# Dal modello a Python/Gurobi

# Indice

Le istruzioni di base di gurobipy

Le quattro classi di variabili

Il modello, una famiglia per blocco

Leggere i risultati

Le tolleranze

Il rilassamento con `relax()`

Tre problemi che il corso riusa

Il protocollo del corso, dall'inizio alla fine

Eseguire il codice

# Obiettivi (1/2)

Alla fine di questo capitolo sarai in grado di:

- scrivere un modello in `gurobipy`, una famiglia di variabili e di vincoli per blocco;
- dichiarare le quattro classi di variabili, ricordando i bound predefiniti;
- leggere i risultati nell'ordine giusto: `Status`, `SolCount`, `ObjVal` e `ObjBound`;

# Obiettivi (2/2)

Alla fine di questo capitolo sarai in grado di:

- usare correttamente le tolleranze nel leggere i valori interi;
- risolvere il rilassamento LP con `relax()` e leggerne i duali.

# Prerequisiti

- Da questo modulo, Capitolo 1 «Che cos'è un modello MIP»: le classi di variabili e la struttura di un modello
- Da questo modulo, Capitolo 2 «Rilassamenti, dualità e bound»: il rilassamento LP, il suo duale e gli attributi `ObjVal` e `ObjBound` del solver

# Un solo solver

- Il corso usa **un solo solver**, Gurobi da Python
- Questo capitolo mostra come si scrive un modello: una famiglia di vincoli per blocco, con i nomi del modello matematico
- E soprattutto come si *leggono* i risultati, incluso il caso in cui il solver non ha finito

# Le istruzioni di base di gurobipy

# Introduzione: le istruzioni di base di gurobipy

- Un modello si scrive in gurobipy con poche istruzioni, sempre le stesse
- Da qui in poi accanto a ogni modello comparirà la riga di codice che lo scrive
- Conviene vederle una volta su un modello piccolo e completo: lo **zaino binario**

# Lo zaino binario

- Di $n$ oggetti di valore $p_j$ e peso $w_j$ si sceglie un sottoinsieme di peso al più $c$ che valga il più possibile

$$\max \quad \sum_{j=1}^{n} p_j \, x_j$$

$$\text{soggetto a} \quad \sum_{j=1}^{n} w_j \, x_j \leq c,$$

$$x_j \in \{0,1\}, \quad \forall j \in \{1,2,\ldots,n\}.$$

# Lo zaino in gurobipy (1/2)

I moduli e i dati:

```
import gurobipy as gp       # il modulo, sempre abbreviato gp
from gurobipy import GRB    # le costanti: GRB.BINARY, ...

p = [10, 7, 6, 4]           # valori
w = [5, 4, 3, 3]            # pesi
c = 9                       # capacita'
n = len(p)
```

# Lo zaino in gurobipy (2/2)

Il modello, una istruzione per riga del modello scritto:

```
m = gp.Model("zaino")                                  # un modello vuoto
x = m.addVars(n, vtype=GRB.BINARY, name="x") # n binarie
m.setObjective(gp.quicksum(p[j] * x[j] for j in range(n)),
               GRB.MAXIMIZE)
m.addConstr(gp.quicksum(w[j] * x[j] for j in range(n)) <= c,
            name="capacita")
m.optimize()
```

# Le istruzioni (1/3)

- `Model` crea il modello; tutto il resto si appende a lui
- `addVars` aggiunge un'intera famiglia indicizzata:
  - `addVars(n)` dà $x_0, \ldots, x_{n-1}$, si indicizza con `x[j]`
  - `addVars(n, k)` dà la matrice $x_{ij}$, si indicizza con `x[i, j]`
  - per una variabile sola c'è `addVar`
- Gli indici di Python partono da zero, nei modelli scritti partono da uno: è la sola differenza da tenere a mente

# Le istruzioni (2/3)

- `vtype` dichiara il dominio:
  - `GRB.BINARY` per $x \in \{0, 1\}$
  - `GRB.INTEGER` per $x \in \mathbb{Z}_{\geq 0}$
  - senza `vtype` la variabile è continua e non negativa
- Il dominio è un vincolo: dichiararlo male è dichiarare un altro modello
- `quicksum` costruisce una somma $\sum_j c_j x_j$: fa lo stesso di `sum` ma senza costruire espressioni intermedie, ed è quella che si usa sempre

# Le istruzioni (3/3)

- `setObjective` prende l'espressione e il verso, `GRB.MINIMIZE` o `GRB.MAXIMIZE`
- `addConstr` aggiunge *un* vincolo, `addConstrs` una famiglia, e vuole un generatore fra parentesi tonde
  - una riga di codice per una famiglia di vincoli, com'è una riga sola nel modello scritto
- `optimize` risolve

# Il `name`

- Il `name` non serve al solver
- Serve a chi legge il modello esportato o un messaggio di errore
- Vale la pena darlo sempre uguale al nome della famiglia di vincoli nel modello scritto

