Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione

Tesi di Laurea Triennale in Ingegneria Informatica

Modello di programmazione lineare intera per la pianificazione di turni ospedalieri

Candidato
Giulia Forasassi

Relatore
Prof. Fabio Schoen

Correlatori
Matteo Lapucci

Anno Accademico 2018/2019

Indice

Capitolo 1

Introduzione

... [?]

Capitolo 2

Contenuti teorici

2.1 Ottimizzazione

In questo capitolo vengono spiegati i principali concetti teorici che costituiscono il fondamento del problema preso in esame; dapprima si presenteranno le nozioni che stanno alla base di un generale problema di ottimizzazione e successivamente si spiegheranno più nel dettaglio le caratteristiche principali della **Programmazione Lineare Intera**.

2.1.1 Problema di ottimizzazione

Un problema di ottimizzazione è un problema relativo alla scelta della soluzione migliore, detta **soluzione ottima**, tra quella disponibili in base ad un determinato criterio. Vi sono tre elementi principali che caratterizzano un problema di ottimizzazione:

- *le variabili decisionali*: sono variabili di cui si deve determinare il valore ottimo;

- *l'insieme ammissibile*: è l'insieme delle alternative disponibili per il decisore;
- *la funzione obbiettivo*: indica la relazione funzionale tra le variabili decisionali e altre variabili il cui valore debba essere massimizzato o minimizzato; cioè, in altre parole, costituisce un criterio di scelta tra le possibili soluzioni del problema con l'intento di selezionare solo quelle migliori, cioè quelle che minimizzano il costo o massimizzano il profitto.

Quindi un'istanza di un problema di ottimizzazione è definita come una coppia (A, f), dove A è l'insieme delle soluzioni ammissibili ed $f : A \to \Re$ è la funzione obbiettivo che si deve ottimizzare (massimizzare o minimizzare). Il problema di ottimizzazione viene posto chiedendo di trovare i valori $x \in A$ tali che: $f(x) \leq f(y) \quad \forall y \in A$ (problema di minimizzazione). Tali valori x costituiscono delle soluzioni ottime **globali** per il problema; nel caso in cui risulti invece $f(x) \leq f(y) \quad \forall y \in$ *ad un intorno di* x, allora la soluzione x si dice soluzione ottima **locale**.

Nella formulazione di un problema di ottimizzazione un aspetto assai critico e rilevante è costituito dalla corretta definizione dell'insieme delle soluzioni ammissibili; è necessario delimitare questo sottoinsieme dello spazio delle soluzioni, senza escludere punti dello spazio che possano corrispondere ad una soluzione ammissibile e dunque potenzialmente ottima, e al tempo stesso cercando di ridurre al massimo l'insieme entro cui eseguire la ricerca delle soluzioni, per rendere più efficiente e meno oneroso, in termini computazionali, il procedimento di calcolo adottato per individuare le soluzioni ottime. In generale quindi l'insieme A delle soluzioni ammissibili viene definito attraverso un insieme di **vincoli** che limitano e circoscrivono l'insieme entro cui possono assumere valori le variabili del problema.

In particolare tra i problemi di ottimizzazione rivestono notevole importanza i **problemi di programmazione matematica** dove i vincoli che circoscrivono l'insieme A delle soluzioni ammissibili sono costituiti da un certo numero, anche molto ampio ma finito, di equazioni o disequazioni. In generale, quindi, un problema di programmazione matematica viene definito come segue:

$$\begin{aligned} & min\ f(x) \\ & g_i(x)\ \geq\ 0 \quad \forall i \in \{1,..,m\} \\ & h_j(x)\ =\ 0 \quad \forall i \in \{1,..,p\} \end{aligned} \tag{2.1}$$

2.1.2 Programmazione lineare

La programmazione lineare è quella branca della Ricerca Operativa che si occupa di studiare algoritmi di risoluzione per problemi di ottimizzazione lineari. Un problema è detto **lineare** se sia la funzione obbiettivo sia i vincoli sono funzioni lineari. Questo significa che la funzione obbiettivo può essere scritto come:

$$f\ =\ \sum_{i=1}^{N_v} c_i\ x_i\ =\ c^T\ x \tag{2.2}$$

avendo indicato con:

- N_v: il numero delle variabili che descrivono il problema;
- c: il vettore colonna dei coefficienti c_i della funzione obbiettivo;
- x: il vettore colonna delle variabili x_i;
- infine la T all'esponente indica l'operatore di trasposizione.

Esistono tre grandi classi di problemi lineari:

1. **Problemi lineari continui (LP)**: sono problemi che presentano al loro interno solo variabili continue, cioè variabili che possono assumere con continuità tutti i valori contenuti all'interno del loro dominio di esistenza. Per questi tipi di problemi il principale algoritmo di risoluzione è detto *Algoritmo del Simplesso*;

2. **Problemi lineari interi (ILP)**: sono problemi lineari che presentano al loro interno solo variabili intere, cioè variabili che possono assumere solo i valori interi contenuti all'interno del loro dominio di esistenza. Per questa classe di problemi esiste un algoritmo di risoluzione molto importante, chiamato *Branch and Bound*;

3. **Problemi lineari misto-interi (MILP)**: sono problemi che al loro interno presentano sia variabili intere che continue. L'algoritmo di risoluzione principale è detto *Branch and Bound*.

2.2 Programmazione Lineare Intera

La Programmazione Lineare Intera tratta il problema della minimizzazione (o massimizzazione) di una funzione lineare di più variabili, soggetta a vincoli di uguaglianza e disuguaglianza lineari ed alla restrizione che tutte variabili possano assumere **solo valori interi**. Si tratta dunque di problemi

del tipo:

$$\begin{aligned} & min \sum_{j=1}^{n} c_j x_j \\ & \sum_{j=1}^{n} a_{i,j}\, x_j \;=\; b_i \quad \forall i \in \{1, ..., m\} \\ & x_j \;\geq\; 0 \quad \forall j \in \{1, ..., n\} \\ & x_j \;\in \mathbb{N} \quad \forall j \in \{1, ..., n\} \end{aligned} \tag{2.3}$$

La stessa formulazione vale nel caso si voglia massimizzare la funzione obbiettivo.

Moltissimi problemi reali possono essere rappresentati da modelli di Programmazione Lineare Intera; tipicamente si tratta di problemi in cui le variabili di decisione rappresentano quantità indivisibili oppure sono problemi caratterizzati dalla necessità di scegliere tra un numero finito di alternative diverse. In quest'ultimo caso, in particolare, si avranno problemi di **Programmazione Lineare Zero-Uno** o Binaria, cioè problemi in cui le variabili sono binarie e perciò possono assumere solo i valori 0 o 1.

2.2.1 Tecniche di modellazione

In questa sezione vengono presentate alcune tecniche di modellazione che facilitano la formulazione di problemi di programmazione lineare intera.

Variabili binarie

Le variabili binarie vengono usate quando si vuole rappresentare un evento mediante la scelta tra 2 alternative disponibili, cioè:

$$x = \begin{cases} 1, & \text{se l'evento si verifica,} \\ 0, & \text{altrimenti.} \end{cases} \tag{2.4}$$

Vincoli logici

Spesso, nella formulazione di un problema di ottimizzazione è necessario imporre dei vincoli logici alle variabili decisionali, che vorremmo soddisfare, per trovare la soluzione ottima del problema preso in esame. Consideriamo il seguente vincolo:

$$f(x_1, ..., x_n) \ \leq \ b \tag{2.5}$$

Introduciamo la variabile binaria y tale che:

$$y = \begin{cases} 0, & \text{se il vincolo è soddisfatto,} \\ 1, & \text{altrimenti.} \end{cases} \tag{2.6}$$

Quindi si può riscrivere i vincolo nel seguente modo:

$$f(x_1, ..., x_n) - By \ \leq \ b \tag{2.7}$$

dove B è una costante.

Vincoli logici alternativi

Consideriamo la situazione in cui si hanno vincoli logici alternativi, in cui almeno uno di questi, non necessariamente entrambi, deve essere soddisfatto:

$$\begin{aligned} f_1(x_1, ..., x_n) \ &\leq \ b_1 \\ f_2(x_1, ..., x_n) \ &\leq \ b_2 \end{aligned} \tag{2.8}$$

Questa restrizione può essere modellata combinando la tecnica appena introdotta con un vincolo a scelta multipla come segue:

$$\begin{aligned} &f_1(x_1, ..., x_n) - B_1 y_1 \ \leq \ b_1 \\ &f_2(x_1, ..., x_n) - B_2 y_2 \ \leq \ b_2 \\ &y_1 + y_2 \ \leq \ 1 \\ &y_1, y_2 \ \textit{binarie} \end{aligned} \tag{2.9}$$

Vincoli logici condizionali

Questa classe di vincoli è caratterizzata dal fatto che tutti i vincoli sono dipendenti tra loro, cioè, in altre parole, dato un vincolo A esso implica un altro vincolo B. Matematicamente possiamo scrivere ciò:

$$f_1(x_1, ..., x_n) \ > \ b_1 \implies f_2(x_1, ..., x_n) \ \leq \ b_2 \tag{2.10}$$

Poiché questa implicazione non è soddisfatta quando entrambi valgono, il vincolo condizionale è logicamente equivalente ai vincoli alternativi, e quindi di può riscrivere nel seguente modo:

$$f_1(x_1, ..., x_n) \ \leq \ b_1 \ and/or \ f_2(x_1, ..., x_n) \ \leq \ b_2 \tag{2.11}$$

dove almeno uno deve essere soddisfatto.

