Decomposition methods in Integer Programming

Benders Decomposition ● Constraint Programming

29 May 2025

E-BUSINESS RESEARCH CENTER
ELTRUN
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Athens University of Economics and Business
Department of Management Science and Technology

Ioannis Avgerinos ● iavgerinos@aueb.gr

Timeline of Integer Programming:

- **1940's:** Construction of *Simplex algorithm*
- **1958:** Construction of *Cutting Planes* algorithms
- **1960:** Construction of *Branch-and-Bound*

Timeline of Integer Programming:

- **1940's:** Construction of *Simplex algorithm*
- **1958:** Construction of *Cutting Planes* algorithms
- **1960:** Construction of *Branch-and-Bound*

- **Simplex algorithm** solves Linear Programs optimally.
- Most practical problems require variables of discrete values.
- Omitting integrality leads to a **relaxation** of an Integer Program.

Timeline of Integer Programming:

- **1940's:** Construction of *Simplex algorithm*
- **1958:** Construction of *Cutting Planes* algorithms
- **1960:** Construction of *Branch-and-Bound*

- **Cutting planes** algorithms start by solving the linear relaxation of the problem.
- Iteratively, additional constraints restore integrality for variables of continuous values.

Timeline of Integer Programming:

- **1940's:** Construction of *Simplex algorithm*
- **1958:** Construction of *Cutting Planes* algorithms
- **1960:** Construction of *Branch-and-Bound*

- **Cutting planes** algorithms start by solving the linear relaxation of the problem.
- Iteratively, additional constraints restore integrality for variables of continuous values.

$$min \quad x_1 + x_2 \quad 2x_1 + x_2 \geq 5 \quad x_1 + 2x_2 \geq 6 \quad x_1, x_2 \geq 0$$

→ $x_1 = 1.333$, $x_2 = 2.333$, $x_1 + x_2 = 3.666$ **Non-integer solution**

→ $x_1 + x_2 \geq 4$

→ $x_1 = 2$, $x_2 = 2$, $x_1 + x_2 = 4$ **Integer solution**

Timeline of Integer Programming:

- **1940's:** Construction of *Simplex algorithm*
- **1958:** Construction of *Cutting Planes* algorithms
- **1960:** Construction of *Branch-and-Bound*

- **Branch-and-Bound** creates a pair of branches for each variable of continuous value.

e.g., if an integer variable x is set to the continuous value b, then one of the following should be satisfied:

- $x \geq \lceil b \rceil$ e.g., if $b = 2.3$, then $x \geq 3$
- $x \leq \lfloor b \rfloor$ e.g., if $b = 2.3$, then $x \leq 2$

Timeline of Integer Programming:

- **1940's:** Construction of *Simplex algorithm*
- **1958:** Construction of *Cutting Planes* algorithms
- **1960:** Construction of *Branch-and-Bound*

Integer Programming methods combine the following techniques:

- Solving a **relaxation** to obtain a **dual bound**
- Restoring feasibility over the solution of the relaxation to obtain a **primal bound**, by generating linear inequalities called **cuts**.

Benders Decomposition

Classical variant and dual-derived cuts

Benders Decomposition

Overview

In 1962, Jacques F. Benders presented a set of *"partitioning procedures"*, which decompose a MILP into a **master problem** (a relaxation of the original MILP) and a set of **subproblems.**

Benders Decomposition

Overview

In 1962, Jacques F. Benders presented a set of *"partitioning procedures"*, which decompose a MILP into a **master problem** (a relaxation of the original MILP) and a set of **subproblems.**

$$\mathcal{P}: \quad min \quad f(y) + g(x)$$
$$C(y), C(x), C(y,x)$$

$$y \text{ integer variables}$$
$$x \text{ continuous variables}$$

$f(y), \mathrm{g(x)}$:	Non-negative linear cost functions
$C(y), C(x), C(y,x)$:	Constraints
y:	Integer variables
x:	Continuous variables

Benders Decomposition

Overview

In 1962, Jacques F. Benders presented a set of *"partitioning procedures"*, which decompose a MILP into a **master problem** (a relaxation of the original MILP) and a set of **subproblems.**

$$\mathcal{P}: \quad min \quad f(y) + g(x)$$
$$C(y), C(x), C(y,x)$$

$$y \text{ integer variables}$$
$$x \text{ continuous variables}$$

$f(y), \mathrm{g(x)}$: Non-negative linear cost functions
$C(y), C(x), C(y,x)$: Constraints

y: Integer variables
x: Continuous variables

We separate integer and continuous variables:

$$\mathcal{M}: \quad min \quad f(y)$$
$$C(y)$$

$$y \text{ integer variables}$$

Evidently, problem $\mathcal{M}$ is a relaxation of $\mathcal{P}$:

- The minimum objective value of $\mathcal{M}$ is a lower bound of the minimum objective value of $\mathcal{P}$.
- All constraints of $\mathcal{M}$ hold in $\mathcal{P}$, while there are constraints of $\mathcal{P}$ which are not part of $\mathcal{M}$.

Benders Decomposition

Overview

In 1962, Jacques F. Benders presented a set of *"partitioning procedures"*, which decompose a MILP into a **master problem** (a relaxation of the original MILP) and a set of **subproblems.**

$$\mathcal{P}: \quad min \quad f(y) + g(x)$$
$$C(y), C(x), C(y,x)$$

$$y \text{ integer variables}$$
$$x \text{ continuous variables}$$

$f(y), g(x)$: Non-negative linear cost functions
$C(y), C(x), C(y,x)$: Constraints

y: Integer variables
x: Continuous variables

We separate integer and continuous variables:

If $\bar{y}$ is the solution of $\mathcal{M}$:

$$\mathcal{S}: \quad min \quad g(x) + f(\bar{y})$$
$$C(x, \bar{y}), C(x)$$

$$x \text{ continuous variables}$$

If $\mathcal{S}$ has a feasible solution, then its objective value is an **upper bound** of the optimal objective value of $\mathcal{P}$:

- All constraints $C(y)$ are satisfied by construction.
- The remaining constraints of $\mathcal{P}$ are also taken into consideration.

Benders Decomposition

Overview

In 1962, Jacques F. Benders presented a set of *"partitioning procedures"*, which decompose a MILP into a **master problem** (a relaxation of the original MILP) and a set of **subproblems.**

$$\mathcal{P}: \quad min \quad f(y) + g(x)$$
$$C(y), C(x), C(y,x)$$

$$y \text{ integer variables}$$
$$x \text{ continuous variables}$$

$f(y), \mathrm{g(x)}$: Non-negative linear cost functions
$C(y), C(x), C(y,x)$: Constraints

y: Integer variables
x: Continuous variables

We separate integer and continuous variables:

We add an inequality to $\mathcal{M}$, ensuring that if $y = \bar{y}$, then the objective value is set to the upper bound:

$$\mathcal{M}: \quad min \quad f(y) + \theta$$
$$C(y)$$
$$if\ y = \bar{y} \rightarrow \theta = g(x)$$

$$y \text{ integer variables}$$
$$\theta \geq 0$$

Iteratively solving $\mathcal{M}$ and $\mathcal{S}$ and generating new inequalities leads to a convergence.

Benders Decomposition

Overview

In 1962, Jacques F. Benders presented a set of *"partitioning procedures"*, which decompose a MILP into a **master problem** (a relaxation of the original MILP) and a set of **subproblems.**

The construction of new inequalities is derived of **duality** theory; this is why the subproblem(s) should consist of **strictly continuous variables**.

Benders Decomposition

Example

$\mathcal{P}$: min $y + 2 \cdot x$

$x + y \geq 5$

$y \in [0, 10]$

$x \geq 0$

is decomposed into $\mathcal{M}$ and $\mathcal{S}$:

$\mathcal{M}$: min y

$y \in [0, 10]$

$\bar{y}$

$\mathcal{S}$: min $2 \cdot x$

$x \geq 5 - \bar{y}$

$x \geq 0$

Benders Decomposition

Example

$\mathcal{P}$: $min \quad y + 2 \cdot x$

$x + y \geq 5$

$y \in [0, 10]$

$x \geq 0$

is decomposed into $\mathcal{M}$ and $\mathcal{S}$:

$\mathcal{M}$: $min \quad y$

$y \in [0, 10]$

$\bar{y}$

Dual of $\mathcal{S}$: $max \quad u \cdot (5 - \bar{y})$

$u \leq 2$

$u \geq 0$

Benders Decomposition

Example

$\mathcal{P}$: $min \quad y + 2 \cdot x$

$x + y \geq 5$

$y \in [0, 10]$

$x \geq 0$

is decomposed into $\mathcal{M}$ and $\mathcal{S}$:

$\mathcal{M}$: $min \quad y$

$y \in [0, 10]$

$\bar{y}$

$\theta \geq \bar{u} \cdot (5 - y)$

Dual of $\mathcal{S}$: $max \quad u \cdot (5 - \bar{y})$

$u \leq 2$

$u \geq 0$

Benders Decomposition

Example

$\mathcal{P}$: min $y + 2 \cdot x$

$x + y \geq 5$

$y \in [0, 10]$

$x \geq 0$

is decomposed into $\mathcal{M}$ and $\mathcal{S}$:

$\mathcal{M}$: min $y + \theta$

$\theta \geq \bar{u} \cdot (5 - y)$

$y \in [0, 10]$

$\theta \geq 0$

$\bar{y}$

$\theta \geq \bar{u} \cdot (5 - y)$

Dual of $\mathcal{S}$: max $u \cdot (5 - \bar{y})$

$u \leq 2$

$u \geq 0$

Benders Decomposition

Example

Benders decomposition algorithm:

1. Set LB = 0, UB = ∞, t = 1;
2. While(LB < UB):
3. Solve $\mathcal{M}\rightarrow$ get LB, $\bar{y}$;
4. Solve $\mathcal{S} \rightarrow$ get $\bar{u}$, UB;
5. Add cut $\theta \geq \bar{u} \cdot (5 - y)$ to $\mathcal{M}$;
6. t = t + 1;
7. End while;

```
print("-----------------------------------------------------")
print("Iteration    Lower bound     Upper bound     Gap (%)")
print("-----------------------------------------------------")
master = masterProblem() # Construct the master problem 'M'
lowerBound, upperBound, iteration = 0, inf, 1 # Initialize lower bound, upper bound, number of iterations
while(lowerBound < upperBound): # The procedure is repeated until a convergence is reached.
    results = opt.solve(master, tee = False) # Solve 'M'
    y = value(master.y) # Get the value of variable 'y'
    lowerBound = value(master.obj) # Update the value of the lower bound
    subproblem = subProblem(y) # Construct the subproblem 'S'
    results = opt.solve(subproblem, tee = False) # Solve 'S'
    u = value(subproblem.u) # Get the value of variable 'u'
    if value(subproblem.obj) + lowerBound < upperBound: # Update the value of the upper bound, in case of improvement
        upperBound = value(subproblem.obj) + lowerBound
    master.constraints.add(master.θ >= u*(5 - master.y)) # Add cut to 'M'
    print(f"{iteration:<12}{lowerBound:<16}{upperBound:<16}{round(100*(upperBound - lowerBound)/upperBound, 2)}%")
    iteration += 1 # Increase the number of iterations by 1
```

```
-----------------------------------------------------
Iteration    Lower bound     Upper bound     Gap (%)
-----------------------------------------------------
1            0.0             10.0            100.0%
2            5.0             5.0             0.0%
-----------------------------------------------------
```