# Il bin packing con le stesse istruzioni

- $n$ oggetti di dimensione $w_j$ in contenitori di capacità $c$, usandone il meno possibile
- Binarie $x_{ji}$ («l'oggetto $j$ va nel contenitore $i$») e $y_i$ («il contenitore $i$ si usa»)

```
x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
y = m.addVars(k, vtype=GRB.BINARY, name="y")
m.setObjective(y.sum(), GRB.MINIMIZE)
m.addConstrs((x.sum(j, "*") == 1 for j in range(n)),
             name="assegna")
m.addConstrs((gp.quicksum(w[j] * x[j, i] for j in range(n))
              <= c * y[i] for i in range(k)), name="capacita")
```

# Scheduling e commesso viaggiatore

- Lo **scheduling su macchine identiche** ($P||z$) cambia solo l'obiettivo e un vincolo:
  - una continua $T$ per il makespan, min $T$
  - $\sum_j t_j x_{ji} \leq T$ per ogni macchina
- Il **commesso viaggiatore** ha una binaria per arco, $x_{ij}$, e due famiglie di uguaglianze:
  - da ogni città si parte una volta e in ogni città si arriva una volta
  - più i vincoli che impediscono i sottogiri, che sono la parte interessante

# Riepilogo

- Un modello gurobipy usa sempre le stesse istruzioni: `Model`, `addVars`, `setObjective`, `addConstr(s)`, `optimize`
- `vtype` dichiara il dominio, che è parte del modello
- `quicksum` per le somme, una `addConstrs` per famiglia
- Indici da zero in Python, da uno nei modelli scritti

# Le quattro classi di variabili

# Introduzione: le quattro classi di variabili

- Ogni variabile di un modello ha un dominio, e in gurobipy il dominio si dichiara
- Le classi che servono sono quattro: binarie, intere, continue, libere
- Due valori di default si dimenticano spesso, e producono modelli silenziosamente sbagliati

# Le quattro classi di variabili

| Che cosa | `vtype` | Dominio | Uso tipico |
|---|---|---|---|
| decisione sì/no | `GRB.BINARY` | $\{0, 1\}$ | selezione, attivazione, assegnamento |
| conteggio | `GRB.INTEGER` | $\mathbb{Z}$ fra `lb` e `ub` | scatole, turni, operai |
| quantità misurabile | (default) | $[\texttt{lb}, \texttt{ub}] \subseteq \mathbb{R}$ | tempo, denaro, flusso |
| variabile libera | (default) con `lb=-GRB.INFINITY` | $\mathbb{R}$ | duali di uguaglianze |

## Osservazione 1: I due default che si dimenticano

*`GRB.BINARY` implica già `lb` $= 0$ e `ub` $= 1$: non serve ripeterli.*

*Una variabile continua ha `lb = 0` per default: una variabile che deve poter essere negativa — tipicamente un duale di uguaglianza, o una deviazione con segno — va dichiarata con `lb=-GRB.INFINITY`, altrimenti il modello è silenziosamente sbagliato.*

# Riepilogo

- Binarie, intere, continue e libere: quattro classi, dichiarate con `vtype` e i bound
- Le binarie hanno già i bound 0 e 1
- Le continue hanno `lb = 0` per default: una variabile libera va dichiarata con `lb=-GRB.INFINITY`

# Il modello, una famiglia per blocco

# Introduzione: il modello, una famiglia per blocco

- Vediamo come il corso scrive un modello completo in una funzione Python
- Tre regole di scrittura rendono il codice uguale, riga per riga, al modello matematico
- Infine vediamo come controllare che il modello scritto sia quello pensato sulla carta

# Un assegnamento in una funzione

```
def modello(t, c, a):
    m = gp.Model("assegnamento");  m.Params.OutputFlag = 0
    x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
    m.setObjective(gp.quicksum(c[j][h] * x[j, h]
                   for j in range(n) for h in range(k)),
                   GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in range(n)),
                 name="assegna")
    m.addConstrs((gp.quicksum(t[j][h] * x[j, h]
                  for j in range(n)) <= a[h] for h in range(k)),
                 name="disponibilita")
    return m, x
```

# Le tre regole di scrittura del corso (1/2)

1. **Una** `addConstrs` **per famiglia**, nell'ordine del modello matematico
   - il `name` è l'etichetta della famiglia (`"assegna"`, `"disponibilita"`)
   - i nomi generati sono `assegna[0]`, `assegna[1]`, ...: si ritrovano nei duali e nei messaggi del solver

# Le tre regole di scrittura del corso (2/2)

2. **Le variabili si dichiarano tutte insieme**, con `addVars` e gli indici del modello
   - `x.sum(j, "*")` è la somma su tutti gli indici della seconda dimensione: è la scrittura dell'*indice muto* del modello
3. **I dati arrivano come argomenti**, non come variabili globali
   - la stessa funzione serve per l'istanza base e per tutte le varianti delle domande di modellazione

# Controllare il modello

- Il modello scritto ha 9 variabili, 6 vincoli e 18 coefficienti non nulli
- Per controllare che sia quello che si è scritto sulla carta, `m.write("modello.lp")` produce il formato LP leggibile
- È il modo più rapido per accorgersi di un coefficiente sbagliato: il tabulare dell'istanza nella dispensa e questo output devono coincidere riga per riga