2.3 Algoritmi di ILP

Per la soluzione di problemi ILP non esistono metodi universalmente efficienti. Molto spesso è necessario utilizzare algoritmi *ad hoc* che siano in grado di sfruttare la particolare struttura del problema. Esistono però dei metodi applicabili ad una larga classe di PLI, e questi si possono dividere in 3 principali categorie:

1. *Algoritmi esatti*: questi tipi di algoritmi garantiscono di trovare una soluzione ottima, ma potrebbero richiedere un numero esponenziale di iterazioni. In questa categoria sono inclusi il metodo dei Piani di Taglio e l'algoritmo Branch and Bound;

2. *Algoritmi di approssimazione*: sono algoritmi che garantiscono sulla qualità della soluzione trovata;

3. *Algoritmi euristici*: sono algoritmi nei quali non si garantisce la qualità della soluzione.

Consideriamo un problema di Programmazione Lineare Intera:

$$\begin{cases} min\ c^T x \\ Ax\ \geq\ b \\ x\ \geq\ 0,\ con\ x\ intero. \end{cases} \tag{2.12}$$

assumendo che la regione ammissibile di tale problema sia costituita da un numero finito di punti.

Di seguito verranno spiegati, dapprima due tecniche più banali che però non sono sempre possibili da applicare nella pratica, e successivamente i principali metodi per la risoluzione di problemi lineari interi come il metodo dei Piani di Taglio e l'algoritmo del Branch and Bound.

2.3.1 Enumerazione totale

In un problema di Programmazione Lineare binaria una soluzione ottima può essere determinata teoricamente **enumerando** tutte le possibili soluzioni ammissibili. In linea di principio, sarebbe possibile enumerare tutti i vettori binari a n componenti e determinare quelli ammissibili selezionando quelli corrispondenti al valore più basso della funzione obbiettivo. Purtroppo però, ciò è possibile solo per problemi di dimensioni molto ridotte in quanto c'è una crescita esponenziale dei vettori da esaminare con il numero delle variabili del problema.

Quindi, l'enumerazione totale non rappresenta una tecnica praticabile se non in rari casi di problemi con un numero molto basso di variabili.

2.3.2 Soluzione approssimata per arrotondamento

Questa strategia si basa sull'idea di risolvere il **problema rilassato**, cioè il rilassamento lineare, e poi approssimare la soluzione non intera al punto a componenti intere *più vicino*.

Questa strategia sembrerebbe avere senso soprattutto se ci si aspetta che le variabili assumano valori abbastanza grandi, e quindi non costituisce un buon metodo nel caso in cui le variabili del problema assumano valori relativamente piccoli. Infatti nel caso di problemi di Programmazione Lineare Intera Binaria le variabili indicano scelte alternative e ogni arrotondamento può essere totalmente privo di senso. Inoltre anche nel caso in cui si possa trovare un punto ammissibile vicino al punto di ottimo del rilassamento lineare, in realtà, può accadere che tale punto sia molto lontano dalla soluzione ottima intera.

2.3.3 Metodi dei piani di taglio

Si consideri un problema di programmazione lineare intera del tipo:

$$\begin{cases} min\ c^T x \\ Ax\ =\ b \\ x\ \geq\ 0\ con\ x\ intera \end{cases} \tag{2.13}$$

e il corrispettivo problema rilassato:

$$\begin{cases} min\ c^T x \\ Ax\ =\ b \\ x\ \geq\ 0 \end{cases} \tag{2.14}$$

L'idea principale nei metodi dei piani di taglio è risolvere il problema sopra definito trovando la soluzione di una sequenza di problemi di program-

mazione lineare intera. Ovvero, prima di tutto, viene risolto il rilassamento del problema originario trovando la sua soluzione ottima x^*; successivamente se questa soluzione è intera allora è anche soluzione ottima del problema originario (2.13), altrimenti viene aggiunto un vincolo lineare al problema rilassato in modo tale che questa soluzione non sia tra quelle ammissibili per quello rilassato. L'efficienza di questo metodo dipende nella scelta del vincolo che viene aggiunto per escludere x^* come soluzione del problema rilassato.

Uno dei più famosi piani di taglio è chiamato *piano di taglio di Gomory*.

2.3.4 Branch and Bound

Il Branch and Bound (BB), invece, è una metodologia di ricerca della soluzione ottima che effettua un'esplorazione parziale dell'insieme delle soluzioni ammissibili. In particolare la funzione obbiettivo viene calcolata per un sottoinsieme di cardinalità abbastanza piccola delle soluzioni ammissibili con la proprietà di contenere almeno una soluzione ottima.

Facciamo riferimento al generico problema di Programmazione Lineare Intera definito in (2.12). Tale problema può essere riscritto nella forma:

$$\begin{cases} min\ c^T x \\ x \in S \end{cases} \tag{2.15}$$

dove $S = \{x \in \Re^n \mid Ax \geq b,\ x \geq 0,\ x\ intero\}$ è l'insieme ammissibile che ha cardinalità finita e quindi il problema non può essere illimitato inferiormente. Indicheremo con x^* l'ottimo del problema e con $z^* = c^T x^*$ il suo valore ottimo corrispondente.

La strategia è che alla base della tecnica del Branch and Bound vi è la decomposizione del problema originario in **sotto-problemi**. Questo vie-

ne realizzato effettuando una partizione dell'insieme ammissibile S in una famiglia di sottoinsiemi $\{S_1, ..., S_q\}$ con $q \geq 2$.

A seguito di questa partizione si possono considerare q sotto-problemi che indichiamo con $Prob^{(i)}$ del tipo:

$$\begin{cases} min\ c^T x \\ x \in S_i \quad con\ i \in \{1, ..., q\} \end{cases} \tag{2.16}$$

Ora, se $x^{(i)}$ è l'ottimo dell'i-esimo sotto-problema $Prob^{(i)}$ e $z^{(i)} = c^T x^{(i)}$ il valore ottimo corrispondente, si ha che la soluzione ottima del problema originario è data dalla $x^{(i)}$ corrispondente al minimo tra i valori $z^{(i)}$. Identificando il problema originario con il $Prob^{(0)}$ e il suo insieme ammissibile con S_0, si può dire che i nuovi problemi generati sono figli del problema padre. Se un sotto-problema $Prob^{(i)}$ dovesse risultare, a sua volta, di difficile soluzione si può partizionare ulteriormente l'insieme S_i producendo nuovi sotto-problemi e iterando la procedura finché il problema originale non risulti decomposto in problemi elementari di facile risoluzione. Questa generazione progressiva di processi figli produce un **albero di enumerazione**.

In generale, però, non è detto che un sotto-problema sia più facile che risolvere il problema originario ed è per questo motivo che invece della soluzione esatta del problema $Prob^{(i)}$ si preferisce calcolare una limitazione inferiore detta *lower bound* L_i di $z^{(i)}$, cioè un valore $L_i \leq z^{(i)}$. Tale valore viene poi confrontato con il miglior valore della funzione obbiettivo trovato fino a quel momento che viene detto *valore ottimo corrente* che indichiamo con $\tilde{z}$. Se L_i risulta non inferiore a quello del valore ottimo corrente allora nell'insieme S_i non esiste un punto in cui la funzione obbiettivo abbia un valore migliore di $\tilde{z}$. Questo permette di sospendere l'analisi del sotto-problema $Prob^{(i)}$ senza risolverlo e non considerandolo ulteriormente.

La tecnica del **Branch and Bound** è caratterizzata da 2 **fasi** principali:

I) FASE DI BOUNDING: fase nel quale si calcola i lower bound dei sotto-problemi per capire se è necessario scartare o no il sotto-problema considerato;

II) FASE DI BRANCHING: fase nella quale vengono venerati i sotto-problemi e quindi l'albero di enumerazione.

