applicabili a tutti i problemi che possono essere espressi in un determinato formato. Si tratta di problemi in cui bisogna ottimizzare una funzione lineare: tutte le variabili compaiono con esponente 1.

Le soluzioni ammissibili sono quelle che soddisfano un insieme di vincoli rappresentati da:

- Equazioni lineari
- Disequazioni lineari

Sono tecniche completamente diverse da quelle usate fino ad ora perché permette di astrarsi completamente dalla natura specifica del problema.

Esempio

Un politico vuole vincere un'elezione e deve organizzare una campagna pubblicitaria efficace. Il distretto è suddiviso in 3 aree con rispettivamente: 100.000, 200.000 e 50.000 elettori. L'obiettivo è convincere almeno la metà degli elettori registrati in ciascuna area.

Il politico può investire in pubblicità su 4 temi:

- Costruire più strade
- Supportare l'economia verde
- Sussidi agricoli

- Tassa sulla benzina per migliorare il trasporto pubblico

Ogni valore nella tabella rappresenta il numero di migliaia di elettori che si possono convincere investendo 1.000 euro.

theme	urban	suburban	rural
Build roads	-2	5	3
Green economy	8	2	-5
Farm subsidies	0	0	10
Gasoline tax	10	0	-2

A possible solution
€20,000 on advertising to building roads
€0 on advertising green economy
€4,000 on advertising to farm subsidies
€9,000 on advertising to a gasoline tax
Total cost: 33.000€

An alternative solution
€15,000 on advertising to building roads
€5.000 on advertising green economy
€10,000 on advertising to farm subsidies
€6,000 on advertising to a gasoline tax
Total cost: 36.000€

Which is the best possible strategy?

Formalizzazione del problema di programmazione lineare

- Variabili : x_i, quantità in migliaia di euro investiti per pubblicizzare il tema i
- Vincoli : devono essere convinti almeno la metà degli elettori di ogni area
  - Urbana : $-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50$
  - Suburbana :$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100$
  - Rurale : $3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25$
- Funzione obiettivo : minimizzare la somma dei costi di investimento
  - minimizza $x_1 + x_2 + x_3 + x_4$

Linear Program for the Advertising Problem

$$
\begin{array}{lrcrcrcrcr}
\text{minimize} & x_1 & + & x_2 & + & x_3 & + & x_4 & & \\
\text{subject to} & & & & & & & & & \\
 & -2x_1 & + & 8x_2 & + & 0x_3 & + & 10x_4 & \geq & 50 \\
 & 5x_1 & + & 2x_2 & + & 0x_3 & + & 0x_4 & \geq & 100 \\
 & 3x_1 & - & 5x_2 & + & 10x_3 & - & 2x_4 & \geq & 25 \\
 & & & x_1, x_2, x_3, x_4 & & & & & \geq & 0
\end{array}
$$

The solution of this linear program yields the optimal advertising strategy

Una volta che il problema è stato formalizzato come un sistema di equazioni e disuguaglianze lineari, diventa indipendente dal contesto specifico e può essere affrontato con strumenti standard di ottimizzazione.

Interpretazione geometrica

Ogni funzione lineare in 2 variabili definisce una retta nel piano:

- La retta divide il piano in 2 spazi *(1 a sx e 1 a dx)*
- Un'uguaglianza lineare definisce una retta
- Una disequazione lineare definisce un semispazio *(ad esempio tutto ciò che sta sopra o sotto la retta)*

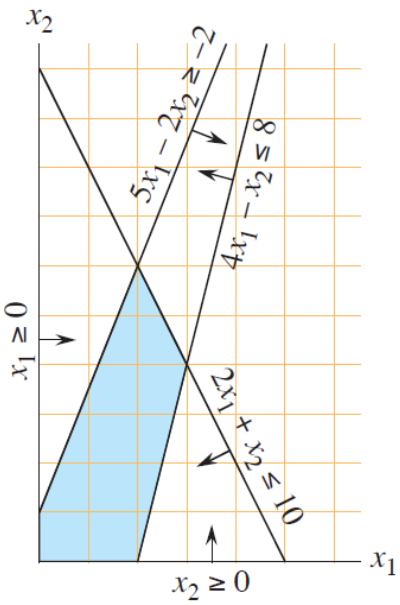
L'intersezione di tutte le rette e i semispazi definiti dai vincoli costituisce la regione ammissibile *(feasible region)*: è l'insieme dei punti che soddisfano tutti i vincoli del problema (sarebbe la regione colorata di blu nell'immagine).