# Il file `modello.lp` (1/2)

```
Minimize
  5 x[0,0] + 10 x[0,1] + 2 x[0,2] + 5 x[1,0]
   + 4 x[1,1] + 6 x[1,2] + 5 x[2,0] + 4 x[2,1]
   + 6 x[2,2]
Subject To
 assegna[0]: x[0,0] + x[0,1] + x[0,2] = 1
 assegna[1]: x[1,0] + x[1,1] + x[1,2] = 1
 assegna[2]: x[2,0] + x[2,1] + x[2,2] = 1
```

# Il file `modello.lp` (2/2)

```
 disponibilita[0]: 2 x[0,0] + 3 x[1,0] + 4 x[2,0] <= 5
 disponibilita[1]: x[0,1] + 4 x[1,1] + 5 x[2,1] <= 6
 disponibilita[2]: 3 x[0,2] + 2 x[1,2] + 3 x[2,2] <= 7
Bounds
Binaries
 x[0,0] x[0,1] x[0,2] x[1,0] x[1,1] x[1,2]
 x[2,0] x[2,1] x[2,2]
End
```

# Riepilogo

- Una `addConstrs` per famiglia, con il `name` uguale all'etichetta del modello
- Variabili dichiarate tutte insieme con `addVars`; `x.sum(j, "*")` è l'indice muto
- I dati come argomenti della funzione
- `m.write("modello.lp")` per confrontare riga per riga con il modello scritto

# Leggere i risultati

# Introduzione: leggere i risultati

- Dopo `optimize()` il solver non dà sempre l'ottimo: può fermarsi prima, o scoprire che non ci sono soluzioni
- I risultati si leggono in un ordine preciso, che non si cambia
- Vediamo che cosa si può dire — e che cosa *non* si deve dire — in ogni caso

# L'ordine di lettura, che non si cambia

1. `Status`. Sempre per primo.
2. `SolCount`: se vale 0, *non esiste* una soluzione e `ObjVal` non si può leggere.
3. `ObjVal` e `ObjBound`: la migliore soluzione trovata e il miglior bound dimostrato.
4. `MIPGap`, `NodeCount`, `Runtime`: quanto resta incerto, e quanto è costato.

# Che cosa si può dire, caso per caso

| Status | valore | SolCount | Che cosa si può dire |
|---|---|---|---|
| OPTIMAL | 2 | $\geq 1$ | $z(MILP) =$ ObjVal, dimostrato |
| INFEASIBLE | 3 | 0 | il modello non ha soluzioni ammissibili |
| UNBOUNDED | 5 | 0 | manca un vincolo, o un bound su una variabile |
| TIME_LIMIT | 9 | 0 | niente: né una soluzione né, in generale, un bound utile |
| TIME_LIMIT | 9 | $\geq 1$ | ObjBound $\leq z(MILP) \leq$ ObjVal |
| SOLUTION_LIMIT | 10 | $\geq 1$ | come sopra |

# Esempio: i quattro casi sull'istanza di assegnamento (1/2)

- **Risoluzione normale.**
  - $\texttt{Status} = 2$, $\texttt{SolCount} = 2$, $\texttt{ObjVal} = \texttt{ObjBound} = 11$
  - $\texttt{MIPGap} = 0$, $\texttt{NodeCount} = 0$: Gurobi ha chiuso tutto nella radice
- **Inammissibile.**
  - con disponibilità $a = (1, 1, 1)$ nessun assegnamento è possibile
  - $\texttt{Status} = 3$, $\texttt{SolCount} = 0$

# Esempio: i quattro casi sull'istanza di assegnamento (2/2)

- **Fermato subito.** Con `TimeLimit = 0`:
  - `Status` $= 9$, `SolCount` $= 0$ e `ObjBound` $= -\infty$
  - non c'è nulla da riportare: né una soluzione né un bound
- **Fermato alla prima soluzione.** Con `SolutionLimit = 1`:
  - `Status` $= 10$, `SolCount` $= 1$, `ObjVal` $= 12$, `ObjBound` $= 10$, `MIPGap` $= 0{,}1667$
  - si riporta un *intervallo*: l'ottimo sta fra 10 e 12, e il 16,7% ne misura l'ampiezza
  - dire «l'ottimo è 12» sarebbe falso

## Osservazione 2: Che cosa *non* si riporta

- *Non si scrive «l'ottimo è `ObjVal`» se `Status` non è `OPTIMAL`.*
- *Non si scrive un gap se `SolCount` è* 0*.*
- *Non si confronta il `Runtime` di due modelli risolti con impostazioni diverse.*
- *Non si legge `ObjBound` a fine risoluzione pensando di leggere il rilassamento della radice: per quello serve `relax()`.*