Questo tipo di algoritmo sarà tanto più efficiente quanto migliori saranno i valori del lower bound, ed a loro volta tali valori approssimeranno tanto meglio il valore ottimo del sotto-problema quanto più efficace sarà stata la decomposizione del problema originario. Di conseguenza l'efficienza del metodo Branch and Bound dipende dalla qualità delle strategie che ne caratterizzano la struttura, che sono:

a) Strategia di Bounding: serve per determinare i lower bound, cioè per calcolare un valore che approssimi per difetto il valore dei sotto-problemi;

b) Strategia di Branching: viene utilizzata per determinare la partizione dell'insieme delle soluzioni ammissibili di un sotto-problema;

c) Strategia per la scelta del sotto-problema da esaminare: ovvero come decidere, ad ogni iterazione, quale sotto-problema selezionare dalla lista dei problemi aperti (insieme dei sotto-problemi che devono ancora essere analizzati)

Strategia di Bounding

La principale strategia per il calcolo del lower bound è il **rilassamento lineare**. Questa tecnica consiste nel considerare il rilassamento lineare del problema $Prob^{(i)}$, cioè il problema ottenuto eliminando il vincolo di interezza.

E quindi si si trovano i valori dei lower bound mediante la soluzione ottima $\bar{x}^{(i)}$ del rilassamento lineare, cioè: $L_i = c^T \bar{x}^{(i)}$. Infatti, il valore ottimo del problema rilassato è sempre minore o uguale al valore ottimo del problema intero $Prob^{(i)}$ ed inoltre se la soluzione ottima del problema rilassato è intera, allora essa è anche soluzione ottima del problema $Prob^{(i)}$.

Strategia di Branching

Vediamo una semplice strategia per separare un generico problema $Prob^{(i)}$ in due sotto-problemi.

Supponiamo di aver risolto il rilassamento lineare di $Prob^{(i)}$, e sia $\bar{x}^{(i)}$ la sua soluzione ottima e $L_i = c^T \bar{x}^{(i)}$ il corrispondente valore ottimo, come detto sopra. Si possono verificare le seguenti situazioni:

- se $\bar{x}^{(i)}$ ha tutte le componenti intere allora è soluzioni di $Prob^{(i)}$, e quindi il problema va chiuso;
- se $L_i \geq \tilde{z}$ il problema non può dare origine ad un punto in cui il valore della funzione obbiettivo sia migliore di quello corrente e quindi il problema va chiuso;
- se nessuno dei casi precedenti si è verificato e quindi $L_i \leq \tilde{z}$, è necessario dividere il problema in due sotto-problemi come segue. Sia $\bar{x}_k^{(i)}$ una componente non intera del vettore $\bar{x}^{(i)}$, indichiamo con α_k la sua parte intera inferiore e con β_k la sua parte intera superiore; separiamo il problema nei due seguenti sotto-problemi:

$$Prob^{(i,1)} = \begin{cases} min\ c^T x \\ x \in S_i \\ x_k \leq \alpha_k \end{cases} \tag{2.17}$$

$$Prob^{(i,2)} = \begin{cases} min\ c^T x \\ x\ \in\ S_i \\ x_k\ \geq\ \beta_k \end{cases} \tag{2.18}$$

È facile verificare che l'unione delle regioni ammissibili di questi due problemi coincide con la regione ammissibile S_i.

Strategia per la scelta del sotto-problema da esaminare

Esistono diverse strategie di scelta, le più usate sono le seguenti:

1. Scelta del sotto-problema con il **minimo lower bound**: questa tecnica ha lo sopo di esaminare per primi quei sotto-problemi in cui è più probabile trovare una soluzione ottima;

2. Scelta con criterio di **priorità LIFO** (Last In First Out): in questo caso i sotto-problemi da esaminare sono gestiti dalla procedura secondo lo schema a pila (stack), e quindi il sotto-problema scelto è quello che da meno tempo si trova nell'insieme dei sotto-problemi da analizzare;

3. Scelta con criterio di **priorità FIFO** (First In First Out): in questo caso, invece, i sotto-problemi da esaminare sono gestiti secondo lo schema a coda, e quindi il sotto-problema scelto è quello che da più tempo si trova nell'insieme dei sotto-problemi da analizzare.

Capitolo 3

Modellazione del problema

3.1 Introduzione al problema

In questo capitolo si vuole spiegare le specifiche del problema preso in esame, introducendone prima i concetti principali e in seguito i dettagli implementativi della soluzione finale.

Il problema affrontato riguarda la **programmazione dei turni**, che è da sempre un'attività molto importante in qualunque settore lavorativo e in particolare in quello ospedaliero, per la gestione della sanità.

Fissato il numero del personale medico e il periodo di tempo in cui si vuole organizzare l'orario di lavoro, l'obbiettivo è quello di ottimizzare la gestione del cambio dei turni all'interno di un ospedale, cercando di soddisfare tutti i vincoli imposti dal problema. Per trovare la soluzione ottima si è modellato il seguente problema come un **problema di programmazione lineare intera** in cui le variabili sono obbligate ad assumere solo valori interi, e la maggior parte di esse sono binarie. Il programma riceve in input le seguenti tipologie di informazioni:

1. *Informazioni generali*: dati che sono globali a tutte le settimane che

compongono il periodo di tempo fissato, come per esempio i tipi di turni disponibili o il genere di contratto scelto;

2. *Informazioni giornaliere*: dati specifici di ogni giorno, come le richieste di un singolo lavoratore per un dato giorno;

3. *Informazioni storiche*: dati che riguardano la programmazione dei turni nelle settimane precedenti al periodo selezionato.

In aggiunta a queste informazioni, per costruire il modello è stato necessario imporre alcuni vincoli, sia soft che hard, con lo scopo di soddisfarne un numero più alto possibile, e desiderando trovare una soluzione ammissibile al problema tale da soddisfare al meglio la maggior parte dei vincoli. Nel prossimo capitolo verranno poi presentati alcuni esperimenti per analizzare l'efficienza e le prestazioni del modello al variare del numero del personale e delle settimane che compongono il periodo di pianificazione.

Di seguito verrà illustrato nel dettaglio il modello relativo al turnaggio degli infermieri, tenendo conto che tale modello non è restrittivo al solo personale infermieristico ma potrebbe essere tranquillamente applicato a casi simili come per i medici o altre figure di rilievo all'interno di un ospedale o in realtà lavorative diverse.

3.2 Definizione del problema

Il problema consiste nel definire la programmazione dei turni lavorativi di un numero fissato di infermieri nell'arco temporale scelto, avendo a disposizione varie tipologie di informazioni e con l'obbiettivo di soddisfare il maggior numero di vincoli possibili.

Sulla base di ciò, viene creato un modello del problema, usando sia variabili intere che intere binarie, con lo scopo di minimizzare il costo complessivo delle penalità dei vari vincoli, ovvero ciò che si deve pagare nel caso uno o più vincoli non siano soddisfatti.

3.2.1 Informazioni generali

Le informazioni generali presenti riguardano i dati che sono comuni a tutte le settimane che compongono il periodo di pianificazione scelto, e sono rappresentate dai seguenti elementi:

- *Periodo complessivo*: il numero di settimane di cui è composto l'intervallo di tempo fissato;
- *Tipi di turni*: ogni giorno è possibile svolgere varie tipologie di turno che corrispondono a diversi momenti della giornata. Nel problema in questione vi sono 3 tipologie di turno: mattina, pomeriggio e notte. Per ognuno di questi sono dati:
  - il numero minimo di assegnamenti consecutivi possibili, cioè il numero minimo totale di turni che ogni infermiere, secondo il proprio contratto, deve svolgere;
  - una matrice delle successioni di turni proibiti, la quale, ad ogni infermiere che ha svolto un turno di notte, vieta di lavorare nel giorno successivo. Quindi le successioni di turni consentite sono: M-M, M-P, M-N, P-M, P-P, P-N, invece quella proibita è: N-M, N-P, N-N.
- *Infermieri*: per ogni infermiere si conosce il nome, l'identificativo, il tipo di contratto scelto e l'insieme delle competenze possedute;

- *Competenze*: lista di competenze che possono possedere gli infermieri. In particolare le competenze prese in considerazione nel seguente problema sono:
  - capo infermiere;
  - infermiere regolare;
  - apprendista.

  Ovviamente se un infermiere è un apprendista del mestiere vige la regola che non può possedere le altre due competenze.

- *Tipo di contratto*: vi sono varie tipologie di contratto che ogni infermiere può avere, in questo problema abbiamo considerato i seguenti:
  - full time;
  - part time;
  - a chiamata.

  Ognuno di questi stabilisce dei limiti sulla distribuzione e sul numero di incarichi che si possono assegnare nel lasso di tempo selezionato. In dettaglio, contengono:
  - il numero minimo e il numero massimo di assegnamenti possibili, cioè i limiti dei turni a cui un infermiere può essere assegnato nel periodo totale;
  - il numero minimo e il numero massimo di giorni lavorativi consecutivi, ovvero i limiti del numero di giorni contigui in cui l'infermiere lavora nel periodo complessivo;

  - il numero minimo e il numero massimo di giorni liberi consecutivi, cioè il numero più piccolo e più grande di giorni in cui l'infermiere può non lavorare;
  - il numero massimo di week-ends lavorativi;
  - un valore booleano che rappresenta la presenza del vincolo Week-end Completo nel contratto dell'infermiere, ovvero se devono essere penalizzati assegnamenti che prevedono che l'infermiere in questione lavori uno e un solo giorno nel fine settimana.