Combinatorial cuts

Cuts beyond duality theory

Combinatorial cuts

Traveling Salesman Problem

Nodes to visit: $25\ [1, \dots, 25]$

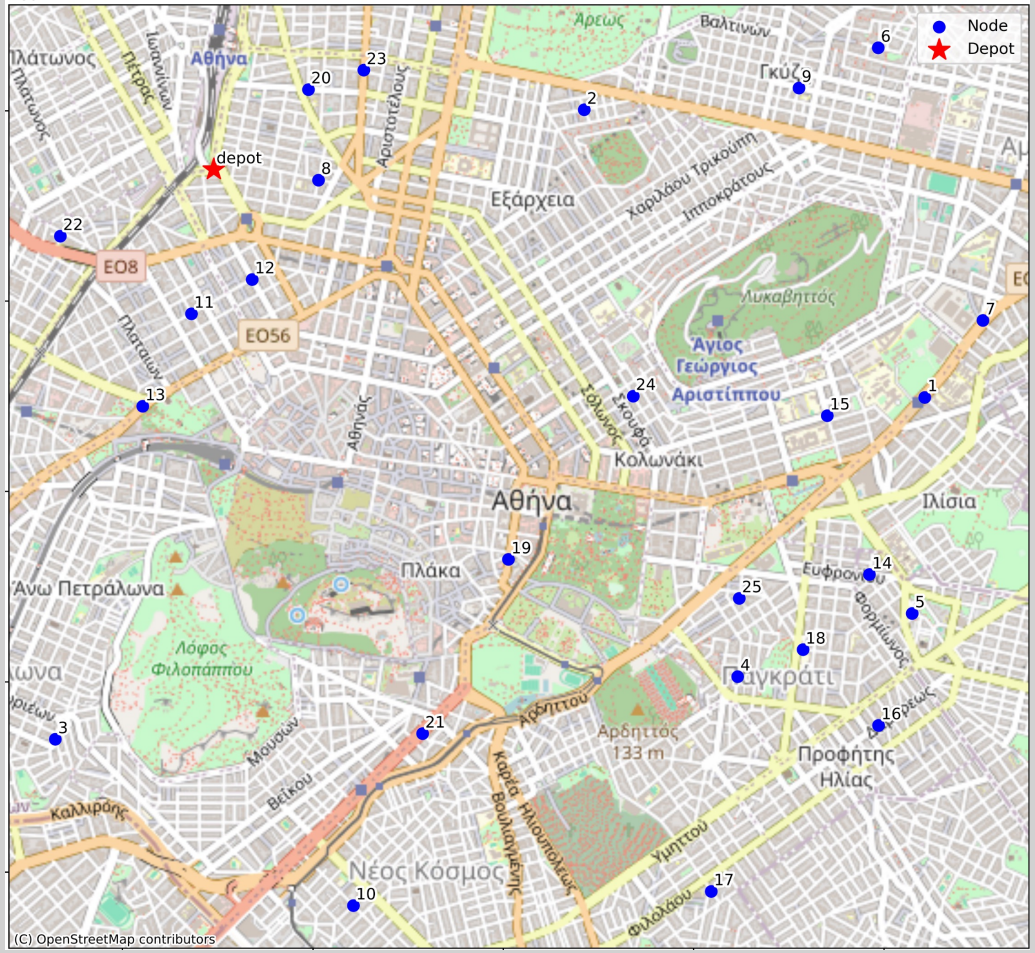
Depot: $depot$

Distances: $d_{ij}: i, j \in [1, \dots, 25, depot]$

Objective: Minimize total covered distance

Constraints:

- Each node is visited exactly once.



Combinatorial cuts

Traveling Salesman Problem

$$
\begin{aligned}
min \quad & \textstyle\sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij} && \\
& \textstyle\sum_{i \in J} x_{ij} = 1 && \forall j \in J \\
& \textstyle\sum_{i \in J} x_{ji} = 1 && \forall j \in J \\
& u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) && \forall i \in J, j \in J \backslash \{depot\} \\
& x_{ij} \in \{0, 1\} && \forall i \in J, j \in J, i \neq j \\
& u_j \in [0, |J|] && \forall j \in J
\end{aligned}
$$

Combinatorial cuts

Traveling Salesman Problem

$$
\begin{aligned}
min \quad & \sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij} & \\
& \sum_{i \in J} x_{ij} = 1 & \forall j \in J \\
& \sum_{i \in J} x_{ji} = 1 & \forall j \in J \\
& u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) & \forall i \in J, j \in J \backslash \{depot\} \\
& & \\
& x_{ij} \in \{0, 1\} & \forall i \in J, j \in J, i \neq j \\
& u_j \in [0, |J|] & \forall j \in J
\end{aligned}
$$

$x_{ij} = 1 \rightarrow$ nodes i and j are connected

$x_{ij} = 0 \rightarrow$ nodes i and j are not connected

Combinatorial cuts

Traveling Salesman Problem

$$
\begin{aligned}
min \quad & \sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij} & \\
& \sum_{i \in J} x_{ij} = 1 & \forall j \in J \\
& \sum_{i \in J} x_{ji} = 1 & \forall j \in J \\
& u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) & \forall i \in J, j \in J \backslash \{depot\} \\
& & \\
& x_{ij} \in \{0, 1\} & \forall i \in J, j \in J, i \neq j \\
& u_j \in [0, |J|] & \forall j \in J
\end{aligned}
$$

u_j: Order of visiting node j

e.g., $u_j = 0 \rightarrow$ node j is visited first

Combinatorial cuts

Traveling Salesman Problem

$$min \quad \sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij}$$

$$\sum_{i \in J} x_{ij} = 1 \qquad \forall j \in J$$

$$\sum_{i \in J} x_{ji} = 1 \qquad \forall j \in J$$

$$u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) \qquad \forall i \in J, j \in J \backslash \{depot\}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in J, j \in J, i \neq j$$

$$u_j \in [0, |J|] \qquad \forall j \in J$$

Objective function: minimization of covered distance

Combinatorial cuts

Traveling Salesman Problem

$$min \quad \sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij}$$

$$\sum_{i \in J} x_{ij} = 1 \qquad \forall j \in J$$

$$\sum_{i \in J} x_{ji} = 1 \qquad \forall j \in J$$

$$u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) \qquad \forall i \in J, j \in J \backslash \{depot\}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in J, j \in J, i \neq j$$

$$u_j \in [0, |J|] \qquad \forall j \in J$$

Each node is visited exactly once:

- Each node is the origin of a trip.
- Each node is the destination of a trip.

Combinatorial cuts

Traveling Salesman Problem

$$
\begin{aligned}
min \quad & \sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij} & \\
& \sum_{i \in J} x_{ij} = 1 & \forall j \in J \\
& \sum_{i \in J} x_{ji} = 1 & \forall j \in J \\
& u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) & \forall i \in J, j \in J \backslash \{depot\} \\
& x_{ij} \in \{0, 1\} & \forall i \in J, j \in J, i \neq j \\
& u_j \in [0, |J|] & \forall j \in J
\end{aligned}
$$

If $x_{ij} = 1$, then the order of visiting node j is larger than the order of visiting node i. If $x_{ij} = 0$, then the right-hand side is always greater/equal than the left-hand side.

Combinatorial cuts

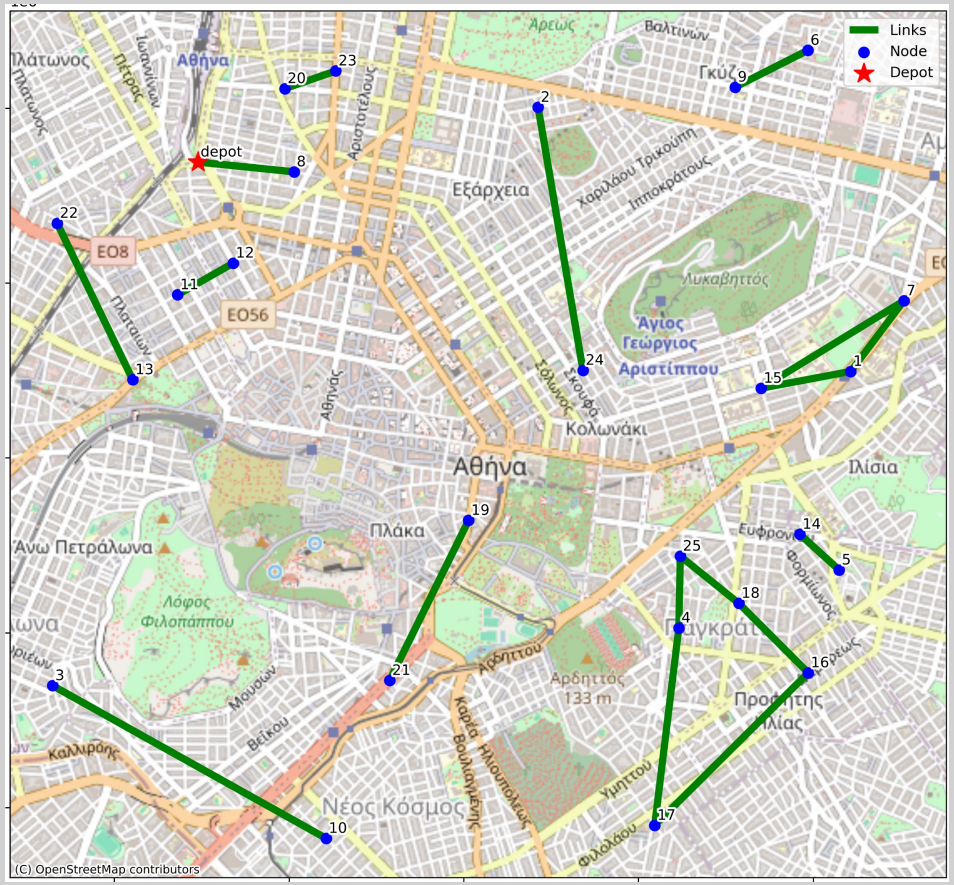
Traveling Salesman Problem

$$
\begin{aligned}
min \quad & \sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij} & \\
& \sum_{i \in J} x_{ij} = 1 & \forall j \in J \\
& \sum_{i \in J} x_{ji} = 1 & \forall j \in J \\
& u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) & \forall i \in J, j \in J \backslash \{depot\} \\
& & \\
& x_{ij} \in \{0, 1\} & \forall i \in J, j \in J, i \neq j \\
& u_j \in [0, |J|] & \forall j \in J
\end{aligned}
$$

Complicating constraints: they imply a weak bound → slow convergence to optimality

Combinatorial cuts

Traveling Salesman Problem



Without these constraints: **Subtours**

\+ The problem is very easy – it can be solved optimally in a few seconds for thousands of nodes.

\- The solution does not define a single route which visits all nodes - **infeasible**.

Combinatorial cuts

Traveling Salesman Problem

A relaxation of the problem provides a lower bound:

$$
\begin{aligned}
min \quad & \textstyle\sum_{i \in J} \sum_{j \in J} d_{ij} \cdot x_{ij} & \\
& \textstyle\sum_{i \in J} x_{ij} = 1 & \forall j \in J \\
& \textstyle\sum_{i \in J} x_{ji} = 1 & \forall j \in J \\
& x_{ij} \in \{0, 1\} & \forall i \in J, j \in J, i \neq j
\end{aligned}
$$

If the solution has subtours, then add an inequality which **prevents** the relaxation from computing the same evidently infeasible solution.