# Riepilogo

- Ordine di lettura: `Status`, `SolCount`, `ObjVal` e `ObjBound`, poi `MIPGap`, `NodeCount`, `Runtime`
- Solo con `OPTIMAL` si parla di ottimo
- Fermato con almeno una soluzione: si riporta l'intervallo $\texttt{ObjBound} \leq z(MILP) \leq \texttt{ObjVal}$
- Con $\texttt{SolCount} = 0$ non c'è né soluzione né gap

# Le tolleranze

# Introduzione: le tolleranze

- Il solver lavora in aritmetica floating point: niente è esatto
- Quattro parametri dicono quanto il solver può sbagliare su interezza, vincoli, costi ridotti e gap
- Conoscerli evita errori nel leggere e nel confrontare i valori delle variabili

# I parametri di tolleranza

| Parametro | Default | Significato |
|---|---|---|
| `IntFeasTol` | $10^{-5}$ | quanto una variabile intera può distare da un intero |
| `FeasibilityTol` | $10^{-6}$ | violazione ammessa su un vincolo lineare |
| `OptimalityTol` | $10^{-6}$ | tolleranza sui costi ridotti |
| `MIPGap` | $10^{-4}$ | gap relativo sotto il quale il solver si ferma |

## Osservazione 3: «Intero» significa «intero entro una tolleranza» (1/2)

*Una binaria può tornare come* 0,9999999997. *Nel testo si scrive* 1*: i valori si arrotondano quando si riportano, e i confronti si fanno sempre con una tolleranza — nel corso,* $10^{-6}$*, la costante* `TOL` *di* `python/mip.py`.

*Scrivere* `if x.X == 1` *è un errore; si scrive* `if x.X > 0.5`.

## Osservazione 4: «Intero» significa «intero entro una tolleranza» (2/2)

*Il default* `MIPGap` $= 10^{-4}$ *significa anche che il solver può fermarsi prima dell'ottimo esatto, dichiarando* `OPTIMAL`*: su istanze con valori grandi vale la pena abbassarlo.*

# Riepilogo

- `IntFeasTol`, `FeasibilityTol`, `OptimalityTol`, `MIPGap`: le quattro tolleranze
- I valori si arrotondano quando si riportano; i confronti si fanno con una tolleranza (`x.X > 0.5`, mai `x.X == 1`)
- `OPTIMAL` vuol dire ottimo entro `MIPGap`: su valori grandi conviene abbassarlo

# Il rilassamento con `relax()`

# Introduzione: il rilassamento con `relax()`

- Il rilassamento lineare dà il bound $z(LP)$ e le variabili duali
- In gurobipy si ottiene con `relax()`, che copia il modello togliendo l'interezza
- Vediamo il codice e i numeri sull'istanza di assegnamento

# Il rilassamento in gurobipy

```
m.update()      # relax() copia il modello: le modifiche
                # pendenti vanno applicate prima
r = m.relax()   # binarie -> 0 <= x <= 1, intere -> x >= lb
r.Params.OutputFlag = 0
r.optimize()
zlp = r.ObjVal
duali = {c.ConstrName: c.Pi for c in r.getConstrs()}
```

# Il rilassamento sull'istanza di assegnamento

- Su quell'istanza $z(LP^+) = z(LP) = 53/5$
- I due rilassamenti coincidono, perché i vincoli di assegnamento implicano già $x_{jm} \leq 1$
- I duali non nulli sono

$$\tilde{\mu} = (2,\ 4{,}8,\ 5), \qquad \tilde{\pi}_2 = -0{,}2$$

- Cioè le tre variabili duali dei vincoli `assegna` e quella del vincolo `disponibilita[1]`: la macchina 2 è la sola risorsa stretta

# Il rilassamento senza i bound, quando serve

- La funzione `rilassamento(m, rafforzato=False)` di `python/mip.py` toglie anche il limite superiore delle binarie
- È il rilassamento di cui la dispensa scrive il duale a mano
- Le due versioni si riportano entrambe, con l'annotazione se coincidono e perché

# Riepilogo

- `m.update()` e poi `m.relax()`: una copia del modello senza interezza
- `r.ObjVal` è $z(LP)$, `c.Pi` il duale di ogni vincolo
- Con o senza i bound $x \leq 1$: si riportano entrambi i rilassamenti, dicendo se coincidono e perché

# Tre problemi che il corso riusa

# Introduzione: tre problemi che il corso riusa

- Fin qui il modello di esempio è sempre stato lo zaino
- I tre problemi di questa sezione — bin packing, makespan, commesso viaggiatore — ritornano nel capitolo delle euristiche
- Lì si costruiscono a mano le soluzioni di next-fit, first-fit, best-fit, LPT e vicino più vicino
- Qui si scrivono i loro modelli, così quelle euristiche hanno un ottimo con cui confrontarsi

# Bin packing: quanti contenitori bastano

# Bin packing

- Ci sono $n \in \mathbb{Z}_{\geq 1}$ oggetti: l'oggetto $j \in \{1, 2, \ldots, n\}$ pesa $w_j \in \mathbb{Q}_{>0}$
- I contenitori sono tutti uguali, di capacità $c \in \mathbb{Q}_{>0}$, e se ne può usare al più uno per oggetto
- Si usi il minimo numero di contenitori
- Servono due famiglie di variabili binarie:
  - $x_{jb} = 1$ se l'oggetto $j$ va nel contenitore $b$
  - $y_b = 1$ se il contenitore $b$ viene usato