3.2.2 Informazioni giornaliere

Le informazioni che possono variare da un giorno all'altro sono le seguenti:

- *Requisiti necessari*: è dato, per ogni turno, per ogni competenza e per ogni giorno della settimana il numero minimo e ottimo di infermieri necessari per compiere il lavoro;
- *Richieste dell'infermiere*: ogni infermiere può esprimere il desiderio di non lavorare un dato giorno in uno specifico turno per problemi o impegni personali. Nel caso nella richiesta fosse presente la parola speciale "Any" nel campo del turno, significa che l'infermiere vorrebbe non lavorare per l'intera giornata e quindi avere un giorno libero.

3.2.3 Informazioni storiche

Oltre a quanto visto in precedenza, è anche necessario tener conto anche delle informazioni storiche, che potrebbero essere presenti per il periodo in cui si vuole calcolare l'orario del personale, per determinare una soluzione accurata del problema. In particolare queste sono rappresentate dalle con-

dizioni a contorno, che vengono usate per controllare i vincoli riguardo agli assegnamenti di infermieri in giorni consecutivi, ed includono:

- i turni lavorati nell'ultimo giorno della settimana precedente al periodo scelto;
- numero di turni di lavoro consecutivi dello stesso tipo, e il numero di turni di lavoro consecutivi in generale;
- numero di giorni liberi consecutivi.

3.2.4 Esempio

Per avere una visione più chiara delle informazioni di cui abbiamo bisogno per fare la programmazione dell'orario ospedaliero si illustra di seguito un esempio.

Supponiamo di voler pianificare l'orario lavorativo in un periodo di tempo pari a 1 settimana avendo a disposizione 10 infermieri. Come abbiamo detto sopra, ogni infermiere può coprire tre tipi di turni: mattina, pomeriggio o notte e inoltre, ad ognuno di essi, è associato una tipologia di contratto tra quelli detti sopra.

A seconda del tipo di contratto che ogni infermiere ha, possono cambiare le quantità dei parametri che si vogliono specificare.

3.3 Vincoli

In questo paragrafo verranno spiegati i vincoli relativi al problema di ottimizzazione in questione, alcuni dei quali sono *hard* ovvero devono essere assolutamente rispettati, mentre gli altri sono *soft*, ovvero si permette di violarli pagando però una penalità nella funzione obbiettivo.

Per maggior chiarezza si riporta di seguito la notazione usata per spiegare i vincoli dal punto di vista matematico:

- I: insieme di tutti gli infermieri;
- T: insieme dei tipi di turni;
- G: insieme dei giorni della settimana del periodo preso in considerazione;
- C: insieme delle competenze degli infermieri;
- I_c: insieme degli infermieri che hanno la competenza c;
- S_{VT}: elenco di successioni vietate di turni;
- I_{cw}: lista di infermieri che hanno il week-end completo, cioè devono lavorare sia sabato che domenica;
- G_S: lista di tutti i sabati presenti nel periodo di pianificazione fissato;
- $minConsLav_i$: numero minimo di giorni consecutivi in cui l'infermiere i deve lavorare secondo il contratto scelto;
- $maxConsLav_i$: numero massimo di giorni consecutivi in cui l'infermiere può lavorare secondo il contratto scelto;
- $minConsLib_i$: numero minimo di giorni consecutivi in cui l'infermiere i può avere giorno libero secondo il contratto scelto;
- $maxConsLib_i$: numero massimo di giorni consecutivi in cui l'infermiere i può avere giorno libero secondo il contratto scelto;
- $hTurniCons_i$: numero di turni consecutivi in cui l'infermiere i ha lavorato nei giorni precedenti al periodo selezionato;

- $hLibCons_i$: numero di turni consecutivi in cui l'infermiere i ha avuto giorno libero nei giorni precedenti al periodo selezionato.

3.3.1 Assegnamento di un singolo giorno

Il vincolo *hard* riguardo agli assegnamenti giornalieri impone che ogni infermiere possa fare al massimo un turno al giorno. Quindi, in altre parole, la somma dei turni fatti da ogni infermiere deve essere minore o uguale ad 1. Perciò si può scrivere matematicamente:

$$\sum_{t \in T} a_{i,t,g} \leq 1 \quad \forall i \in I \quad \forall g \in G \tag{3.1}$$

dove $a_{i,t,g}$ è la **variabile binaria d'assegnamento** definita come segue:

$$a_{i,t,g} = \begin{cases} 1, & \text{se l'infermiere i è assegnato al turno t il giorno g,} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.2}$$

Si riporta il codice Python relativo al vincolo:

```
model.addConstrs((quicksum(a[i, t, g] for t in T) <= 1
  for i in I for g in G))
```

3.3.2 Livelli minimi di personale

Il seguente vincolo *hard* impone che il numero di infermieri per ogni turno e per ciascuna competenza sia almeno pari al requisito minimo.
Matematicamente corrisponde a scrivere la seguente formula:

$$\sum_{i \in I_c} a_{i,t,g} \geq min_{t,c,g} \quad \forall t \in T \quad \forall c \in C \quad \forall g \in G \tag{3.3}$$

dove $min_{t,c,g}$ è il numero minimo di infermieri necessari per la data competenza c a ricoprire il turno t, nel giorno g.

Quindi per ogni turno, per ogni giorno e per ogni competenza viene calcolato il numero di infermieri obbligando tale numero ad essere maggiore o uguale al numero minimo necessario, sempre per ogni turno, competenza e giorno.

Il codice Python corrispondente è il seguente:

```
model.addConstrs((quicksum(a[i, t, g] for i in I[c])
      >= min[t, c, g] for t in T
               for g in G
               for c in C))
```

3.3.3 Successioni di turni valide

Infine, l'ultimo vincolo *hard* impone che gli assegnamenti dei turni infermieristici in due giorni consecutivi devono appartenere alla successione legale stabilita. Matematicamente si può scrivere:

$$\begin{aligned} & a_{i,t_a,g} + a_{i,t_b,g+1} \ \leq \ 1 \\ & \forall i \in I \quad \forall g \in \{0, ..., |G| - 2\} \quad \forall \ t_a, t_b \in T \quad \textit{tale che } (t_a, t_b) \in S_{VT} \end{aligned} \tag{3.4}$$

Cioè, in altre parole, un infermiere non può lavorare nei turni t_a e t_b in due giorni consecutivi della settimana se questa coppia di turni (t_a, t_b) appartiene alla successione di turni proibita, dove ricordiamo che le successioni di turni proibite sono le seguenti:

- mattina-notte;
- pomeriggio-notte;
- notte-notte.

Il codice Python corrispondente è il seguente:

```
model.addConstrs(((a[i, ta, g] + a[i, tb, g+1] <= 1)
                  for i in I
                  for g in range(len(G)-1)
                  for ta in T
                  for tb in T
                  if (ta, tb) in Svt)
```

3.3.4 Personale insufficiente per una copertura ottimale

Il seguente vincolo *soft* ha l'obbiettivo di fare in modo che il numero di infermieri per ogni turno e per ciascuna competenza sia il più vicino possibile al requisito ottimale. Gli infermieri extra, cioè quelli al di sopra del valore ottimale, contrariamente a quelli mancanti, non sono considerati nel costo. Tale vincolo si può scrivere matematicamente nel seguente modo:

$$\begin{aligned} s^{(1)}_{t,c,g} &\geq 0 \quad \forall t \in T \quad \forall c \in C \quad \forall g \in G \\ s^{(1)}_{t,c,g} &\geq opt_{t,c,g} - \sum_{i \in I_c} a_{i,t,g} \quad \forall c \in C \quad \forall t \in T \quad \forall g \in G \\ S^{(1)} &= \sum_{t \in T} \sum_{c \in C} \sum_{g \in G} s^{(1)}_{t,c,g} \end{aligned} \tag{3.5}$$

dove:

- $s^{(1)}_{t,c,g}$: è una variabile intera che, dati turno, competenza e giorno, rappresenta la penalità del vincolo;
- $opt_{t,c,g}$: numero ottimo di infermieri dati turno, competenza e giorno;
- $S^{(1)}$: è la penalità complessiva che, moltiplicata per il peso, deve essere pagata come costo.