Combinatorial cuts

Traveling Salesman Problem

A relaxation of the problem provides a lower bound:

$$
\begin{aligned}
min \quad & \textstyle\sum_{i\in J}\sum_{j\in J} d_{ij}\cdot x_{ij} & \\
& \textstyle\sum_{i\in J} x_{ij} = 1 & \forall j \in J \\
& \textstyle\sum_{i\in J} x_{ji} = 1 & \forall j \in J \\
& x_{ij} \in \{0,1\} & \forall i \in J, j \in J, i \neq j
\end{aligned}
$$

If $\bar{x}_{ij}$ are the values of variables x_{ij}, then:

$$\sum_{(i,j):\bar{x}_{ij}=1} x_{ij} \leq |J| - 1$$

Combinatorial cuts

Traveling Salesman Problem

A relaxation of the problem provides a lower bound:

$$
\begin{aligned}
min \quad & \textstyle\sum_{i\in J}\sum_{j\in J} d_{ij}\cdot x_{ij} & \\
& \textstyle\sum_{i\in J} x_{ij} = 1 & \forall j \in J \\
& \textstyle\sum_{i\in J} x_{ji} = 1 & \forall j \in J \\
& x_{ij} \in \{0,1\} & \forall i \in J, j \in J, i \neq j
\end{aligned}
$$

If $\bar{x}_{ij}$ are the values of variables x_{ij}, then:

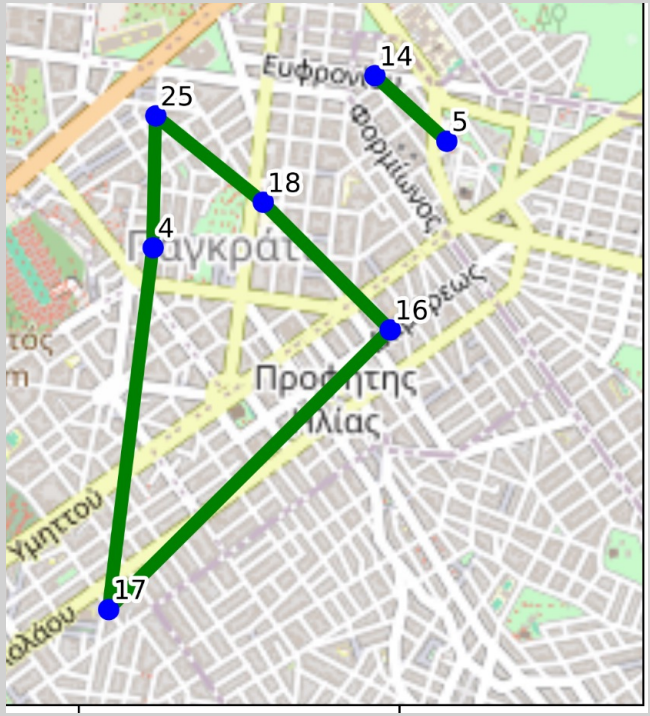
$$\sum_{(i,j):\bar{x}_{ij}=1} x_{ij} \leq |J| - 1$$

If $x_{ij} = 1$ for all trips (i,j): $\bar{x}_{ij} = 1$, then the left-hand side is set to $|J|$.
Adding this cut ensures that at least one trip is eliminated.

Combinatorial cuts

Traveling Salesman Problem

A combinatorial cut can be generated for each subtour:



$$\left.\begin{aligned} x_{4,17} &= 1 \\ x_{17,16} &= 1 \\ x_{16,18} &= 1 \\ x_{18,25} &= 1 \\ x_{25,4} &= 1 \end{aligned}\right\} \quad x_{4,17} + x_{17,16} + x_{16,18} + x_{18,25} + x_{25,4} \leq 4$$

or if S is the set of subtours, and each subtour s is a subset of trips (i, j): $\bar{x}_{ij} = 1$, then:

$$\sum_{(i,j) \in s} x_{ij} \leq |s| - 1 \qquad \forall s \in S$$

A Branch-and-Cut algorithm:

1. Set LB = 0, t = 1;
2. While True:
3. Solve relaxation → get LB, $\bar{x}_{ij}$, set $S = \emptyset$;
4. For each node $j \in J$:
5. Generate a subtour s starting from j, append s to S;
6. Add combinatorial cut $\sum_{(i,j)\in S} x_{ij} \leq |s| - 1$;
7. End for;
8. If $S = \emptyset$:
9. End while;
10. Else:
11. t = t + 1;

Combinatorial cuts

Traveling Salesman Problem

```
print("-----------------------------------------------------")
print(f"Iteration       Lower bound     Time")
print("-----------------------------------------------------")
startTime = time.time()
plotRoute(0, [])
milp = relaxation(nodes, distances)
iteration = 1
while True:
    results = opt.solve(milp, tee = False, timelimit = 60)
    links = [(i, j) for i in milp.Nodes for j in milp.Nodes if i != j and value(milp.x[i, j]) > 0.9]
    plotRoute(iteration, links)
    for j in milp.Nodes:
        route = generateRoute(milp, j)
        if len(route) == len(nodes)+1:
            break
        else:
            milp.constraints.add(sum(milp.x[route[i-1], route[i]] for i in range(1, len(route))) <= len(route)-2)
    print(f"{iteration}             {value(milp.obj)}           {int(time.time() - startTime)}")
    if len(route) == len(nodes)+1:
        break
    else:
        iteration += 1
```

```
-----------------------------------------------------
Iteration       Lower bound     Time
-----------------------------------------------------
1               1619.0          9
2               1697.0          13
3               1729.0          17
4               1729.0          21
5               1748.0          25
6               1751.0          29
[Finished in 30.8s]
```

Constraint Programming

Introduction - TSP

Constraint Programming

Example: Sudoku puzzle

Sudoku game:

A 9x9 grid must be filled with numeric values from 1 to 9.

- At each row, each one of the 9 cells should have a unique value.
- At each column, each one of the 9 cells should have a unique value.
- Each 3x3 subgrid (9 cells) should have a unique value per cell.

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Example: Sudoku puzzle

Sudoku game:

Given a matrix $sudoku_{i,j}$, $i = \{1, \dots, 9\}$, $j = \{1, \dots, 9\}$ and fixed values $\overline{sudoku}_{i,j}$ for certain cells, we should solve a satisfiability problem for which the following conditions hold:

- $sudoku_{i,j} \neq sudoku_{i,k}$ $\quad \forall i = \{1, \dots, 9\}, j = \{1, \dots, 9\},\ k = \{1, \dots, 9\} \neq j$
- $sudoku_{i,j} \neq sudoku_{k,j}$ $\quad \forall j = \{1, \dots, 9\}, i = \{1, \dots, 9\},\ \ k = \{1, \dots, 9\} \neq i$
- $sudoku_{3 \cdot i+a,3 \cdot j+b} \neq sudoku_{3 \cdot i+c,3 \cdot j+d}$ $\forall i = \{0,1,2\},\ j = \{0,1,2\},$ $a = \{1,2,3\}, b = \{1,2,3\},$ $c = \{1,2,3\} \neq a,$ $d = \{1,2,3\} \neq b$
- $sudoku_{i,j} = \overline{sudoku}_{i,j}$ $\quad \forall (i,j) \colon \exists\ \overline{sudoku}_{i,j}$

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Example: Sudoku puzzle

Sudoku game:

Given a matrix $sudoku_{i,j}$, $i = \{1, \dots, 9\}$, $j = \{1, \dots, 9\}$ and fixed values $\overline{sudoku}_{i,j}$ for certain cells, we should solve a satisfiability problem for which the following conditions hold:

- $sudoku_{i,j} \neq sudoku_{i,k}$ $\quad \forall i = \{1, \dots, 9\}, j = \{1, \dots, 9\},\ k = \{1, \dots, 9\} \neq j$
- $sudoku_{i,j} \neq sudoku_{k,j}$ $\quad \forall j = \{1, \dots, 9\}, i = \{1, \dots, 9\},\ \ k = \{1, \dots, 9\} \neq i$
- $sudoku_{3 \cdot i+a,3 \cdot j+b} \neq sudoku_{3 \cdot i+c,3 \cdot j+d}$ $\quad \forall i = \{0,1,2\},\ j = \{0,1,2\},$ $a = \{1,2,3\}, b = \{1,2,3\},$ $c = \{1,2,3\} \neq a,$ $d = \{1,2,3\} \neq b$
- $sudoku_{i,j} = \overline{sudoku}_{i,j}$ $\quad \forall (i,j) \colon \exists\ \overline{sudoku}_{i,j}$

9	1	3	4	2	7	5	8	6
6	8	7	9	1	5	3	2	4
2	5	4	6	8	3	1	7	9
4	7	9	1	3	2	6	5	8
1	6	2	5	9	8	7	4	3
5	3	8	7	6	4	2	9	1
3	4	5	8	7	1	9	6	2
7	2	6	3	4	9	8	1	5
8	9	1	2	5	6	4	3	7

Constraint Satisfaction Problem (CSP):

A CSP consists of:

- a **finite** set of variables
- a **domain** – a **finite** set of values – for each variable
- a **finite** set of constraints – logic relations over the variables

The solution of a CSP is a **complete** (for all variables) and **consistent** (it satisfies all constraints) assignment of values to variables.

Constraint Satisfaction Problem (CSP):

A CSP consists of:

- a **finite** set of variables - $sudoku_{i,j}$
- a **domain** – a **finite** set of values – for each variable – 1 to 9
- a **finite** set of constraints – logic relations over the variables – the uniqueness of values in rows/columns/3x3 subgrids

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming (CP):

Constraint Programming is the computer implementation of an algorithm for solving a CSP [Brailsford et al., 1999].

Integer “Programming” or Constraint “Programming”?

In Integer Programming, “programming” refers to **mathematical programming**, as introduced by George Dantzig, creator of the Simplex algorithm.

In Constraint Programming, “programming” refers to the translation of logic constraints from natural language to a programming language.

CP for combinatorial optimisation problems:

- Finds feasible solutions
- Chooses the **best solution** (i.e., the feasible solution which has the minimum/maximum value of a designated function).

- **Weaker than Integer Programming** at proving optimality
- Not necessarily weaker at finding better solutions – multiple feasible solutions are explored more easily than in an IP framework

Constraint Programming

Principles

Each CP solver may incorporate different algorithms to explore for feasible solutions, but fundamental principles are globally implemented:

- CP search
- Backtracking
- Propagation
- Pruning

Constraint Programming

Principles

Each CP solver may incorporate different algorithms to explore for feasible solutions, but fundamental principles are globally implemented:

- CP search – exploring possible assignments of values to variables
- Backtracking – reverting to previous decisions in case of infeasibilities
- Propagation – adding new constraints based on current decisions
- Pruning – removing inconsistent values from domains

Constraint Programming

Principles

In the previous example:

- $sudoku_{i,j} \neq sudoku_{i,k} \quad \forall i = \{1, \dots, 9\}, j = \{1, \dots, 9\},\ k = \{1, \dots, 9\} \neq j$
- $sudoku_{i,j} \neq sudoku_{k,j} \quad \forall j = \{1, \dots, 9\}, i = \{1, \dots, 9\},\ k = \{1, \dots, 9\} \neq i$
- $sudoku_{3 \cdot i+a,3 \cdot j+b} \neq sudoku_{3 \cdot i+c,3 \cdot j+d} \quad \forall i = \{0,1,2\},\ j = \{0,1,2\},\ a = \{1,2,3\}, b = \{1,2,3\},\ c = \{1,2,3\} \neq a,\ d = \{1,2,3\} \neq b$
- $sudoku_{i,j} = \overline{sudoku}_{i,j} \quad \forall (i,j): \exists\ \overline{sudoku}_{i,j}$

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Principles

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:
  - There exists a cell at the same row with the same value: $sudoku_{2,8} = 2$
- As a result, we **revert** the previous decision.

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Principles

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:
  - There exists a cell at the same row with the same value: $sudoku_{2,8} = 2$
- As a result, we **revert** the previous decision.

Reverting a previous decision due to conflicts with the constraints of the problem is called **backtracking**.

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:
  - There exists a cell at the same row with the same value: $sudoku_{2,8} = 2$
- To revert the previous decision, we add the following constraint:
  - $sudoku_{2,2} \neq 2$

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:
  - There exists a cell at the same row with the same value: $sudoku_{2,8} = 2$
- To revert the previous decision, we add the following constraint:
  - $sudoku_{2,2} \neq 2$

Adding constraints based on current/previous decisions is called **propagation**.

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:
  - There exists a cell at the same row with the same value: $sudoku_{2,8} = 2$
- To revert the previous decision, we add the following constraint:
  - $sudoku_{2,2} \neq 2$
  - The domain of $sudoku_{2,2}$ becomes $\{4, 8\}$.