## Bin packing

$$\min \quad \sum_{b=1}^{k} y_b \tag{3.1a}$$

$$\text{soggetto a} \quad \sum_{b=1}^{k} x_{jb} = 1, \qquad \forall j \in \{1, 2, \ldots, n\}, \tag{3.1b}$$

$$\sum_{j=1}^{n} w_j \, x_{jb} - c \, y_b \leq 0, \qquad \forall b \in \{1, 2, \ldots, k\}, \tag{3.1c}$$

$$x_{jb} \in \{0, 1\}, \quad \forall j \in \{1, 2, \ldots, n\}, \ \forall b \in \{1, 2, \ldots, k\}, \tag{3.1d}$$

$$y_b \in \{0, 1\}, \qquad \forall b \in \{1, 2, \ldots, k\}. \tag{3.1e}$$

# Bin packing: descrizione del modello

- L'obiettivo conta i contenitori accesi
- I vincoli $\sum_b x_{jb} = 1$ dicono che ogni oggetto finisce in esattamente un contenitore
- I vincoli di capacità sono scritti come *attivazione*: finché $y_b = 0$ il contenitore $b$ non può ricevere niente, e appena $y_b = 1$ accoglie fino a $c$
- I domini sono le due famiglie di binarie, una per coppia oggetto-contenitore e una per contenitore

# Bin packing: la costruzione in gurobipy

```
def modello_bpp(w, c, k):
    n = len(w)
    m = nuovo_modello("bin_packing")
    x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
    y = m.addVars(k, vtype=GRB.BINARY, name="y")
    m.setObjective(y.sum(), GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in R(n)),
                 name="oggetto")
    m.addConstrs((gp.quicksum(w[j] * x[j, b] for j in R(n))
                  <= c * y[b] for b in R(k)), name="capacita")
    return m, x, y
```

# Bin packing sull'istanza

## Bin packing sull'istanza

$$
\begin{aligned}
\min \quad & y_1 + y_2 + y_3 \\
\text{soggetto a} \quad & x_{11}+x_{12}+x_{13} = 1 \\
& x_{21} + x_{22} + x_{23} = 1 \\
& x_{31} + x_{32} + x_{33} = 1 \\
& x_{41} + x_{42} + x_{43} = 1 \\
& 5x_{11} + 4x_{21} + 3x_{31} + 3x_{41} - 7y_1 \le 0 \\
& 5x_{12} + 4x_{22} + 3x_{32} + 3x_{42} - 7y_2 \le 0 \\
& 5x_{13} + 4x_{23} + 3x_{33} + 3x_{43} - 7y_3 \le 0 \\
& x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}, x_{41}, x_{42}, x_{43} \in \{0,1\} \\
& y_1, y_2, y_3 \in \{0,1\}
\end{aligned}
$$

# Bin packing: il conteggio sull'istanza

- Il peso totale è 15, quindi nessuna soluzione può usare meno di $\lceil 15/7 \rceil = 3$ contenitori
- L'ottimo ne usa esattamente 3: il conteggio è quindi stretto
- Nel modello i contenitori possibili sono tre, il numero che serve davvero
- Con sei contenitori il modello avrebbe 42 colonne e le ultime tre resterebbero vuote in ogni soluzione ottima

## Osservazione 5: Il rilassamento del bin packing è debolissimo

*Rilassando $y_b$ a $y_b \geq 0$ il modello compra frazioni di contenitore, e l'ottimo dell'LP scende a $\sum_j w_j/c = 15/7 \approx 2{,}14$: il rilassamento non sa che un contenitore si apre tutto intero.*

*È il motivo per cui su questo problema il bound duale è poco utile e le euristiche contano di più.*

# Makespan: il carico della macchina più carica

# Makespan su macchine identiche

- Ci sono $n \in \mathbb{Z}_{\geq 1}$ lavori, di durata $d_j \in \mathbb{Q}_{>0}$, e $k \in \mathbb{Z}_{\geq 1}$ macchine identiche
- Ogni lavoro va su una macchina sola e non si interrompe
- Si minimizzi il carico della macchina più carica
- Variabili:
  - $x_{jm} = 1$ se il lavoro $j$ va sulla macchina $m$
  - $z \geq 0$ il carico della macchina più carica