Il codice Python corrispondente è il seguente:

```
s1 = model.addVars(T, C, G, vtype=GRB.INTEGER)

model.addConstrs(s1[t, c, g] >= 0
        for t in T for c in C for g in G)

model.addConstrs((s1[t, c, g] >= (opt[t, c, g]
  - quicksum(a[i, t, g] for i in I[c]))
           for c in C for t in T for g in G)

S1 = quicksum(s1[t, c, g]
          for t in T for c in C for g in G)
```

Cioè, l'idea che sta alla base di questo vincolo è confrontare la quantità di infermieri assegnati con il valore ottimo per ogni turno e per ogni competenza. Quindi vi sono due casi possibili:

1. se $\sum_{i \in I_c} a_{i,t,g} \geq opt_{t,c,g} \longrightarrow s^{(1)}_{t,c,g} = 0$
2. se $\sum_{i \in I_c} a_{i,t,g} < opt_{t,c,g} \longrightarrow s^{(1)}_{t,c,g} = opt_{t,c,g} - \sum_{i \in I_c} a_{i,t,g}$

E quindi chiamando $z = opt_{t,c,g} - \sum_{i \in I_c} a_{i,t,g}$ segue:

$$\begin{aligned} z \leq 0 &\longrightarrow s^{(1)}_{t,c,g} = 0 \\ z > 0 &\longrightarrow s^{(1)}_{t,c,g} = z \end{aligned} \tag{3.6}$$

SPIEGARE MEGLIO!!

3.3.5 Assegnamenti consecutivi

Tale vincolo *soft* ha lo scopo di far rispettare il numero minimo e massimo di assegnamenti consecutivi che è possibile fare. Sia nel caso del limite minimo che nel caso del limite massimo è necessario tener conto, se esistono, delle assegnazioni fatte nel periodo precedente a quello selezionato.

Limite minimo

L'assegnazione di un infermiere in più giorni lavorativi consecutivi deve essere maggiore o uguale al numero minimo previsto nel contratto di quest'ultimo. Per realizzare questo vincolo l'idea è quella di confrontare i giorni in cui l'infermiere ha lavorato con quelli in cui avrebbe dovuto lavorare per soddisfare tale vincolo. Quindi vengono creati due tipi di variabili, $L_{i,g}$ e $L^R_{i,g}$,

Matematicamente si può scrivere:

$$\begin{aligned}
L_{i,g} &\geq a_{i,t,g} \quad \forall i \in I \quad \forall t \in T \quad \forall g \in G \\
L_{i,g} &\leq \sum_{t \in T} a_{i,t,g} + a_{i,t_{notte},g-1} \quad \forall i \in I \quad \forall g \in \{1, ..., |G| - 1\} \\
L_{i,g+1} &\geq a_{i,t_{notte},g} \quad \forall i \in I \quad \forall g \in \{0, ..., |G| - 2\} \\
L^R_{i,g} &\geq L_{i,g} \quad \forall i \in I \quad \forall g \in G \\
L^R_{i,g+n} &\geq L^R_{i,g} - L^R_{i,g-1} \quad \forall i \in I \quad \forall n \in \{0, ..., minConsLav_i - 1\} \quad \forall g \in \{1, ..., |G| - n - 1\} \\
L^R_{i,n} &\geq L^R_{i,0} \quad \forall i \in I \quad \forall n \in \{1, ..., minConsLav_i - hTurniCons_i - 1\} \\
L^R_{i,0} &= 1 \quad \forall i \in I \; tale \; che \; 0 < hTurniCons_i < minConsLav_i \\
S^{(2)}_{min} &= \sum_{i \in I} \sum_{g \in G} L^R_{i,g} - L_{i,g}
\end{aligned} \tag{3.7}$$

dove:

- $L_{i,g}$: variabile binaria che tiene conto se l'infermiere ha lavorato o meno quel giorno, cioè:

$$L_{i,g} = \begin{cases} 1, & \text{se l'infermiere i ha lavorato il giorno g,} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.8}$$

- $L^R_{i,g}$: variabile binaria che ci dice se l'infermiere avrebbe dovuto lavorare

quel giorno secondo l'assegnamento che era stato programmato, cioè:

$$L^R_{i,g} = \begin{cases} 1, & \text{se l'infermiere i avrebbe dovuto lavorare il giorno g,} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.9}$$

- $S^{(2)}_{min}$: penalità complessiva riguardo al limite minimo che, moltiplicata per il peso corrispondente, deve essere pagata.

Il codice Python corrispondente è il seguente:

```
L = model.addVars(I, G, vtype=GRB.BINARY)

LR = model.addVars(I, G, vtype=GRB.BINARY)

model.addConstrs(L[i, g] >= a[i, t, g]
             for i in I
             for t in T
             for g in G)

model.addConstrs(L[i, g] <=
    quicksum(a[i, t, g] for t in T) +
      a[i, 'night', g-1]
             for i in I
             for g in range(1, len(G)))

model.addConstrs(L[i, g+1] >= a[i, 'night', g]
             for i in I
             for g in range(len(G) - 1))

model.addConstrs(LR[i, g] >= L[i, g] for i in I for g in G)


```

```
model.addConstrs(LR[i, g+n] >=
    (LR[i, g] - LR[i, g-1])
             for i in I
             for n in range(0, minConsLav[i])
             for g in range(1, len(G) - n))

model.addConstrs(LR[i, n] >= LR[i, 0]
             for i in I
             for n in range(1, minConsLav[i] - hTurniCons[i]))

model.addConstrs(LR[i, 0] == 1
             for i in I
             if 0 < hTurniCons[i] < minConsLav[i])

S2_min = quicksum(LR[i, g] - L[i, g] for i in I for g in G)
```

Limite massimo

L'assegnazione di un infermiere in più giorni lavorativi consecutivi deve essere minore o uguale al numero massimo previsto nel contratto di quest'ultimo. In termini matematici si ha:

$$
\begin{aligned}
s^{(2)}max_{i,g} &\geq 0 \quad \forall i \in I \quad \forall g \in G \\
s^{(2)}max_{i,g} &\geq \Big(\sum_{j=g-maxConsLav_i}^{g} L_{i,j}\Big) - maxConsLav_i \\
&\qquad \forall i \in I \quad \forall g \in \{maxConsLav_i, ..., |G| - 1\} \\
s^{(2)}max_{i,0} &\geq hTurniCons_i + L_{i,0} - maxConsLav_i \quad \forall i \in I \\
s^{(2)}max_{i,g} &\geq L_{i,g} + s^{(2)}max_{i,g-1} - 1 \quad \forall i \in I \quad \forall g \in \{1, ..., maxConsLav_i - 1\} \\
S^{(2)}_{max} &= \sum_{i \in I} \sum_{g \in G} s^{(2)}max_{i,g}
\end{aligned}
\tag{3.10}
$$

dove:

- $s^{(2)}max_{i,g}$: variabile binaria che indica la penalità per l'infermiere i nel giorno d nel caso il vincolo non sia rispettato;
- $S^{(2)}_{max}$: penalità complessiva riguardo al limite massimo che, moltiplicata per il peso corrispondente, deve essere pagata.

Il codice Python corrispondente è il seguente:

```
s2_max = model.addVars(I, G, vtype=GRB.BINARY)

model.addConstrs(s2_max[i, g] >= 0 for i in I for g in G)

model.addConstrs((s2_max[i, g] >= (quicksum(L[i, j] for j in
    range(g - maxConsLav[i], g+1))
          - maxConsLav[i]))
                      for i in I
                      for g in range(maxConsLav[i], len(G)))

model.addConstrs((s2_max[i, 0] >=
    (hTurniCons[i] + L[i, 0] - maxConsLav[i]))
                       for i in I)

model.addConstrs((s2_max[i, g] >= L[i, g] +
    use_S2_max[i, g-1] - 1)
                 for i in I
                 for g in range(1, maxConsLav[i]))

S2_max = quicksum(s2_max[i, g] for i in I for g in G)
```

3.3.6 Giorni liberi consecutivi

Il seguente vincolo *soft* impone di rispettare il numero minimo e massimo di giorni liberi consecutivi. Anche in questo caso vengono presi in considerazione le condizioni a contorno che riguardano la storia precedente al periodo di pianificazione selezionato. La penalità per ogni giorno extra o mancante viene moltiplicata per il peso corrispondente e aggiunta alla funzione obbiettivo.