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

In the previous example:

- We assign values to variables, in search of a feasible solution. For example, we assign value 2 at $sudoku_{2,2}$. We notice that there are conflicts in some constraints:
  - There exists a cell at the same row with the same value: $sudoku_{2,8} = 2$
- To revert the previous decision, we add the following constraint:
  - $sudoku_{2,2} \neq 2$
  - The domain of $sudoku_{2,2}$ becomes $\{4, 8\}$.

Reducing the domains of variables as a result of propagation is called **pruning**.

9	1	3				5		
6	**2**	7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Principles

9	1	3				5		
6	**8**	7					2	4
2	5	**4**		8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Modeling in CP: Logic constraints

- Disjunctions (OR)/Conjunctions (AND):

  For example, cells $sudoku_{6,3}$ and $sudoku_{7,3}$ can only get values 5 **or** 8:

  - $sudoku_{6,3} = 5 \vee sudoku_{6,3} = 8$
  - $sudoku_{7,3} = 5 \vee sudoku_{7,3} = 8$

  The same cells can not get value 1 **and** 2 **and** 3 and … 9 except for 5 or 8:

  - $sudoku_{6,3} \neq 1 \wedge sudoku_{6,3} \neq 2 \wedge \ldots sudoku_{6,3} \neq 9$
  - $sudoku_{7,3} \neq 1 \wedge sudoku_{7,3} \neq 2 \wedge \ldots sudoku_{7,3} \neq 9$

9	1	3				5		
6	**8**	7					2	4
2	5	**4**		8			7	
	7	9						
		2		9			4	3
		5,8			4		9	
	4	**5,8**			1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Modeling in CP: Conditional constraints

- If a (set of) condition(s) is True, then a constraint is imposed:

  For example, if $sudoku_{6,3} = 5$, then $sudoku_{7,3} = 8$:

  - if $sudoku_{6,3} = 5 \rightarrow sudoku_{7,3} = 8$
  - if $sudoku_{6,3} = 8 \rightarrow sudoku_{7,3} = 5$
  - if $sudoku_{7,3} = 5 \rightarrow sudoku_{6,3} = 8$
  - if $sudoku_{7,3} = 8 \rightarrow sudoku_{6,3} = 5$

9	1	3				5		
6	**8**	7					2	4
2	5	**4**		8			7	
	7	9						
		2		9			4	3
		5,8			4		9	
	4	**5,8**			1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Modeling in CP: Conditional constraints

- If a (set of) condition(s) is True, then a constraint is imposed:

  For example, if $sudoku_{6,3} = 5$, then $sudoku_{7,3} = 8$:

  - if $sudoku_{6,3} = 5 \rightarrow sudoku_{7,3} = 8$
  - if $sudoku_{6,3} = 8 \rightarrow sudoku_{7,3} = 5$
  - if $sudoku_{7,3} = 5 \rightarrow sudoku_{6,3} = 8$
  - if $sudoku_{7,3} = 8 \rightarrow sudoku_{6,3} = 5$

All these constraints are identical – **modeling flexibility.**

9	1	3				5		
6	**8**	7					2	4
2	5	**4**		8			7	
	7	9						
		2		9			4	3
		5,8			4		9	
	4	**5,8**			1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Modeling in CP: Predicates

Although each solver follows customised modeling conventions, several constraints are universally applied in CSPs. For example, imposing that all variables of a subset receive **different values** is a common constraint.

To standardise Constraint Programming modeling, the community has introduced certain **functions** which impose such common constraints. These functions are called **predicates,** and the imposed constraints are called **global constraints**.

Constraint Programming

Modeling in CP: Predicates

Although each solver follows customised modeling conventions, several constraints are universally applied in CSPs. For example, imposing that all variables of a subset receive **different values** is a common constraint.

To standardise Constraint Programming modeling, the community has introduced certain **functions** which impose such common constraints. These functions are called **predicates,** and the imposed constraints are called **global constraints**.

E.g., all solvers have incorporated a predicate called $allDifferent(X)$, ensuring that all variables of set X receive different values.

Constraint Programming

Modeling in CP: Predicates

To impose that all cells at the same row receive different values:

- $allDifferent(sudoku_{i,j} | j = \{1, \dots, 9\}) \qquad \forall i = \{1, \dots, 9\}$

meaning that variables $sudoku_{i,j}$ for a fixed index i get unique values.

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Modeling in CP: Predicates

To impose that all cells at the same row receive different values:

- $allDifferent(sudoku_{i,j} | j = \{1, \dots, 9\}) \qquad \forall i = \{1, \dots, 9\}$

meaning that variables $sudoku_{i,j}$ for a fixed index i get unique values.

Previous constraint:

- $sudoku_{i,j} \neq sudoku_{i,k} \qquad \forall i = \{1, \dots, 9\}, j = \{1, \dots, 9\}, \; k = \{1, \dots, 9\} \neq j$

Scale: 9 x 9 x 8 = 648 constraints

New constraint:

- $allDifferent(sudoku_{i,j} | j = \{1, \dots, 9\}) \qquad \forall i = \{1, \dots, 9\}$

Scale: 9 constraints

9	1	3				5		
6		7					2	4
	5			8			7	
	7	9						
		2		9			4	3
					4		9	
	4				1	9		
7		6			9			5
		1			6	4		7

Constraint Programming

Modeling in CP: Predicates

```
model = CpoModel()

# Variables
cellValues = [[model.integer_var(1, 9, name = f"valueOf_{i},{j}") for j in range(9)] for i in range(9)]

# Given values are fixed to the respective variables.
for i in range(9):
    for j in range(9):
        if sudokuMatrix[i][j] != None:
            model.add(cellValues[i][j] == sudokuMatrix[i][j])
# Each row has different values.
for i in range(9):
    model.add(model.all_diff([cellValues[i][j] for j in range(9)]))
# Each column has different values.
for j in range(9):
    model.add(model.all_diff([cellValues[i][j] for i in range(9)]))
# Each 3x3 grid has different values.
for i in range(3):
    for j in range(3):
        model.add(model.all_diff([cellValues[i*3 + k][j*3 + l] for k in range(3) for l in range(3)]))

sol = model.solve(TimeLimit = 60, trace_log = False)

if sol:
    plotGrid(sudokuMatrix, sol)
```

Constraint Programming

Modeling in CP: Traveling Salesman Problem

Nodes to visit: $25\ [1, \dots, 25]$

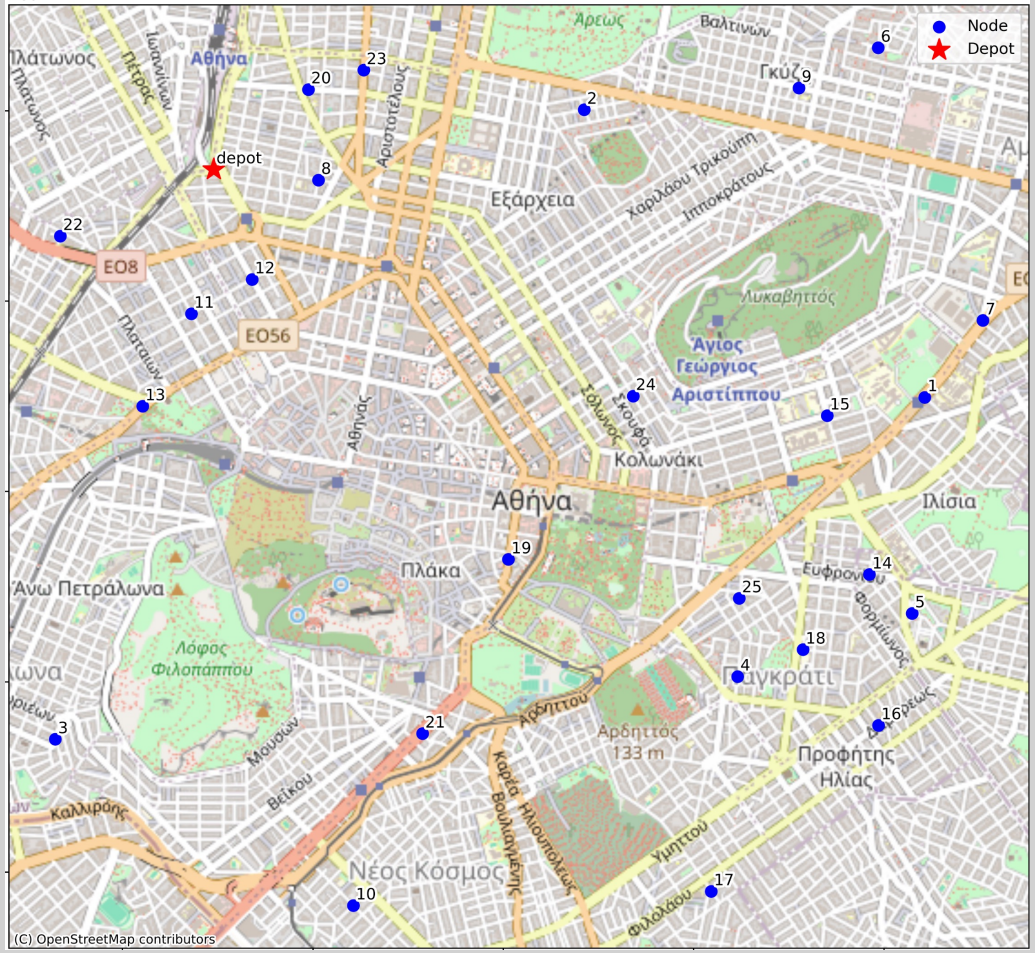
Depot: $depot$

Distances: $d_{ij}: i, j \in [1, \dots, 25, depot]$

Objective: Minimize total covered distance

Constraints:

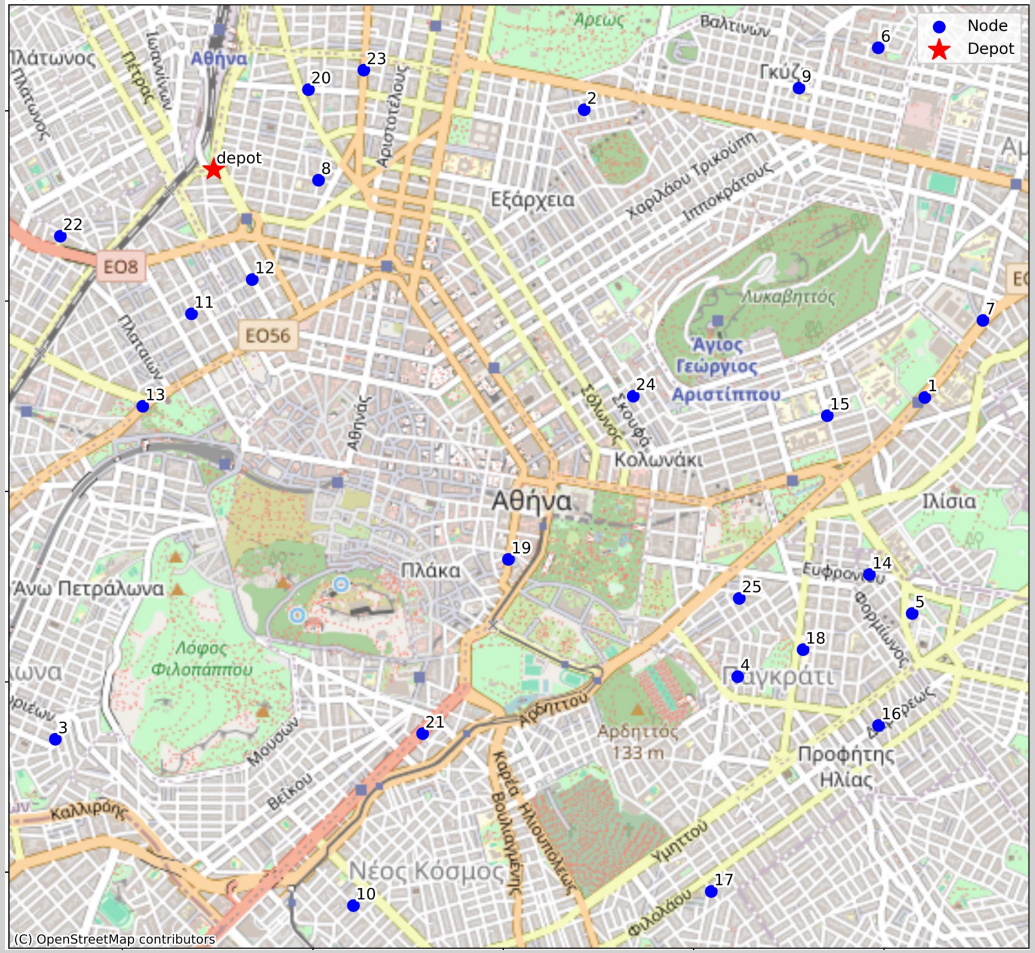
- Each node is visited exactly once.