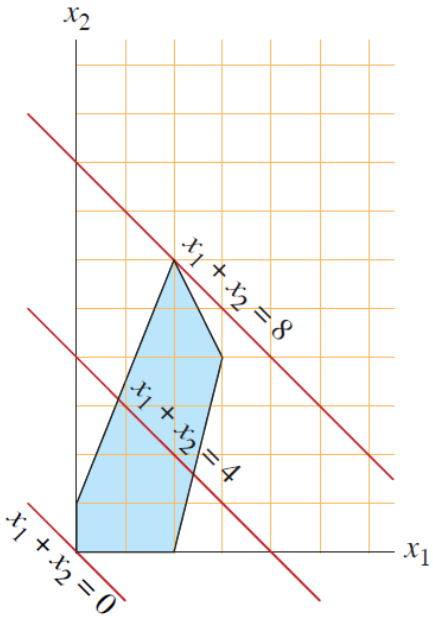
Per ogni valore della funzione obiettivo, si ottiene una retta parallela (sarebbero le rette colorate in rosso). Tra queste rette, dobbiamo selezionare quella che ha

- L'intersezione più alta con la regione ammissibile → se si sta massimizzando
- L'intersezione più bassa con la regione ammissibile → se si sta minimizzando

Passaggi :

1. Si parte dall'origine
2. Si muove una retta nella direzione opposta a quella della funzione obiettivo → fino a trovare l'intersezione con la regione ammissibile





Procedura in spazi a dimensioni superiori

In questo caso:

- Le uguaglianze lineari definiscono iperpiani
- Le disequazioni definiscono semispazi in dimensioni superiori
- La regione ammissibile diventa un *politopo* chiamato *simplesso*

Approccio :

1. Si parte da una soluzione iniziale *(ad esempio dall'origine)*
2. Si muove l'iperpiano definito dalla funzione obiettivo nella direzione opposta fino a raggiungere un'intersezione con il *simplesso*

La soluzione la si trova sempre in uno dei vertici del *simplesso* → ciò semplifica il problema poiché basa valutare i vertici.

Classificazione dei problemi LP

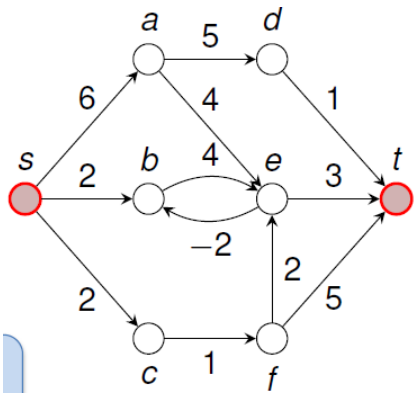
- Problema inammissibile : *infeasible*
    - La regione ammissibile è vuota → nessun punto soddisfa tutti i vincoli
- Problema non limitato : *unbounded*
    - La regione ammissibile esiste, ma non c'è un valore ottimale finito per la funzione obiettivo

- Problema finito : *finite*
    - La regione ammissibile esiste e la funzione obiettivo ha un valore ottimale finito

Problemi risolvibili con LP

Shortest path

- Input
    - Sia $G = (V, E)$ un grafo orientato
    - Ogni arco $(u, v) \in E$ ha un peso w
    - Sia la coppia di vertici $s, t \in V$, con s sorgente e t destinazione
- Obiettivo : trovare un percorso di peso minimo che collega s a t
- Descrizione percorso :
    - Sia il percorso $p = (v_0 = s, v_1, \dots, v_k = t)$
    - Sia il peso del percorso $w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$
- Formulazione come LP :
    - L'obiettivo è massimizzare d_t *(il peso del nodo destinazione* t*)*
    - Vincoli :
        - Di Bellman : per ogni arco (u, v), $d_v \leq d_u + w(u, v)$
        - Con nodo sorgente : $d_s = 0$