Limite minimo

Il numero di giorni liberi consecutivi che ogni infermiere può prendere deve essere maggiore o uguale al numero minimo dei giorni liberi che gli spetterebbero secondo le regole del suo contratto. Matematicamente si ha:

$$\begin{aligned}
O^R_{i,g} &\geq (1 - L_{i,g}) \quad \forall\, i \in I \quad \forall\, g \in G \\
O^R_{i,g+n} &\geq (O^R_{i,g} - O^R_{i,g-1}) \quad \forall\, i \in I \quad \forall\, n \in \{0, ..., minConsLib_i - 1\} \quad \forall\, g \in \{1, ..., |G| - n - 1\} \\
O^R_{i,n} &\geq O^R_{i,0} \quad \forall i \in I \quad \forall n \in \{0, ..., minConsLib_i - hLibCons_i - 1\} \\
O^R_{i,0} &= 1 \quad \forall i \in I \quad tale\ che\ 0 < hLibCons_i < minConsLib_i \\
S^{(3)}_{min} &= \sum_{i \in I} \sum_{g \in G} O^R_{i,g} - (1 - L_{i,g})
\end{aligned} \tag{3.11}$$

dove:

- $O^R_{i,g}$: variabile binaria che tiene conto se l'infermiere avrebbe dovuto avere giorno libero, cioè:

$$O^R_{i,g} = \begin{cases} 1, & \text{se l'infermiere i avrebbe dovuto riposarsi il giorno g,} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.12}$$

- $S^{(3)}_{min}$: penalità complessiva riguardo al limite minimo che, moltiplicata per il peso corrispondente, deve essere pagata.

Il codice Python corrispondente è il seguente:

```
OR = model.addVars(I, G, vtype=GRB.BINARY)

model.addConstrs(OR[i, g] >= (1 - L[i, g])
                 for i in I
                 for g in G)

model.addConstrs(OR[i, g+n] >= (OR[i, g] - OR[i, g-1]])
                 for i in I
                 for n in range(0, minConsLib[i])
                 for g in range(1, len(G) - n))

model.addConstrs(OR[i, n] >= OR[i, 0]
                 for i in I
                 for n in range(0, minConsLib[i] - hLibCons[i]))

model.addConstrs(OR[i, 0] == 1
                 for i in I
                 if 0 < hLibCons[i] < minConsLib[i])

S3_min = quicksum(OR[i, g] - (1 - L[i, g])
                          for i in I
                          for g in G)
```

Limite massimo

Il numero dei giorni liberi consecutivi associato ad ogni infermiere deve essere minore o uguale al numero massimo di giorni liberi consecutivi

disponibili. In termini matematici si ha:

$$
\begin{aligned}
s^{(3)}max_{i,g} &\geq 0 \quad \forall i \in I \quad \forall g \in G \\
s^{(3)}max_{i,g} &\geq (\sum_{j=g-maxConsLib_i}^{g} (1 - L_{i,j}) - maxConsLib_i) \quad \forall i \in I \quad \forall g \in \{maxConsLib_i, ..., |G| - 1\} \\
s^{(3)}max_{i,0} &\geq hLibCons_i + 1 - L_{i,0} - maxConsLib_i \quad \forall i \in I \\
s^{(3)}max_{i,g} &\geq s^{(3)}max_{i,g-1} - L_{i,g} \quad \forall i \in \{1, ..., maxConsLib_i - 1\} \\
S^{(3)}_{max} &= \sum_{i \in I} \sum_{g \in G} s^{(3)}max_{i,g}
\end{aligned}
\tag{3.13}
$$

Il codice Python corrispondente è il seguente:

```
s3_max = model.addVars(I, G, vtype=GRB.BINARY)

model.addConstrs(s3_max[i, g] >= 0 for i in I for g in G)

model.addConstrs((s3_max[i, g] >= (quicksum(1 - L[i, j]
    for j in range(g - maxConsLib[i], g+1)) - maxConsLib[i]))
                    for i in I
                    for g in range(maxConsLib[i], len(G)))

model.addConstrs((s3_max[i, 0] >=
  (hLibCons[i] + 1 -  L[i, 0] - maxConsLib[i]))
                                        for i in I)

model.addConstrs((s3_max[i, g] >=
  s3_max[i, g-1] - L[i, g])
    for i in I for g in range(1, maxConsLib[i]))

S3_max = quicksum(s3_max[i, g] for i in I for g in G)
```

3.3.7 Preferenze infermieri

In questo vincolo *soft* si cerca di soddisfare le richieste presentate da ciascun infermiere nella pianificazione dell'orario lavorativo. In particolare ogni

assegnazione ad un turno indesiderato è penalizzata dal peso corrispondente. Matematicamente si può scrivere come segue:

$$S^{(4)} = \sum_{i \in I} \sum_{t \in T} \sum_{g \in G} a_{i,t,g} \quad tale\ che\ (i,g,t) \in R \tag{3.14}$$

dove:

- $S^{(4)}$ rappresenta la penalità complessiva da pagare nella funzione obbiettivo nel caso il vincolo non sia rispettato;
- R: lista delle richieste fatte dagli infermieri dove sono specificati i giorni e i turni in cui ognuno di loro vorrebbe non lavorare.

Il codice Python corrispondente è il seguente:

```
S4 = quicksum(a[i, t, g] for (i, g, t) in R)
```

3.3.8 Week-end completo

Il seguente vincolo *soft* impone che ogni infermiere che deve ricoprire il proprio turno nel week-end, deve lavorare sia sabato che domenica. Nel caso lavorasse solo uno dei due giorni è necessario pagare una penalità. In formule matematiche:

$$\begin{aligned} L^{!SAB}_{i,g} &\geq L_{i,g} - L_{i,g+1} \quad \forall i \in I_{cw} \quad \forall g \in G_S \\ L^{!DOM}_{i,g} &\geq L_{i,g+1} - L_{i,g} \quad \forall i \in I_{cw} \quad \forall g \in G_S \\ S^{(5)} &= \sum_{i \in I_{cw}} \sum_{g \in G_S} L^{!SAB}_{i,g} + L^{!DOM}_{i,g} \end{aligned} \tag{3.15}$$

dove:

- $L_{i,g}^{!SAB}$ è la variabile binaria che tiene conto dei giorni in cui l'infermiere ha lavorato solo di di sabato, ovvero:

$$L_{i,g}^{!SAB} = \begin{cases} 1, & \text{se l'infermiere i ha lavorato solo il sabato} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.16}$$

- $L_{i,g}^{!DOM}$ è la variabile binaria che indica quali infermieri hanno lavorato solo la domenica, cioè:

$$L_{i,g}^{!DOM} = \begin{cases} 1, & \text{se l'infermiere i ha lavorato solo la domenica} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.17}$$

- $S^{(5)}$: è la penalità da pagare nel caso gli infermieri a cui hanno assegnato il week-end completo abbiano lavorato solo il sabato o solo la domenica.

Il codice Python corrispondente è il seguente:

```
Lsab = model.addVars(Icw, GS, vtype=GRB.BINARY)

Ldom = model.addVars(Icw, GS, vtype=GRB.BINARY)

model.addConstrs(Lsab[i, g] >=
    (L[i, g] - L[i, g+1]) for i in Icw for g in GS)

model.addConstrs(Ldom[i, g] >= L[i, g+1] - L[i, g)
    for i in Icw for g in GS)

S5 = quicksum(Lsab[i, g] + Ldom[i, g]
    for i in Icw for g in GS)
```

3.3.9 Assegnamenti totali

Questo vincolo *soft* ha l'obbiettivo di limitare il numero di assegnamenti totali fatti per ogni infermiere. Cioè, in altre parole, per ogni infermiere il

numero totale dei giorni lavorativi deve essere compreso tra il minimo e il massimo previsti nel suo contratto. Le differenze, in entrambi i casi, vengono moltiplicate per i corrispettivi pesi nella funzione obbiettivo. Tale vincolo si può scrivere matematicamente nel seguente modo:

$$\begin{aligned} s_i^{(6)} &\geq 0 \quad \forall i \in I \\ s_i^{(6)} &\geq minTotLav_i - \sum_{t \in T} \sum_{g \in G} a_{i,t,g} \quad \forall i \in I \\ s_i^{(6)} &\geq \sum_{t \in T} \sum_{g \in G} a_{i,t,g} - maxTotLav_i \quad \forall i \in I \\ S^{(6)} &= \sum_{i \in I} s_i^{(6)} \end{aligned} \tag{3.18}$$

dove:

- $minTotLav_i$: numero minimo di assegnamenti totali;
- $maxTotLav_i$: numero massimo di assegnamenti totali;
- $s_i^{(6)}$: penalità di ogni singolo infermiere;
- $S^{(6)}$: penalità complessiva da pagare nel caso il vincolo non sia soddisfatto.