Constraint Programming

Modeling in CP: Traveling Salesman Problem

$$
\begin{aligned}
min \quad & \textstyle\sum_{i\in J}\sum_{j\in J} d_{ij}\cdot x_{ij} & \\
& \textstyle\sum_{i\in J} x_{ij} = 1 & \forall j \in J \\
& \textstyle\sum_{i\in J} x_{ji} = 1 & \forall j \in J \\
& u_i + 1 - u_j \le |J|\cdot(1-x_{ij}) & \forall i \in J, j \in J\backslash\{depot\} \\
& & \\
& x_{ij} \in \{0,1\} & \forall i \in J, j \in J, i \neq j \\
& u_j \in [0,|J|] & \forall j \in J
\end{aligned}
$$



Constraint Programming

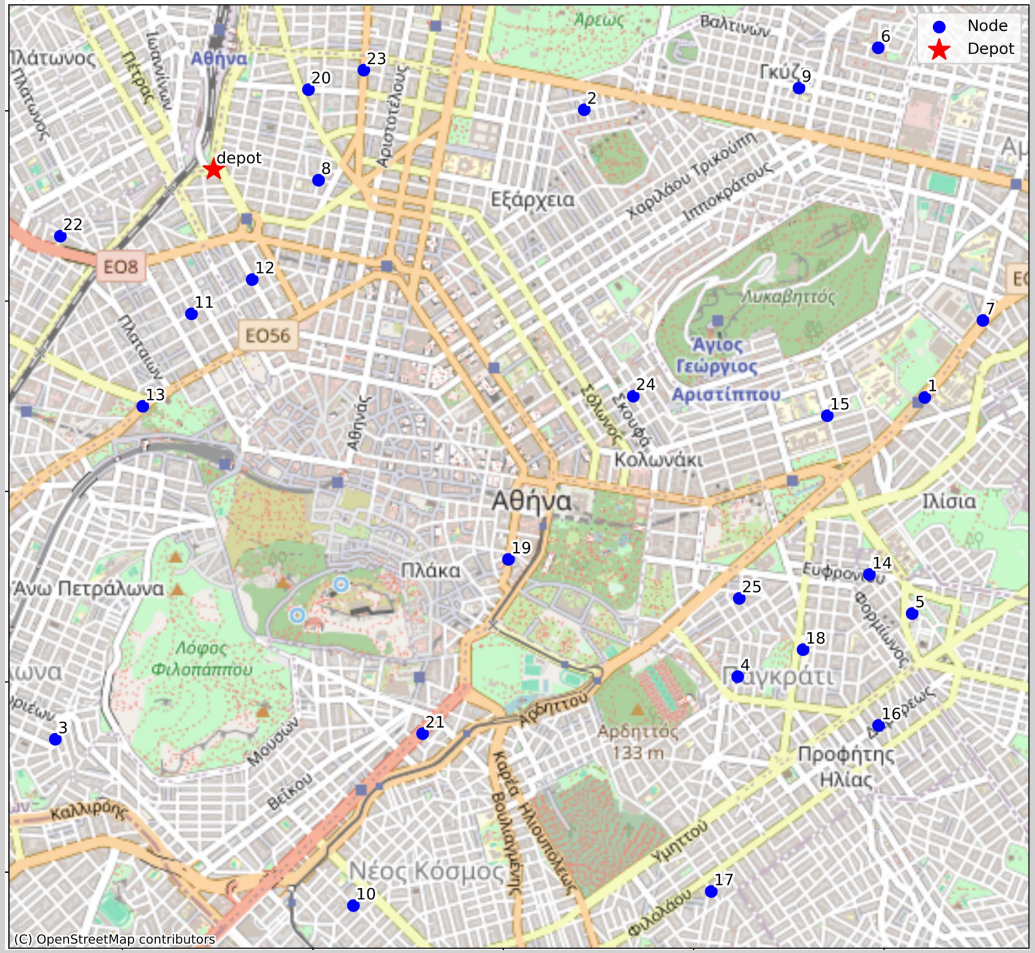
Modeling in CP: Traveling Salesman Problem

TSP can be easily modeled as a CSP:

x_i: variables indicating the **next** stop of node i

Domain of x_i: $[1, \dots, 25, depot] \backslash \{i\}$

E.g., if $x_1 = 2$, then node '2' succeeds node '1'.



Constraint Programming

Modeling in CP: Traveling Salesman Problem

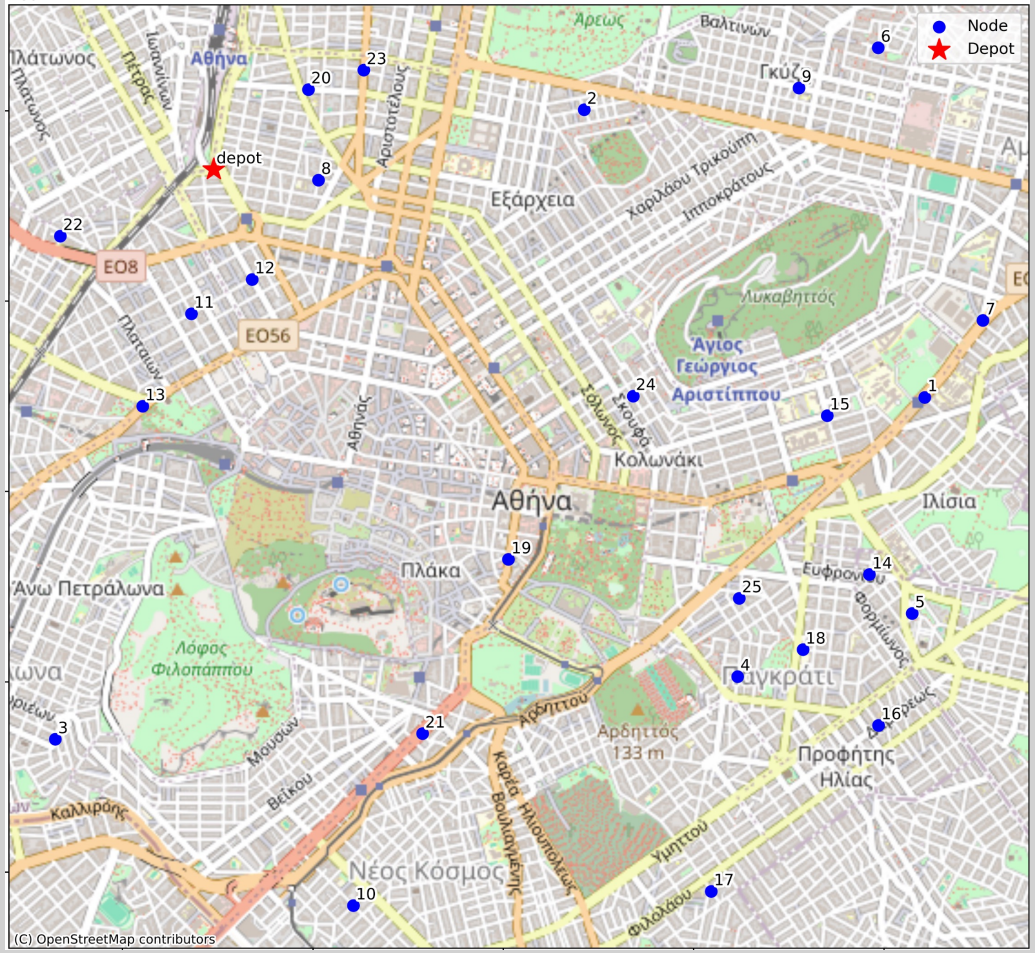
TSP can be easily modeled as a CSP:

x_i: variables indicating the **next** stop of node i

Domain of x_i: $[1, \dots, 25, depot] \backslash \{i\}$

E.g., if $x_1 = 2$, then node '2' succeeds node '1'.

$$allDifferent(x_i | i \in [1, \dots, 25, depot])$$



Constraint Programming

Modeling in CP: Traveling Salesman Problem

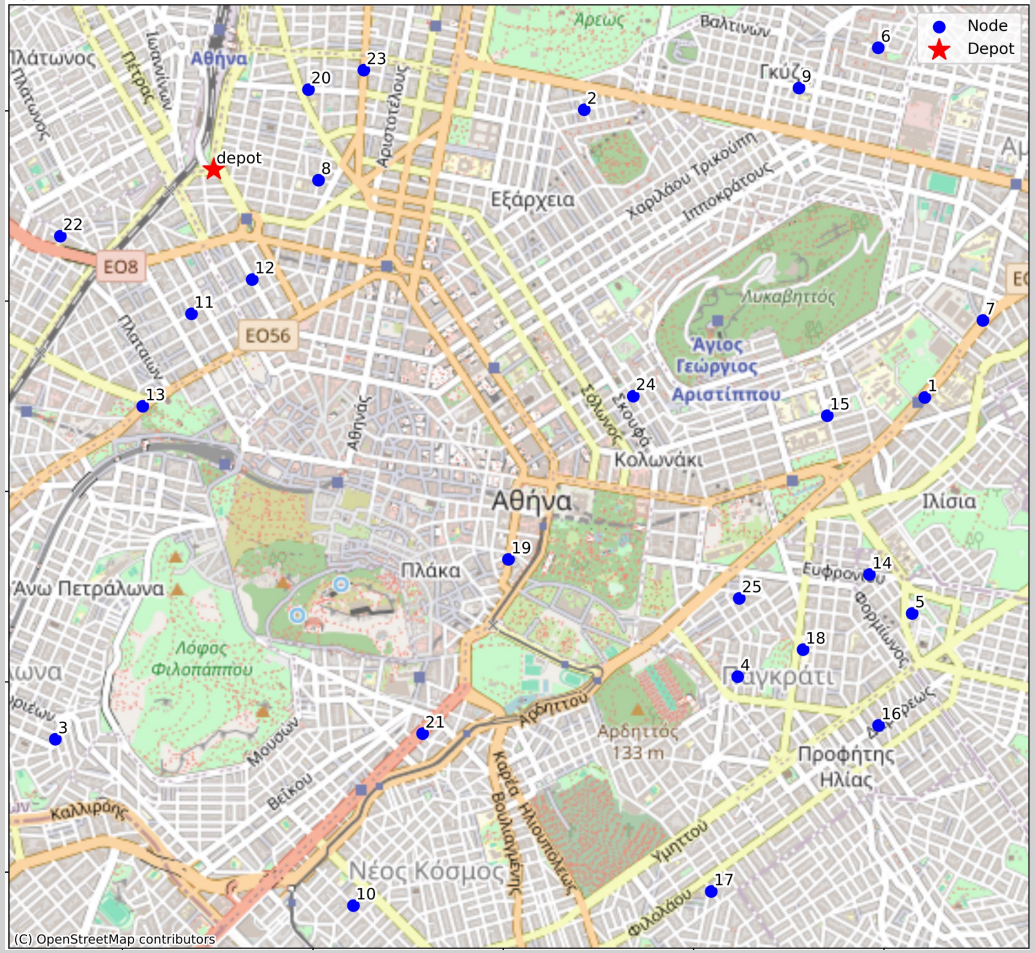
Subtour elimination constraints in IP:

$$u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) \qquad \forall i \in J, j \in J \backslash \{depot\}$$

Subtour elimination constraints in CP:

$y_i \in [0, \ldots, 25]$: the position of the node in the route

All 26 nodes are visited → all values in [0, 25] will be assigned.