## Makespan su macchine identiche

$$\min \quad z \tag{3.2a}$$

$$\text{soggetto a} \quad \sum_{m=1}^{k} x_{jm} = 1, \qquad \forall j \in \{1, 2, \ldots, n\}, \tag{3.2b}$$

$$\sum_{j=1}^{n} d_j \, x_{jm} - z \leq 0, \qquad \forall m \in \{1, 2, \ldots, k\}, \tag{3.2c}$$

$$x_{jm} \in \{0, 1\}, \quad \forall j \in \{1, 2, \ldots, n\}, \ \forall m \in \{1, 2, \ldots, k\}, \tag{3.2d}$$

$$z \geq 0. \tag{3.2e}$$

# Makespan: descrizione del modello

- L'obiettivo è la sola variabile $z$: nessun dato vi compare
- Sono le $k$ righe di carico a darle significato, dicendo che nessuna macchina lavora più a lungo di $z$
- Il minimo la schiaccia allora sul carico della macchina più carica: è la tecnica *min-max*
- I vincoli $\sum_m x_{jm} = 1$ assegnano ogni lavoro a una macchina sola; i domini dichiarano binarie le assegnazioni e non negativo il carico

# Makespan: la costruzione in gurobipy

```
def modello_cmax(d, k):
    n = len(d)
    m = nuovo_modello("makespan")
    x = m.addVars(n, k, vtype=GRB.BINARY, name="x")
    z = m.addVar(name="z")
    m.setObjective(z, GRB.MINIMIZE)
    m.addConstrs((x.sum(j, "*") == 1 for j in R(n)),
                 name="lavoro")
    m.addConstrs((gp.quicksum(d[j] * x[j, mm] for j in R(n))
                  <= z for mm in R(k)), name="carico")
    return m, x, z
```

# Makespan sull'istanza

## Makespan sull'istanza

$$
\begin{array}{lrrrrrrrrl}
\min & & & & & & & & z & \\
\text{soggetto a} & x_{11} & +x_{12} & & & & & & & = 1 \\
 & & & x_{21} & +x_{22} & & & & & = 1 \\
 & & & & & x_{31} & +x_{32} & & & = 1 \\
 & & & & & & & x_{41} & +x_{42} & = 1 \\
 & 3x_{11} & & +4x_{21} & & +5x_{31} & & +6x_{41} & & -z \leq 0 \\
 & & 3x_{12} & & +4x_{22} & & +5x_{32} & & +6x_{42} & -z \leq 0 \\
 & x_{11}, & x_{12}, & x_{21}, & x_{22}, & x_{31}, & x_{32}, & x_{41}, & x_{42} & \in \{0,1\} \\
 & & & & & & & & z & \geq 0
\end{array}
$$

# Makespan: il conteggio sull'istanza

- Il carico totale è 18 e le macchine sono due: nessuna soluzione può scendere sotto $18/2 = 9$
- L'ottimo vale esattamente 9: i lavori si dividono in $6 + 3$ e $5 + 4$
- Qui il conteggio chiude il problema da solo

# Commesso viaggiatore: la formulazione MTZ

# Commesso viaggiatore

- Ci sono $n \in \mathbb{Z}_{\geq 2}$ città e una distanza $d_{ij} \in \mathbb{Q}_{>0}$ fra ogni coppia
- Si trovi il giro di lunghezza minima che tocca ogni città esattamente una volta e torna al punto di partenza

# Commesso viaggiatore: le variabili

- $x_{ij} = 1$ se il giro va da $i$ a $j$
- Le due famiglie «si esce una volta» e «si entra una volta» non bastano: ammettono anche soluzioni fatte di *sottocicli* separati
- La formulazione di Miller–Tucker–Zemlin aggiunge una variabile $u_i$ per ogni città diversa dalla prima, che ne registra la posizione lungo il giro

## TSP, formulazione MTZ

$$\min \quad \sum_{i=1}^{n} \sum_{j \neq i} d_{ij}\, x_{ij} \tag{3.3a}$$

$$\text{soggetto a} \quad \sum_{j \neq i} x_{ij} = 1, \qquad \forall i \in \{1, 2, \ldots, n\}, \tag{3.3b}$$

$$\sum_{i \neq j} x_{ij} = 1, \qquad \forall j \in \{1, 2, \ldots, n\}, \tag{3.3c}$$

$$u_i - u_j + n\, x_{ij} \leq n - 1, \qquad \forall i, j \in \{2, 3, \ldots, n\},\ i \neq j, \tag{3.3d}$$

$$x_{ij} \in \{0, 1\}, \qquad \forall i, j \in \{1, 2, \ldots, n\},\ i \neq j, \tag{3.3e}$$

$$u_i \in [1,\ n - 1], \qquad \forall i \in \{2, 3, \ldots, n\}. \tag{3.3f}$$

# TSP: descrizione del modello (1/2)

- L'obiettivo somma le distanze degli archi percorsi
- Le due famiglie di uguaglianze dicono che da ogni città si esce una volta e in ogni città si entra una volta
- I vincoli MTZ sono il cuore della formulazione:
  - se $x_{ij} = 0$ la riga diventa $u_i - u_j \leq n - 1$, sempre vera perché le $u$ stanno fra 1 e $n - 1$: non vieta niente
  - se $x_{ij} = 1$ diventa $u_j \geq u_i + 1$, cioè «se vado da $i$ a $j$, la posizione di $j$ è la successiva»