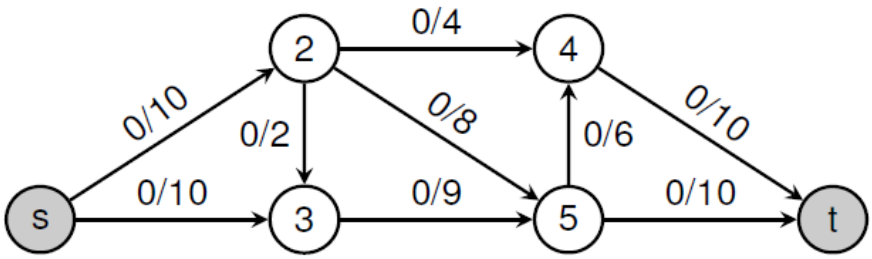
Invece di lavorare direttamente con i costi degli archi, il problema viene riformulato considerando il valore d_v, che rappresenta la distanza *(o il peso cumulativo)* dal nodo sorgente a quello di destinazione.

Dato che il valore $d_s = 0$, il valore massimo di d_t corrisponde al cammino più breve possibile tra s e t che rispetta i vincoli di Bellman. Massimizzare d_t equivale a trovare il cammino che minimizza il peso, perché:

- d_t riflette il peso cumulativo minimo sotto i vincoli imposti
- Qualsiasi valore maggiore violerebbe i vincoli, rendendo la soluzione non valida

MaxFlow

- Input :
    - Sia $G = (V, E)$ un grafo orientato con capacità sugli archi $c(u, v) \geq 0$
    - Sia definita una coppia $s, t \in V$, dove s è la sorgente e t è la destinazione
- Obiettivo : massimizzare il flusso totale da s a t
- Formulazione come LP :
    - L'obiettivo è massimizzare $\sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs}$ *(flusso totale uscente meno quello entrante)*
    - Vincoli :
        - Di capacità per ogni arco : $f_{uv} \leq c_{uv}$
        - Conservazione del flusso : $\sum_{v \in V} f_{uv} = \sum_{v \in V} f_{vu}$ per ogni nodo $u \neq s, t$
        - Non negatività : $f_{uv} \geq 0$



Problema LP in forma Standard

Un problema generico di LP è un problema in cui l'obiettivo è ottimizzare ua funzione lineare soggetta a vincoli lineari:

- Componenti :
  - Funzione lineare : $f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{n} x_i a_i$
  - Equazioni lineari : : $f(x_1, x_2, \dots, x_n) = b$
  - Disequazioni lineari : : $f(x_1, x_2, \dots, x_n) \leq b$
- Forma standard :
  - Massimizzare : $f(x_1, x_2, \dots, x_n) = \sum_{i=1}^{n} x_i c_i$
  - Soggetta a :
    - $\sum_{i=1}^{n} a_{ij} x_i \leq b_j$, per $j = 1,2, \dots, m$
    - $x_i \geq 0$, per $i = 1,2, \dots, n$

Conversione

- Motivi per cui un problema non è in forma standard :
  - L'obiettivo è una minimizzazione anziché una massimizzazione
  - Alcune variabili non hanno vincoli di non negatività
  - Esistono vincoli di uguaglianza
  - Esistono disuguaglianze con $\geq$ anziché con $\leq$
- Conversione :
  - Minimizzazione → Massimizzazione : $\min \ = \ -\max$
  - Variabili senza non negatività : $x_i = x_i' - x_i''$, con $x_i', x_i'' \geq 0$
  - Vincoli di uguaglianza : suddivisi in $\leq$ e $\geq$
  - Disuguaglianze $\geq$: convertite in $-a \leq -b$

1. min = - max
2. $x_i = x'_i - x''_i \qquad x'_i, x''_i \geq 0$
3. $x_i + x_j = b \ \rightarrow \ x_i + x_j \leq 0$ e $x_i + x_j \geq 0$
4. $x_i + x_j \geq b \ \rightarrow \ -x_i - x_j \leq b$