Constraint Programming

Modeling in CP: element predicate

$element$ predicate:

CP modeling allows using **variables** as **indices** of other variables or parameters.
For example:

y_{x_i}: the position of the next node of i
d_{ix_i}: the distance from node i to its successor

$element$ predicate:

CP modeling allows using **variables** as **indices** of other variables or parameters.
For example:

y_{x_i}: the position of the next node of i
d_{ix_i}: the distance from node i to its successor

The $element$ predicate assigns values of variables to indices:
$element(y, x_i)$: returns y_{x_i}
$element(d_i, x_i)$: returns d_{ix_i}

Constraint Programming

Modeling in CP: Traveling Salesman Problem

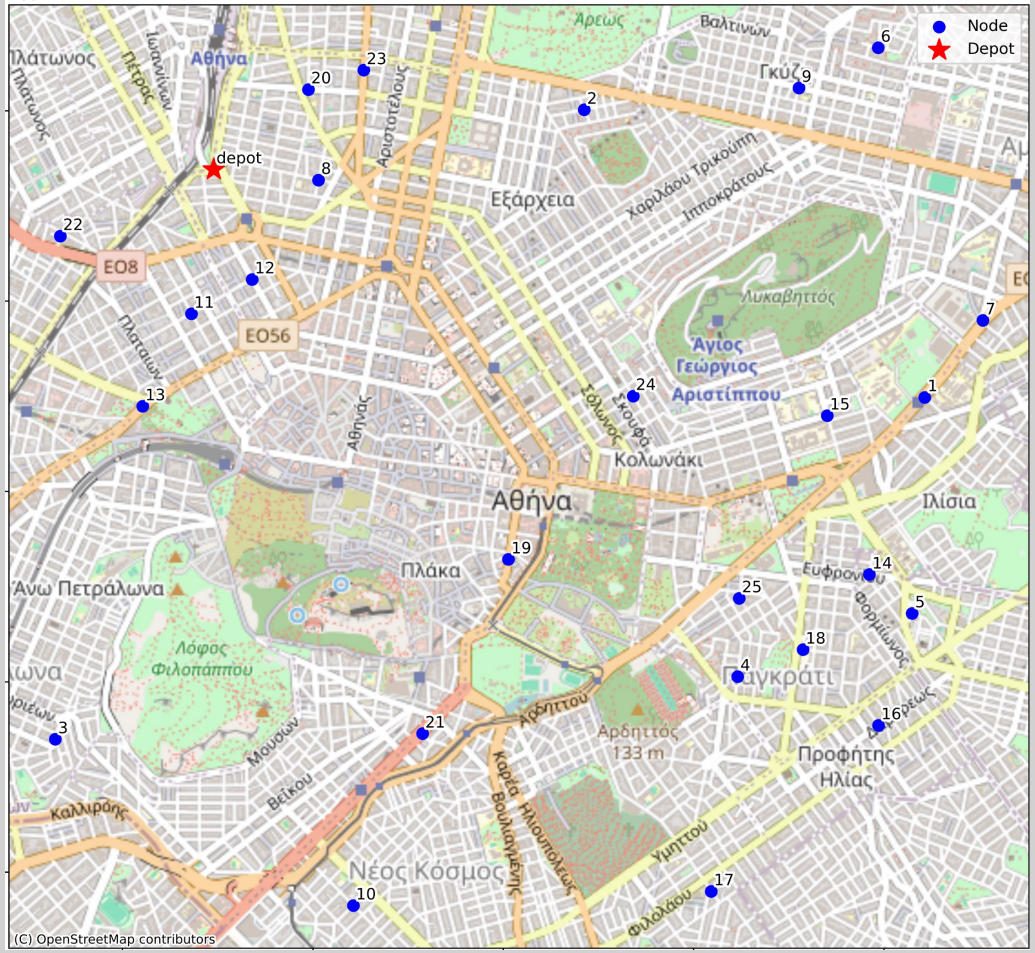
Subtour elimination constraints in IP:

$$u_i + 1 - u_j \leq |J| \cdot (1 - x_{ij}) \qquad \forall i \in J, j \in J \backslash \{depot\}$$

Subtour elimination constraints in CP:

$y_i \in [0, \ldots, 25]$: the position of the node in the route

$$element(y, x_i) = y_i + 1 \;\; \forall i \neq depot$$



Constraint Programming

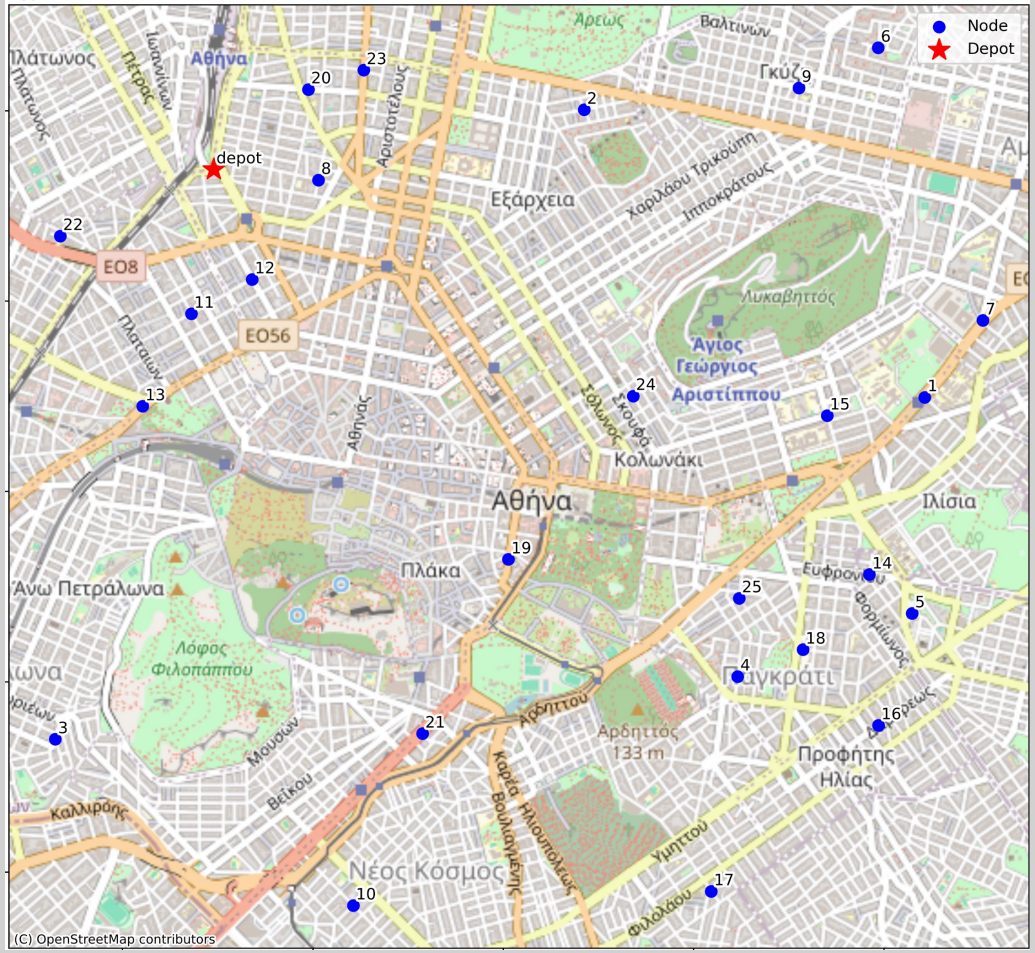
Modeling in CP: Traveling Salesman Problem

TSP in CP formulation:

$$allDifferent(x_i | i \in [1, \dots, 25, depot])$$
$$element(y, x_i) = y_i + 1 \qquad \forall i \neq depot$$

$x_i \in [1, \dots, 25, depot] \backslash i$

$y_i \in [0, \dots, 25]$: the position of the node in the route



Constraint Programming

Modeling in CP: Traveling Salesman Problem

TSP in CP formulation:

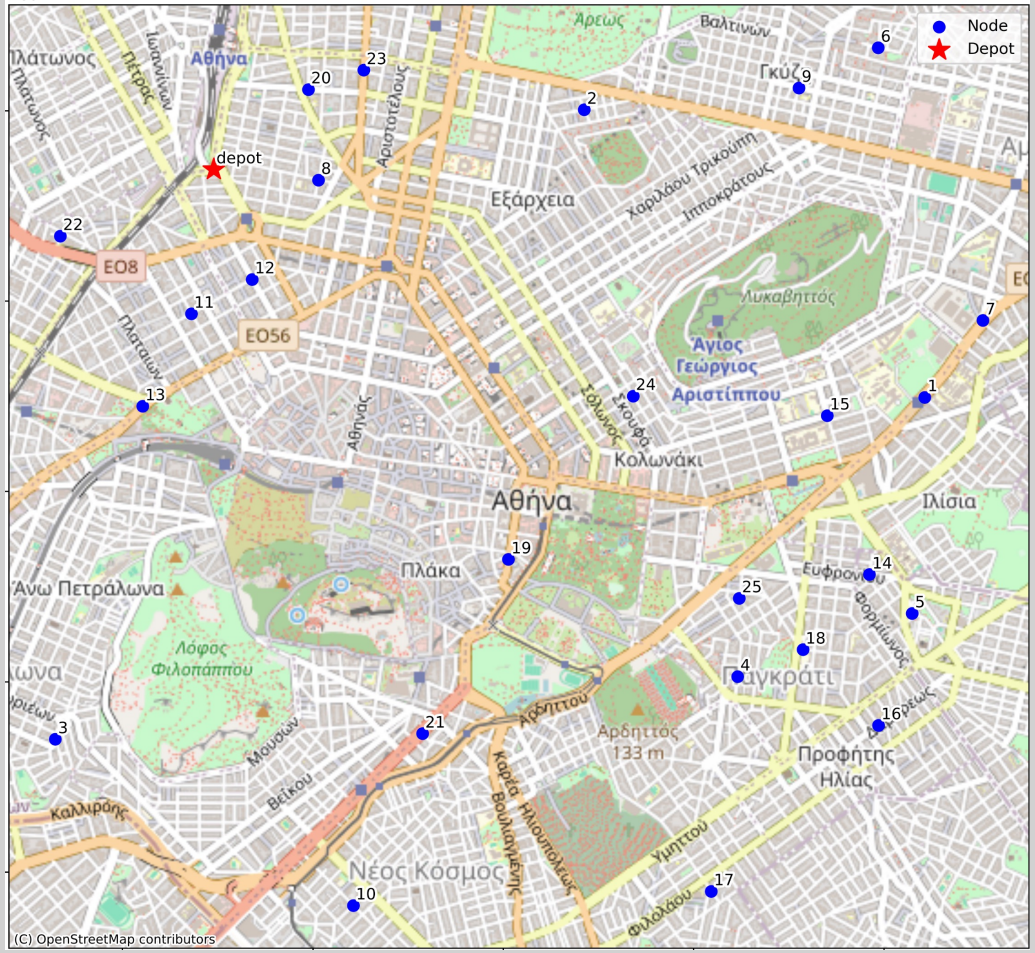
$$allDifferent(x_i | i \in [1, \dots, 25, depot])$$
$$element(y, x_i) = y_i + 1 \qquad \forall i \neq depot$$