Il codice Python corrispondente è il seguente:

```
s6 = model.addVars(I, vtype=GRB.INTEGER)

model.addConstrs(s6[i] >= 0 for i in I)

model.addConstrs(s6[i] >= (minTotLav[i] - quicksum(a[i, t, g]
    for t in T for g in G))
      for i in I)

model.addConstrs(s6[i] >=
```

```
  (quicksum(a[i, t, g] for t in T for g in G)
                          - maxTotLav[i]) for i in I)

S6 = quicksum(s6[i] for i in I)
```

3.3.10 Fine settimana totali lavorati

Tale vincolo *soft* impone che per ogni infermiere il numero di week-ends lavorativi deve essere minore o uguale al massimo. Il numero di week-ends lavorati in eccesso viene aggiunto alla funzione obbiettivo moltiplicato per il suo peso, dove un week-end è considerato lavorato se almeno uno dei due giorni l'infermiere ha svolto il suo turno. In termini matematici:

$$\begin{aligned} L^w_{i,g} &\geq a_{i,t,g} \quad \forall t \in T \quad \forall i \in I \quad \forall g \in GS \\ L^w_{i,g} &\geq a_{i,t,g+1} \quad \forall t \in T \quad \forall i \in I \quad \forall g \in GS \\ s^{(7)}_i &\geq 0 \quad \forall i \in I \\ s^{(7)}_i &\geq (\sum_{g \in S} L^w_{i,g}) - maxTotWknd_i \quad \forall i \in I \\ S^{(7)} &= \sum_{i \in I} s^{(7)}_i \end{aligned} \tag{3.19}$$

dove:

- $L^w_{i,g}$ è la variabile binaria definita come segue:

$$L^w_{i,g} = \begin{cases} 1, & \text{se l'infermiere i ha lavorato l'intero week-end} \\ 0, & \text{altrimenti.} \end{cases} \tag{3.20}$$

- $maxTotWknd_i$: numero massimo di week-ends in cui l'infermiere può lavorare;
- $s^{(7)}_i$: variabile intera che rappresenta la penalità che ogni infermiere deve pagare;

- $S^{(7)}$: penalità complessiva da pagare nella funzione obbiettivo.

Il codice Python corrispondente è il seguente:

```
Lw = model.addVars(I, GS, vtype=GRB.BINARY)

model.addConstrs(Lw[i, g] >= a[i, t, g]
                 for t in T
                 for i in I
                 for g in GS)

model.addConstrs(Lw[i, g] >= a[i, t, g+1]
                 for t in T
                 for i in I
                 for g in GS)

s7 = model.addVars(I, vtype=GRB.INTEGER)

model.addConstrs(s7[i] >= 0 for i in I)

model.addConstrs(s7[i] >= quicksum(Lw[i, g] for g in GS)
                 - maxTotWknd[i] for i in I)

S7 = quicksum(s7[i] for i in I)
```

3.4 Funzione obbiettivo

Lo scopo di questo programma è minimizzare la funzione obbiettivo, la quale è definita dalla somma delle penalità dei vincoli soft, moltiplicate dai *pesi* corrispondenti w_i, che servono a calcolare il costo totale da pagare in

caso alcuni di questi non vengano rispettati. Ovvero:

$$Obj = w_1 S^{(1)} + w_{2min} S^{(2)}_{min} + w_{2max} S^{(2)}_{max} + w_3 S^{(3)}_{min} + w_3 S^{(3)}_{max} + w_4 S^{(4)} + w_5 S^{(5)} + w_6 S^{(6)} + w_7 S^{(7)} \tag{3.21}$$

Il codice Python corrispondente è il seguente:

```
obj = w1 * S1 + w2_min * S2_min + w2_max * S2_max +
      w3 * (S3_min + S3_max) + w4 * S1 + w4 * S5 +
        w6 * S6 + w7 * S7

model.setObjective(obj, GRB.MINIMIZE)

model.optimize()
```

Capitolo 4

Risultati sperimentali

4.1 Introduzione

In questo capitolo verranno illustrati una serie di esperimenti eseguiti per testare le prestazioni del modello creato per gestire la pianificazione dei turni ospedalieri. In generale, variando il numero di infermieri o il numero dei giorni che compongono il periodo selezionato, si andranno ad analizzare diversi parametri tra cui:

1. Tempo di calcolo della soluzione ottima;
2. Gap percentuale del problema, ovvero la distanza tra la soluzione ottima corrente trovata e il lower bound;
3. Quali e quanti vincoli vengono violati nei diversi casi.

4.1.1 Tempo di esecuzione

Numero di infermieri fissato

Periodo di tempo fissato

4.1.2 Gap percentuale

4.1.3 Vincoli violati

Capitolo 5

Interfaccia

Capitolo 6

Conclusioni

...

Appendice A

Gurobi

A.1 Introduzione a Gurobi

L'ottimizzatore usato per la pianificazione dell'orario degli infermieri è Gurobi, che è un solutore di ottimizzazione commerciale usato in vari tipi di programmazione: lineare, quadratica, quadratica vincolata, lineare e quadratica intera mista, quadratica intera mista vincolata.

Gurobi, il cui nome deriva dai nomi dei suoi fondatori (Zonghao Gu, Edward Rothberg e Robert Bixby), è stato introdotto nel 2008 e supporta una varietà di linguaggi di programmazione e modellazione, inclusi:

- Interfacce Object-Oriented per C++, Java, .NET e Python;
- Interfacce Matrix-Oriented per C, MATLAB e R;
- Collegamenti a linguaggi di modellazione standard: AIMMS, AMPL, GAMS e MPL.

È possibile capire il ruolo di questo ottimizzatore dallo schema sottostante (Figura A.1).

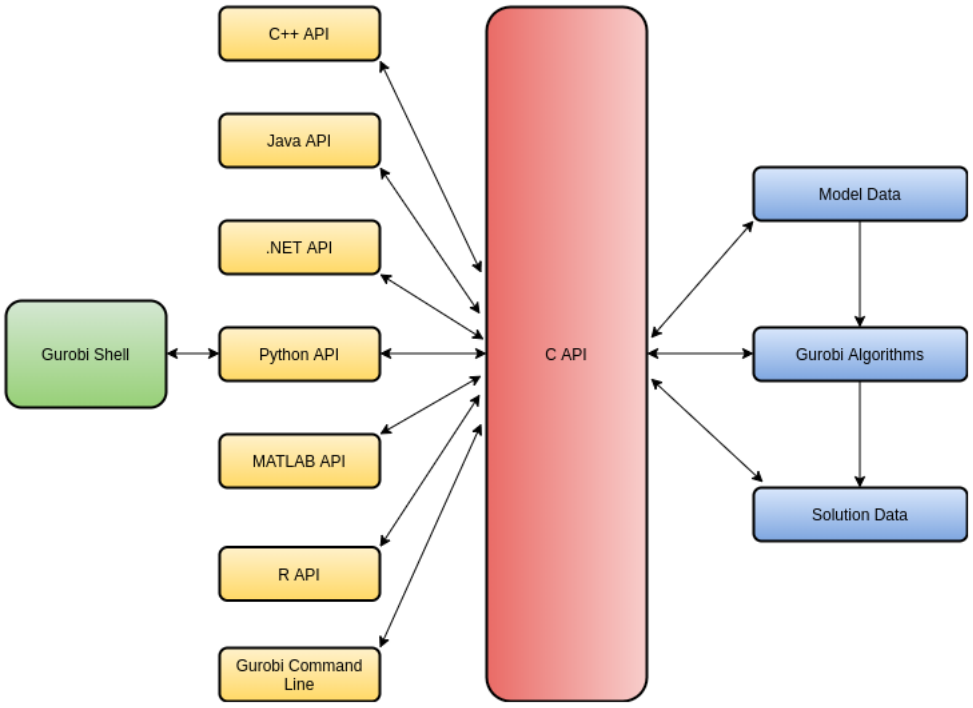

Figura A.1: Gurobi API

A.2 Principali API per Python

In questa sezione vengono spiegate le funzionalità principali di Gurobi per il linguaggio Python, per dare una visione generale al lettore di come è possibile modellare un problema di ottimizzazione usando questo tipo di ottimizzatore.

A.2.1 Modello

A.2.2 Variabili

Le variabili di decisione rappresentano il risultato dell'ottimizzazione. In una soluzione ammissibile, i valori calcolati per le variabili di decisione soddisfano tutti i vincoli del modello. Alcuni di questi vincoli sono associati a

singole variabili, mentre altri rappresentano relazioni tra esse. Ci sono varie tipologie di variabili disponibili tra cui: continue, intere o intere binarie, semi-continue e infine semi-intere.

Variabili continue

Le variabili continue possono assumere qualsiasi valore tra il loro limite inferiore e quello superiore. Nella programmazione matematica, la convenzione è che le variabili sono non negative se non diversamente indicato, quindi se non si forniscono esplicitamente i limiti per una variabile, si dovrebbe presumere che il limite inferiore sia 0 e quello superiore sia ∞. Una variabile con limiti superiori e inferiori uguali ad ∞ vine detta variabile libera. Inoltre le variabili possono violare i loro limite di tolleranza, che in questo caso è `FeasibilityTol`. È possibile ridurre questo valore, ma a causa di errori numerici potrebbe non essere possibile ottenere la precisione desiderata.