# TSP: descrizione del modello (2/2)

- Un sottociclo che non tocca la città 1 richiederebbe una catena di posizioni sempre crescenti che si richiude su se stessa
- Questo è impossibile
- La città 1 non ha la sua $u$ proprio perché è il punto in cui il giro si chiude

# TSP: la costruzione in gurobipy (1/2)

Le variabili e l'obiettivo:

```
def modello_tsp(D):
    n = len(D)
    m = nuovo_modello("tsp")
    x = m.addVars(((i, j) for i in R(n) for j in R(n)
                   if i != j), vtype=GRB.BINARY, name="x")
    u = m.addVars(R(1, n), lb=1, ub=n - 1, name="u")
    m.setObjective(gp.quicksum(D[i][j] * x[i, j]
                   for i, j in x), GRB.MINIMIZE)
```

# TSP: la costruzione in gurobipy (2/2)

I vincoli, una famiglia per blocco:

```
m.addConstrs((gp.quicksum(x[i, j] for j in R(n) if j != i)
              == 1 for i in R(n)), name="esce")
m.addConstrs((gp.quicksum(x[i, j] for i in R(n) if i != j)
              == 1 for j in R(n)), name="entra")
m.addConstrs((u[i] - u[j] + n * x[i, j] <= n - 1
              for i in R(1, n) for j in R(1, n) if i != j),
             name="mtz")
return m, x, u
```

# TSP sull'istanza

- Sull'istanza di quattro città del capitolo delle euristiche il giro ottimo è $1 \to 2 \to 4 \to 3 \to 1$ e misura 22
- Il modello dell'istanza ha quindici colonne — dodici archi e tre posizioni — e quattordici righe

# TSP sull'istanza

## TSP sull'istanza

$$
\begin{aligned}
\min\ & 4x_{12}+5x_{13}+9x_{14}+4x_{21}+9x_{23}+9x_{24}+5x_{31}+9x_{32}+4x_{34}+9x_{41}+9x_{42}+4x_{43} \\
\text{soggetto a}\ & x_{12}+x_{13}+x_{14} = 1 \\
& x_{21}+x_{23}+x_{24} = 1 \\
& x_{31}+x_{32}+x_{34} = 1 \\
& x_{41}+x_{42}+x_{43} = 1 \\
& x_{21}+x_{31}+x_{41} = 1 \\
& x_{12}+x_{32}+x_{42} = 1 \\
& x_{13}+x_{23}+x_{43} = 1 \\
& x_{14}+x_{24}+x_{34} = 1 \\
& 4x_{23}+u_2-u_3 \le 3 \\
& 4x_{24}+u_2-u_4 \le 3 \\
& 4x_{32}-u_2+u_3 \le 3 \\
& 4x_{34}+u_3-u_4 \le 3 \\
& 4x_{42}-u_2+u_4 \le 3 \\
& 4x_{43}-u_3+u_4 \le 3 \\
& x_{12},\ x_{13},\ x_{14},\ x_{21},\ x_{23},\ x_{24},\ x_{31},\ x_{32},\ x_{34},\ x_{41},\ x_{42},\ x_{43} \in \{0,1\} \\
& u_2,\ u_3,\ u_4 \ge 1
\end{aligned}
$$

## Osservazione 6: MTZ è comoda, non è la più forte

*I vincoli MTZ sono $O(n^2)$ e si scrivono in tre righe di `gurobipy`, ma il loro rilassamento lineare è debole: le u continue assorbono quasi tutto e l'LP si avvicina poco all'ottimo intero.*

*Le formulazioni che eliminano i sottocicli con i tagli di connessione danno bound molto migliori, al prezzo di un numero esponenziale di vincoli da generare a mano a mano. Per le dimensioni di questo corso MTZ basta.*

# Riepilogo (1/2)

**Bin packing**:

- Binarie $x_{jb}$ e $y_b$; capacità scritta come attivazione $\sum_j w_j x_{jb} \leq c\, y_b$
- Sull'istanza l'ottimo è $3 = \lceil 15/7 \rceil$; il rilassamento $(15/7)$ è debolissimo

**Makespan**:

- Tecnica min-max: $\min z$ con $\sum_j d_j x_{jm} \leq z$ per ogni macchina
- Sull'istanza l'ottimo è $9 = 18/2$: il conteggio chiude il problema

# Riepilogo (2/2)

**Commesso viaggiatore (MTZ)**:

- Uguaglianze «si esce una volta» e «si entra una volta» più le posizioni $u_i$
- $u_i - u_j + n\,x_{ij} \leq n - 1$ vieta i sottocicli che non toccano la città 1
- Comoda da scrivere, ma con rilassamento debole
- Sull'istanza il giro ottimo misura 22

# Il protocollo del corso, dall'inizio alla fine

# Introduzione: il protocollo del corso

- Ogni esercizio del corso segue la stessa sequenza, dai dati alla tabella dei bound
- Ogni passo produce un numero che si controlla in automatico
- Il risultato finisce in un solo file, da cui lo leggono dispensa, sito e verifiche