$x_i \in [1, \dots, 25, depot] \backslash i$

$y_i \in [0, \dots, 25]$: the position of the node in the route

Objective function:

min $\sum_i element(d_i, x_i)$



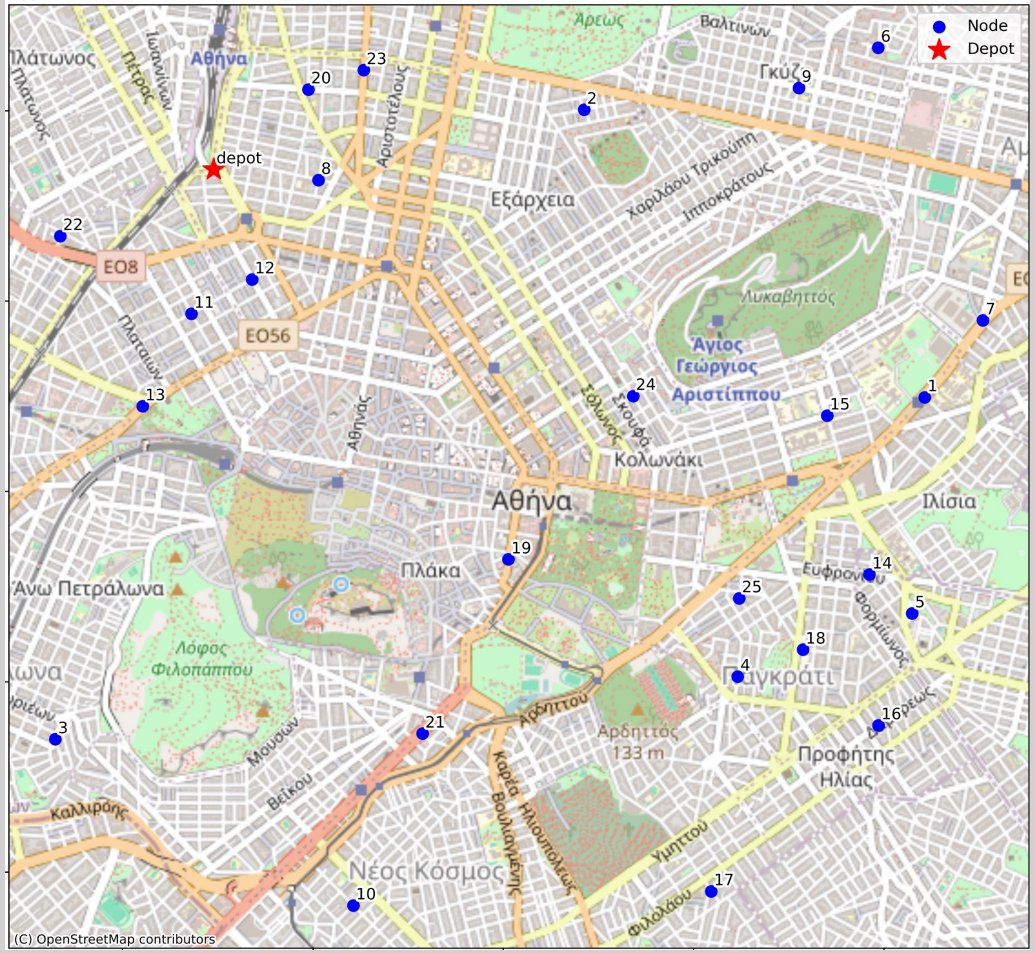
Constraint Programming

Modeling in CP: Traveling Salesman Problem

```
model = CpoModel()

x = [model.integer_var(0, len(nodes)-1, name = f'nextOf_{i}') for i in nodes]
y = [model.integer_var(0, len(nodes)-1, name = f'positionOf_{i}') for i in nodes]

model.add(model.all_diff(x))
for i in range(len(nodes)):
    if nodes[i] != 'depot':
        model.add(model.element(y, x[i]) == y[i]+1)

model.add(model.minimize(sum(model.element(distances[i], x[i]) for i in range(len(nodes)))))

sol = model.solve(TimeLimit = 10, trace_log = False)
```

More variants of TSP:

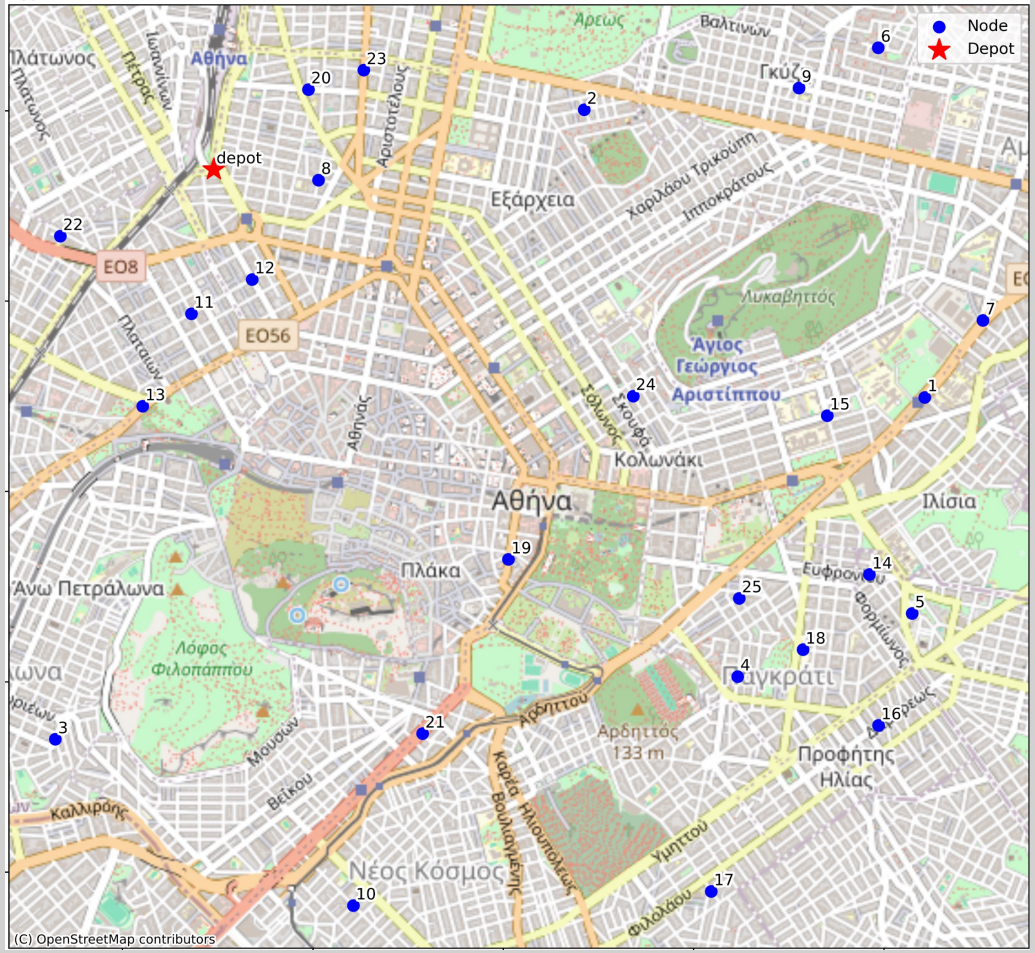
- Pickup and Delivery TSP
  - We can visit node 6 only after node 1 has been visited.
  - We can visit node 9 only after node 2 has been visited.
  - We can visit node 20 only after node 3 has been visited.



More variants of TSP:

- Pickup and Delivery TSP
  - We can visit node 6 only after node 1 has been visited.
  - We can visit node 9 only after node 2 has been visited.
  - We can visit node 20 only after node 3 has been visited.

$$y_1 < y_6$$
$$y_2 < y_9$$
$$y_3 < y_{20}$$



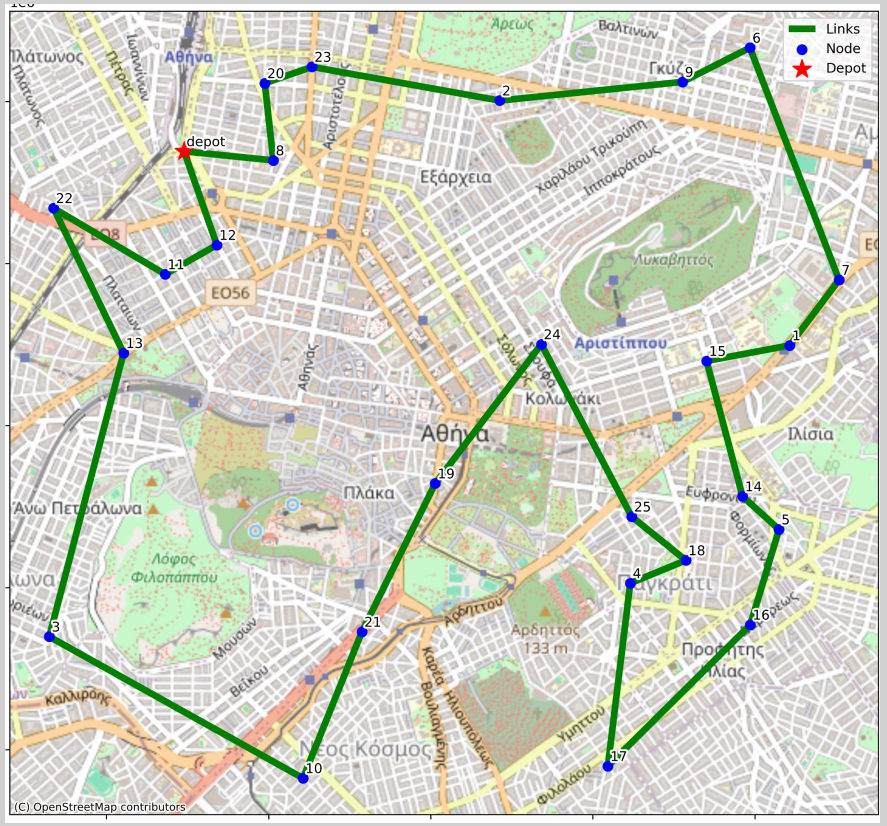
Constraint Programming

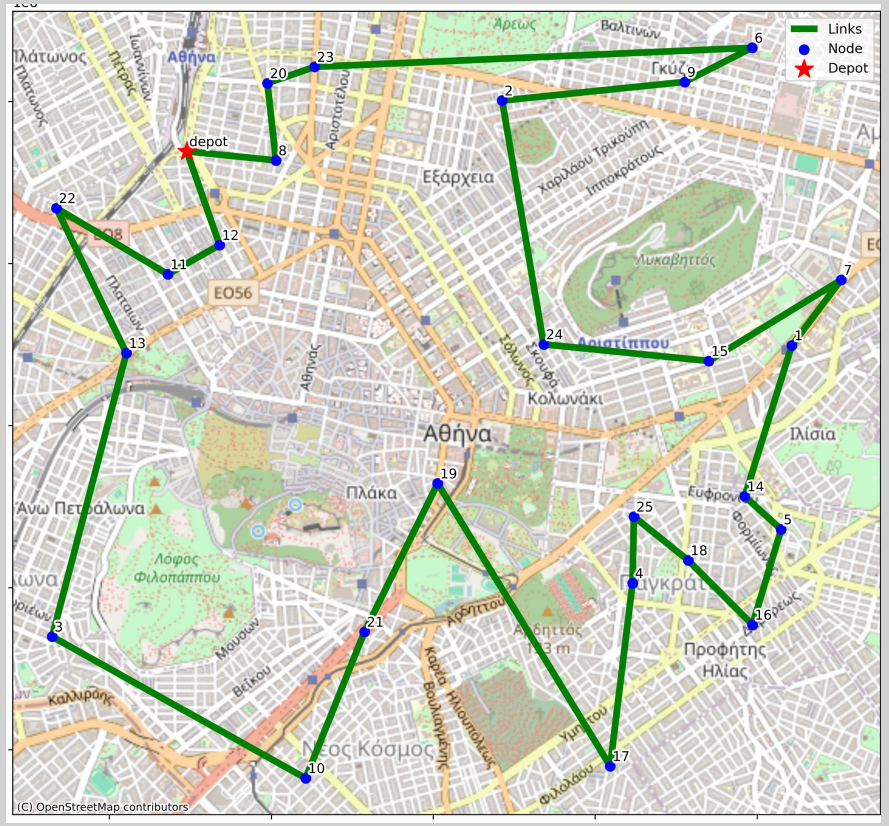
Modeling in CP: Traveling Salesman Problem

```
model = CpoModel()

x = [model.integer_var(0, len(nodes)-1, name = f'nextOf_{i}') for i in nodes]
y = [model.integer_var(0, len(nodes)-1, name = f'positionOf_{i}') for i in nodes]

model.add(model.all_diff(x))
for i in range(len(nodes)):
    if nodes[i] != 'depot':
        model.add(model.element(y, x[i]) == y[i]+1)
for i in range(len(pickups)):
    model.add(y[nodes.index(pickups[i])] < y[nodes.index(deliveries[i])])

model.add(model.minimize(sum(model.element(distances[i], x[i]) for i in range(len(nodes)))))

sol = model.solve(TimeLimit = 10, trace_log = False)
```