Variabili intere

Le variabili intere, in generale, sono maggiormente vincolate rispetto a quelle continue, infatti possono assumere solo valori interi.

????

In particolare i limiti inferiore e superiore di una .... Una soluzione non è considerata ammissibile a meno che tutte le variabili

Variabili binarie

Le variabili binarie sono particolari variabili intere che possono assumere solo valori pari a 0 o ad 1. Ancora una volta, a causa dei limiti dell'aritmetica a precisione finita, le variabili binarie assumeranno spesso valori che non sono

esattamente.... L'entità della violazione di integrità consentita è controllata dal parametro `IntFeasTol`.

Variabili semi-continue e semi-intere

Le variabili semi-continue possono assumere valori pari a 0 o ad un valore compreso tra i limiti inferiore e superiore specificati. Una variabile semi-intera può assumere anche un valore intero.

Entrambi questi tipi di variabili possono violare queste restrizioni, in questo caso la tolleranza pertinente è `IntFeasTol`.

Metodo per aggiungere variabili al modello

Per aggiungere una nuova variabile al modello creato è necessario usare il metodo `addVar()`, il quale se ritorna un valore diverso da 0 indica che si è verificato un problema durante l'aggiunta della variabile. Gli argomenti che possiamo specificare in questo metodo sono:

- `model`: modello al quale la variabile è aggiunta;
- `lb`: lower bound;
- `ub`: upper bound;
- `obj`: coefficiente obbiettivo;
- `vtype`: tipologia della variabile in questione; le opzioni sono:
  - `GRB_CONTINUOUS`;
  - `GRB_BINARY`;
  - `GRB_INTEGER`;
  - `GRB_SEMICONT`;

  - `GRB_SEMINT`.

- `varname`: nome della variabile.

Si riporta un esempio per illustrare l'uso di questo metodo:

```
x = model.addVar(0.0, 1.0, 0.0, GRB.BINARY, name='x')
```

In questo caso il primo e il secondo argomento sono, rispettivamente, il limite inferiore e superiore della variabile x. Il terzo argomento rappresenta il coefficiente di obbiettivo lineare. Successivamente viene specificato il tipo della variabile (binaria), e come ultimo argomento ne viene indicato il nome.

Nel caso si vogliano aggiungere più variabili dello stesso tipo si può usare il metodo `addVars()`.

Si noti che con entrambi questi metodi le nuove variabili non vengono effettivamente aggiunte fino a quando non si aggiorna il modello, non lo si ottimizza o non lo si scrive su disco.

A.2.3 Vincoli

Un vincolo rappresenta una restrizione sui valori che un insieme di variabili può assumere. Vi sono varie tipologie di vincoli disponibili: lineare, quadratico, SOS, generale.

Vincoli lineari

Un vincolo lineare consente di limitare il valore di un'espressione lineare. Un semplice esempio può essere il seguente: $3x + 4y \leq 5z$

Si noti che le API Gurobi orientate alla matrice (C, MATLAB e R) richiedono che il lato destro di un vincolo sia una costante, mentre le API orientate agli oggetti (C++, Java, .NET e Python) consentono arbitrari espressioni lineari in entrambi i membri della comparazione.

Gurobi supporta un set limitato di comparatori: in particolare, è possibile vincolare un'espressione a essere minore o uguale, maggiore o uguale o semplicemente uguale ad un'altra. Invece non vengono supportati i comparatori di minore e maggiore stretto o il comparatore di disuguaglianza.

Vincoli SOS

Un vincolo SOS (insieme ordinato speciale) è un vincolo altamente specializzato che pone restrizioni ai valori che possono assumere le variabili in un determinato **elenco**. Esistono due tipi di vincoli SOS:

- *Vincolo SOS di tipo 1*: in cui al massimo una variabile nell'elenco specificato può assumere un valore diverso da 0;
- *Vincolo SOS di tipo 2*: al massimo due variabili nell'elenco specificato e ordinato possono assumere un valore diverso da 0 e tali variabili devono essere contigue nell'elenco.

Le variabili di un vincolo SOS possono essere continue, intere o binarie.

Quindi un vincolo SOS è descritto da un elenco di variabili e un elenco di pesi corrispondenti che servono per ordinare l'elenco delle variabili in questione.

Vincoli quadratici

Un vincolo quadratico consente di limitare il valore di un'espressione quadratica. Un esempio è il seguente: $3x^2 + 4y^2 + 5z \leq 10$.

I vincoli quadratici sono spesso molto più difficili da soddisfare rispetto a quelli lineari. ma Gurobi può gestire sia quelli convessi che non. Tuttavia gli algoritmi predefiniti di Gurobi accettano solo alcune **forme di vincoli**

quadratici, noti per avere regioni convesse realizzabili, come per esempio quelle elencate qui di seguito:

- $x^T Q x + q^T x \leq b$;
- $x^T x \leq y^2$, dove x è un vettore di variabili e y una variabile non negativa;
- $x^T x \leq yz$, dove x è un vettore di variabili, e y e z sono variabili non negative.

Quindi verrà accettato un vincolo quadratico solo se si è in grado di trasformarlo in una di queste forme.

Si noti che altri risolutori quadratici non convessi spesso trovano solo soluzioni localmente ottime, invece gli algoritmi su cui si basa Gurobi esplorano l'intero spazio di ricerca, quindi forniscono un limite inferiore valido a livello globale e quindi, dato un tempo sufficiente, potranno trovare una soluzione ottima globale.

Vincoli generali

I vincoli sopra descritti sono generalmente gestiti direttamente dagli algoritmi di ottimizzazione sottostanti. Gurobi include una serie aggiuntiva di vincoli chiamati vincoli generali progettati per una questione di praticità e funzionalità, in quanto permettono di definire facilmente determinate relazioni variabili.

Vi sono due tipologie di vincoli generali:

- *Vincoli di funzione*, come per esempio $y = f(x)$, dove x e y sono due variabili decisionali e $f()$ è la funzione scelta in una lista di funzioni tra le seguenti: polinomiale, esponenziale, logaritmica, potenza, seno, coseno e tangente;

- *Vincoli semplici generali*, che consentono di stabilire relazioni comuni ma più dirette tra le variabili decisionali, e quelli supportati da Gurobi sono:
  - vincolo `MAX`
  - vincolo `MIN`
  - vincolo `ABS`
  - vincolo `AND`
  - vincolo `OR`
  - vincolo `INDICATOR`
  - vincolo lineare a tratti

Metodo per aggiungere vincoli al modello

Per aggiungere un nuovo vincolo al modello preso in esame viene usato il metodo `addConstr()`, dove i principali argomenti che accetta sono:

- `lhs`: membro sinistro del vincolo;
- `rhs`: membro destro del vincolo;
- `sense`: senso del vincolo, che può essere:
  - `GRB.EQUAL`;
  - `GRB.LESS_EQUAL`;
  - `GRB.GREATER_EQUAL`.
- `name`: nome de nuovo vincolo.

Un esempio dell'utilizzo di questo metodo è il seguente:

```
model.addConstr(x + 2y, GRB.EQUAL, 3Z, name='C_0')
```

A.2.4 Funzione obbiettivo

Ogni modello di ottimizzazione ha una funzione obbiettivo, che è la funzione definita mediante le variabili decisionali che si desidera minimizzare o massimizzare. La maggior parte dei problemi ha molteplici soluzioni ottimali, quella restituita da Gurobi dipende dal tipo di problema preso in esame. In generale questo ottimizzatore restituisce un'unica soluzione ottimale per modelli continui e una sequenza di soluzioni migliorative per modelli discreti.

Gli algoritmi di Gurobi lavorano per risolvere un modello fino a quando non trovano una soluzione ottimale entro le tolleranze specificate. Per modelli discreti, nonostante sia possibile chiedere al risolutore di trovare una soluzione con il miglior valore obbiettivo, è molto comune fermarsi quando l'obbiettivo della soluzione rientra in un intervallo specificato del valore ottimale. Questo Gap dell'ottimalità è controllato dal parametro `MIPGap` il cui valore predefinito è 0,01

Le funzioni obbiettivo possono essere: lineari, quadratiche, lineare a tratti e multi-obbiettivo lineari.

Metodi principali della funzione obbiettivo

Per importare l'obbiettivo del modello uguale ad un'espressione lineare o quadratica viene usato il metodo `setObjective()`, che ha come parametri:

- `linExpr` o `quadExpr`: espressione lineare o quadratica:
- `sense`: si può specificare se si desidera minimizzarla o massimizzarla.

Per esempio sia $f(x)$ la funzione obbiettivo, supponiamo di volerla minimizzare:
`model.setObjective(`$f(x)$`, GRB.MINIMIZE)` Una volta settata la funzione obbittivo per minimizzarla è necessario richiamare il metodo `optimize()` sul

modello in questione:

```
model.optimize()
```

Bibliografia