A Conversion Example

$$\min 2x_1 - 3x_2 \qquad x_1 + x_2 = 7 \quad x_1 - 2x_2 \leq 4 \quad x_1 \geq 0$$

$$\max -2x_1 + 3x_2 \qquad x_1 + x_2 = 7 \quad x_1 - 2x_2 \leq 4 \quad x_1 \geq 0$$

$$\max -2x_1 + 3x'_2 - 3x''_2 \qquad x_1 + x'_2 - x''_2 = 7 \quad x_1 - 2x'_2 + 2x''_2 \leq 4 \quad x_1, x'_2, x''_2 \geq 0$$

$$\max -2x_1 + 3x'_2 - 3x''_2 \qquad x_1 + x'_2 - x''_2 \leq 7 \quad x_1 + x'_2 - x''_2 \geq 7 \quad x_1 - 2x'_2 + 2x''_2 \leq 4 \quad x_1, x'_2, x''_2 \geq 0$$

$$\max -2x_1 + 3x_2 - 3x_3 \qquad x_1 + x_2 - x_3 \leq 7 \quad -x_1 - x_2 + x_3 \leq -7 \quad x_1 - 2x_2 + 2x_3 \leq 4 \quad x_1, x_2, x_3 \geq 0$$

Problema LP in forma Slack

Una variabile di *slack* è una variabile aggiunta al problema per trasformare un vincolo di disuguaglianza in uno di uguaglianza:

$$x_1 - 2x_2 \leq 4, \qquad \text{diventa } x_1 - 2x_2 + x_4 = 4, \text{con } x_4 \geq 0$$

Un problema LP in questa forma ha solo vincoli di uguaglianza e non negatività. La forma è:

$$\max c^T x, \qquad \text{soggetto a} : Ax = b, x \geq 0$$

Ogni problema LP standard può essere trasformato in forma *slack* (e viceversa).

$$\begin{aligned} &\max -2x_1 + 3x_2 - 3x_3 \\ &x_1 + x_2 - x_3 \leq 7 \\ &\text{-}x_1 - x_2 + x_3 \leq -7 \\ &x_1 - 2x_2 + 2x_3 \leq 4 \\ &x_1, x_2, x_3 \geq 0 \end{aligned} \quad \Longleftrightarrow \quad \begin{aligned} &\max -2x_1 + 3x_2 - 3x_3 \\ &x_1 + x_2 - x_3 + x_4 = 7 \\ &\text{-}x_1 - x_2 + x_3 + x_5 = -7 \\ &x_1 - 2x_2 + 2x_3 + +x_6 = 4 \\ &x_1, x_2, x_3, x_4, x_5, x_6, \geq 0 \end{aligned}$$

Algoritmi per risolvere problemi LP

Dato un problema LP P con n variabili e m vincoli, esistono algoritmi efficienti per risolverlo in tempo polinomiale:

- Algoritmo dell'ellissoide
- Metodi dei punti interni

Ma sono troppo complessi, per questo utilizziamo altri algoritmi.

Algoritmo Simplex

L'algoritmo del simplesso utilizza un approccio di Local Search :

- Si parte da un vertice del simplesso
- Crea un vertice "adiacente" che migliori la funzione obiettivo
- Si ferma quando raggiunge un ottimo locale *(che è anche un ottimo globale, data la convessità)*

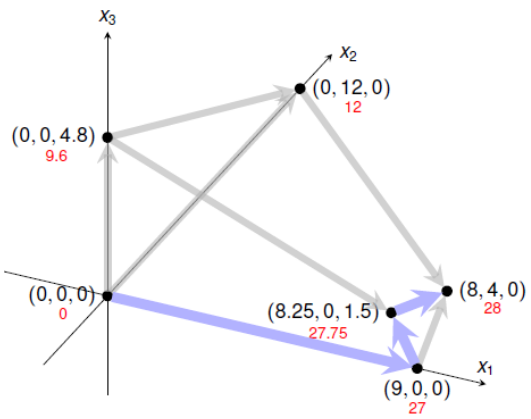
Passaggi :