## La sequenza di ogni esercizio

dati $\rightarrow$ modello $\rightarrow$ euristica e verifica $\rightarrow$ LP e duale

$\rightarrow$ MIP $\rightarrow$ tabella $\rightarrow$ figure e notebook

# Il protocollo in codice (1/2)

Dati, modello, euristica e verifica:

```
m, x = modello(t, c, a)                          # (1) dati, (2) modello

e = best_fit(t, a, lambda j, h, ra: c[j][h], "costo")
ub = sum(c[j][h] for (j, h) in e.x)     # (3) euristica
# vincoli, bound E interezza:
assert ammissibile(m, {f"x[{j},{h}]": 1 for (j, h) in e.x})
```

# Il protocollo in codice (2/2)

Duale, rilassamenti, MIP e tabella:

```
d = duale(t, c, a)                    # (4) duale scritto a mano
mano = {f"mu[{j}]": min(c[j]) for j in range(n)}
lb, viol = valuta(d, mano);  assert viol <= 1e-9
zlp, zlp_raff, pi = due_rilassamenti(m, d)  # dualita' forte

z = risolvi(m)                                  # (5) il MIP
riga = registra_bound("assegnamento", ub, lb, zlp,
                      zlp_raff, z)              # (6) tabella
salva_dati(pd.DataFrame([riga]), "fam07_1_bound")
```

# I quattro numeri del protocollo

- Sull'istanza di assegnamento di questo capitolo il protocollo produce

$$LB = 10, \qquad z(LP) = z(LP^{+}) = \tfrac{53}{5}, \qquad z(MILP) = 11, \qquad UB = 11$$

- La riga finisce in `dati/sched1_bound.csv`
- Da lì la leggono la dispensa, la pagina del sito e `verifica_numeri.py`
- **Un solo posto in cui il numero esiste**

# I quattro numeri sull'istanza di assegnamento

**I quattro numeri che ogni esercizio produce**



# Riepilogo

- Dati $\rightarrow$ modello $\rightarrow$ euristica e verifica $\rightarrow$ LP e duale $\rightarrow$ MIP $\rightarrow$ tabella $\rightarrow$ figure e notebook
- Ogni numero è controllato da un `assert`
- Sull'istanza di assegnamento:
  $LB = 10 \leq z(LP) = 53/5 \leq z(MILP) = 11 = UB$
- Un solo posto in cui il numero esiste: il file CSV in `dati/`

# Eseguire il codice

# Introduzione: eseguire il codice

- Il codice del corso si può eseguire in due modi: in locale o su Colab
- In entrambi i casi basta la licenza inclusa in `gurobipy`
- Vediamo i comandi e perché la licenza è sufficiente

# In locale

Dalla cartella `python/`:

```
python3 -m venv .venv && source .venv/bin/activate
pip install gurobipy pandas matplotlib
python3 fam07_1_assegnamento.py  # un capitolo
python3 esegui_tutti.py          # tutti, piu' i notebook
python3 verifica_numeri.py       # gli assert sui numeri
```

# Su Colab

- Ogni capitolo ha un notebook che si apre dal badge sulla pagina del sito
- La prima cella installa `gurobipy` e scarica i tre moduli comuni (`stile.py`, `mip.py`, `euristiche.py`) se non sono già presenti
- Sul computer non serve installare niente

## Osservazione 7: La licenza inclusa in `gurobipy` basta, e perché

*La licenza size-limited del pacchetto pip permette modelli fino a **2000 variabili e 2000 vincoli**. Le istanze di questo corso sono piccolissime — il modello più grande ha poche decine di variabili — e ci stanno tutte con enorme margine.*

*Per istanze più grandi si attiva gratuitamente la licenza accademica da portal.gurobi.com. Se un modello supera il limite, Gurobi lo segnala con un errore esplicito a `optimize()`: non produce silenziosamente un risultato sbagliato.*

# Riepilogo

- In locale: un ambiente virtuale, `pip install gurobipy pandas matplotlib`, poi gli script
- Su Colab: il notebook installa tutto da solo
- La licenza del pacchetto pip (2000 variabili, 2000 vincoli) basta per tutto il corso

# Riepilogo finale (1/2)

**Scrivere il modello**:

- Poche istruzioni: `Model`, `addVars`, `setObjective`, `addConstrs`, `optimize`
- Il dominio (`vtype`, `lb`) fa parte del modello
- Una `addConstrs` per famiglia, con il nome del modello scritto; controllo con `m.write`

**Leggere i risultati**:

- `Status` prima di tutto, poi `SolCount`, `ObjVal`, `ObjBound`
- Se il solver non ha finito si riporta un intervallo, non un ottimo

# Riepilogo finale (2/2)

**Tolleranze e rilassamento**:

- Interi entro una tolleranza: confronti con `> 0.5`
- `relax()` dà $z(LP)$ e i duali

**Problemi e protocollo**:

- Bin packing, makespan e TSP (MTZ) sono i modelli di riferimento per le euristiche
- Il protocollo porta dai dati alla tabella $LB \leq z(LP) \leq z(MILP) \leq UB$