- Si prende il problema LP in forma *slack*
  - Il set di vincoli è un sistema di equazioni lineari e ogni punto ammissibile corrisponde a una soluzione di questo sistema
- Soluzione : distinguere tra variabili base e non base
  - Variabili base : $\neq 0$
  - Variabili non base : $= 0$
- Ottimo : si raggiunge in un vertice del simplesso
  - Ogni vertice del simplesso è identificato in modo univoco da un set di variabili base

$$\boxed{\begin{aligned} &\max_x c^T x \\ &\quad \text{s..t.} \\ &\quad Ax = b \\ &\quad x \geq 0 \end{aligned}}$$

Il metodo Simplex utilizza un approccio a discesa del gradiente per cercare un ottimo locale tra i vertici del simplesso:

- Si parte da un vertice del simplesso
- Ad ogni passo, si passa a un vertice adiacente sostituendo una variabile base con una non base *(scegliendo quella che massimizza l'incremento della funzione obiettivo)*
- Si ferma quando si raggiunge un ottimo locale *(che è anche un ottimo globale grazie alla convessità)*



Dualità

Data la formulazione di un problema di massimizzazione, la dualità consente di formulare un problema di minimizzazione correlato, che ha lo stesso valore obiettivo ottimale.

La teoria della dualità è più generale della LP e permette di dimostrare che una soluzione è ottimale:

- Dato un problema di massimizzazione → si definisce un problema di minimizzazione correlato tale che i 2 problemi abbiano lo stesso valore obiettivo ottimale
  - Esempio : il duale del problema di massimo flusso è il *MinCut*

Il problema originale si chiama PRIMAL, quello correlato DUAL

Primal problem in standard form

$$\begin{array}{ll} \max \sum_{i=1}^{n} c_i x_i & \\ \text{s.t.} & \\ \sum_{i=1}^{n} a_{ij}\, x_i \leq b_j & \text{for } j = 1, 2, \dots, m \\ x_i \geq 0 & \text{for i} = 1, 2, \dots, n \end{array}$$

Dual problem in standard form

$$\begin{array}{ll} \min \sum_{j=1}^{m} b_j y_j & \\ \text{s.t.} & \\ \sum_{j=1}^{m} a_{ij}\, y_j \leq c_i & \text{for i} = 1, 2, \dots, n \\ y_j \geq 0 & \text{for j} = 1, 2, \dots, m \end{array}$$

How to Obtain the Dual Problem

- Mechanically, to form the dual, we have to
  - change the maximization to a minimization
  - exchange the roles of coefficients on the right-hand side (b_j) and in the objective function (c_i)
  - Replace $\leq$ with $\geq$
  - Each of the constraints in the primal corresponds to a variable in the dual

Primal	Dual
$\max\ 3x_1 + x_2 + 4x_3$	$\min\ 30y_1 + 24y_2 + 36y_3$
$x_1 + x_2 + 3x_3 \leq 30$	$y_1 + 2y_2 + 4y_3 \geq 3$
$2x_1 + 2x_2 + 5x_3 \leq 24$	$y_1 + 2y_2 + y_3 \geq 1$
$4x_1 + x_2 + 2x_3 \leq 36$	$3y_1 + 5y_2 + 2y_3 \geq 4$
$x_1, x_2, x_3 \geq 0$	$y_1, y_2, y_3 \geq 0$

Linear Programming Duality

- Weak linear-programming duality
  - Let $\bar{x}$ be any feasible solution to the primal linear program and $\bar{y}$ be any feasible solution to the dual linear program
  - Then, $\sum_{i=1}^{n} c_i \bar{x}_i \leq \sum_{j=1}^{m} b_j \bar{y}_j$

- If $\sum_{i=1}^{n} c_i \bar{x}_i = \sum_{j=1}^{m} b_j \bar{y}_j$ then $\bar{x}$ and $\bar{y}$ are optimal

- Linear-programming duality
  - If a primal problem and its corresponding dual problem both are feasible and bounded, then for any pair of optimal solutions x^* and y^* it holds that

$$c^T x^* = b^T y^*$$

Fundamental Theorem of Linear Programming

- For each linear program P in standard form one of the following properties holds:
  1. It has an optimal solution with a finite objective value,
  2. It is infeasible,
  3. It is unbounded

- If P is unfeasible, D is unbounded
- If P is unbounded, D is unfeasible
- If P and D are feasible and finite they have the same optimal value