Constraint Programming

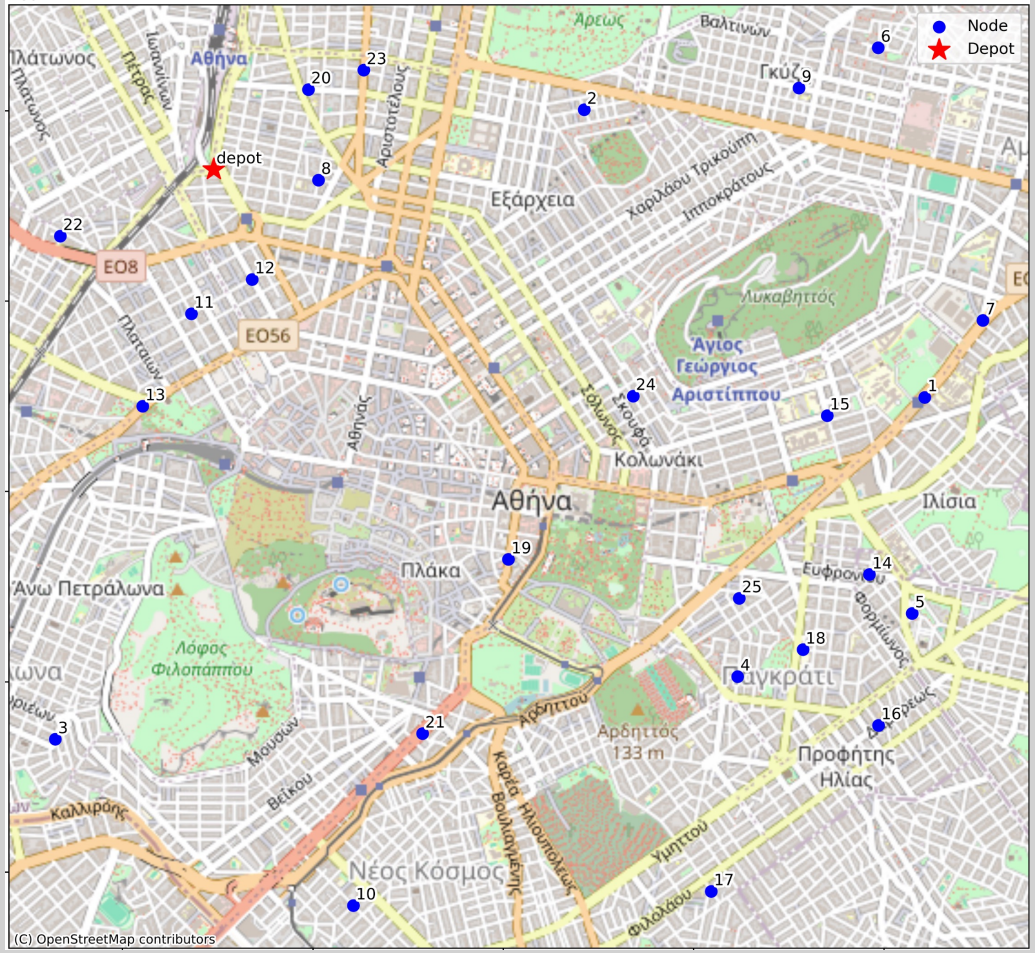
Modeling in CP: Traveling Salesman Problem





More variants of TSP:

- Capacity constraints
  - Node 1 is the pickup point of a parcel to be delivered to node 2.
  - Node 3 is the pickup point of a parcel to be delivered to node 4.
  - …
  - Node 13 is the pickup point of a parcel to be delivered to the depot.
  - No more than 2 parcels can fit in the vehicle.



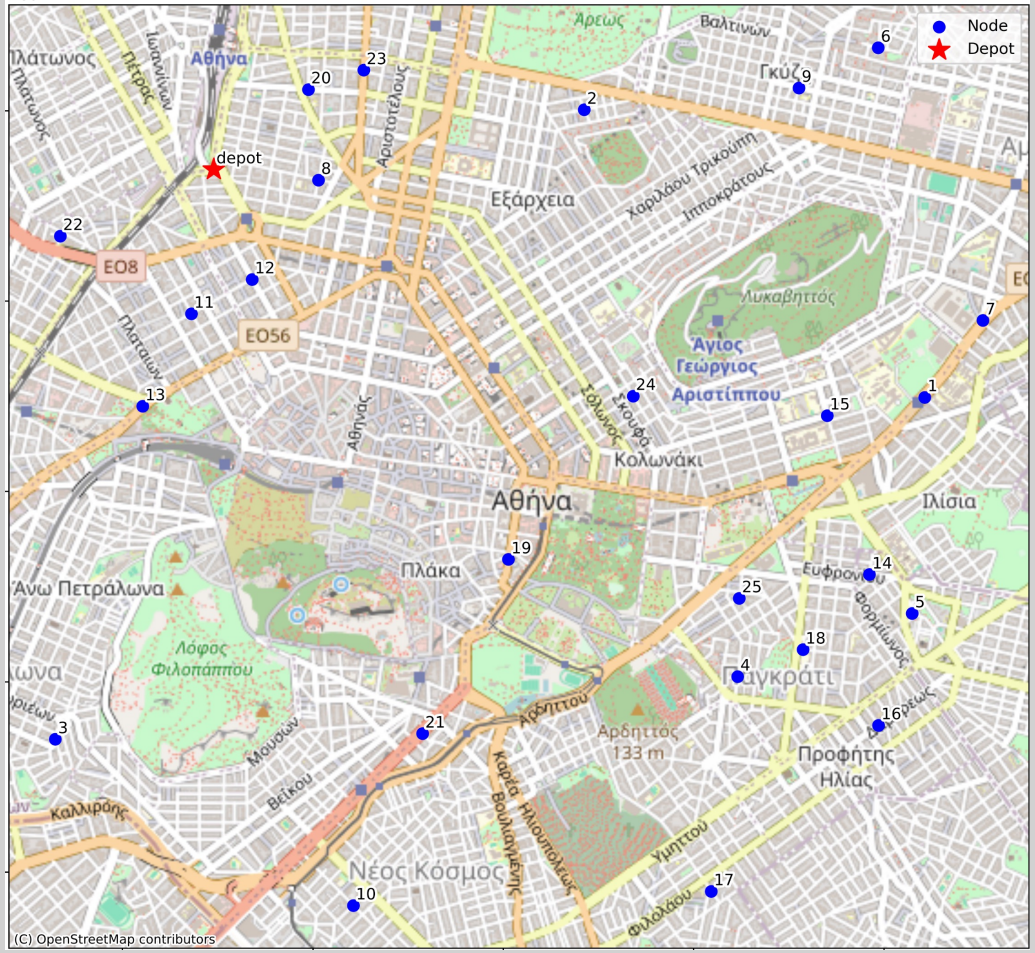
Constraint Programming

Modeling in CP: Traveling Salesman Problem

TSP in CP formulation:

$allDifferent(x_i | i \in [1, \dots, 25, depot])$
$element(y, x_i) = y_i + 1 \qquad \forall i \neq depot$

$x_i \in [1, \dots, 25, depot] \backslash i$
$y_i \in [0, \dots, 25]$: the position of the node in the route
$l_i \in \{0, 1, 2\}$: loaded parcels at node i



Constraint Programming

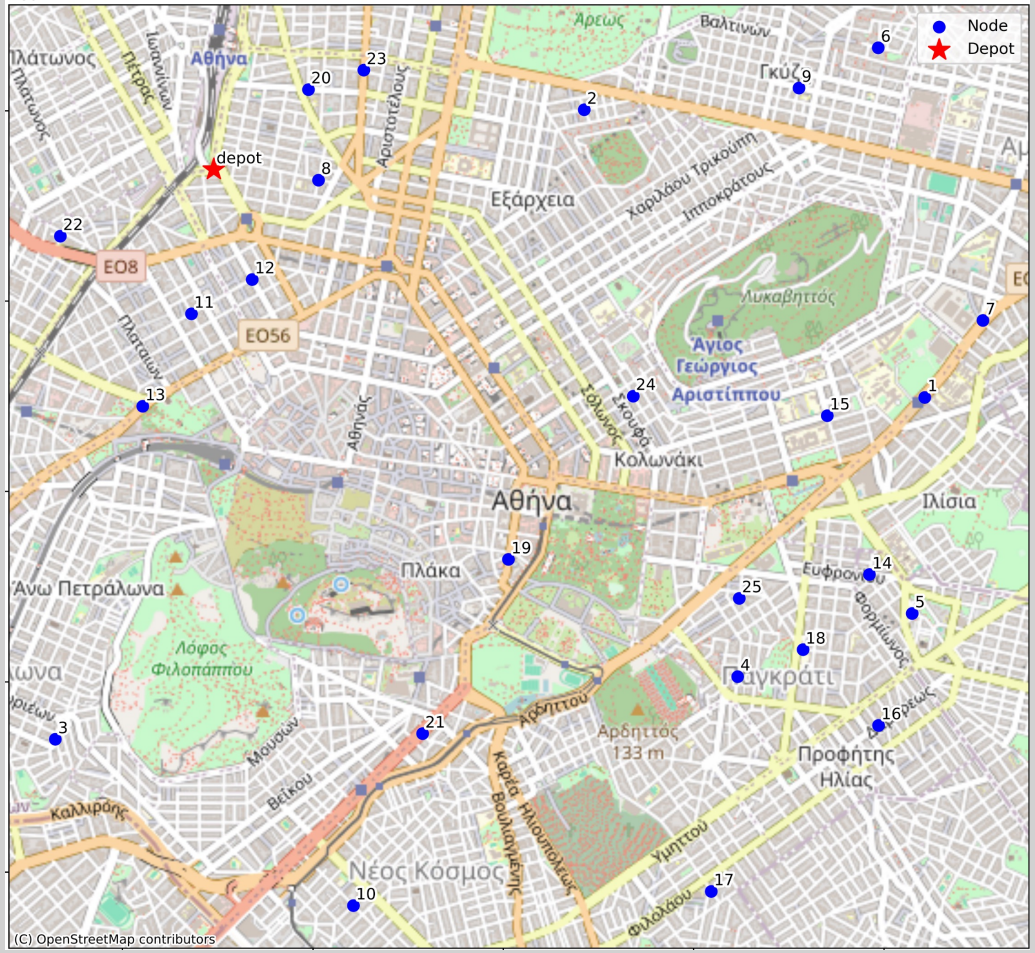
Modeling in CP: Traveling Salesman Problem

TSP in CP formulation:

$allDifferent(x_i | i \in [1, \dots, 25, depot])$
$element(y, x_i) = y_i + 1 \qquad \forall i \neq depot$

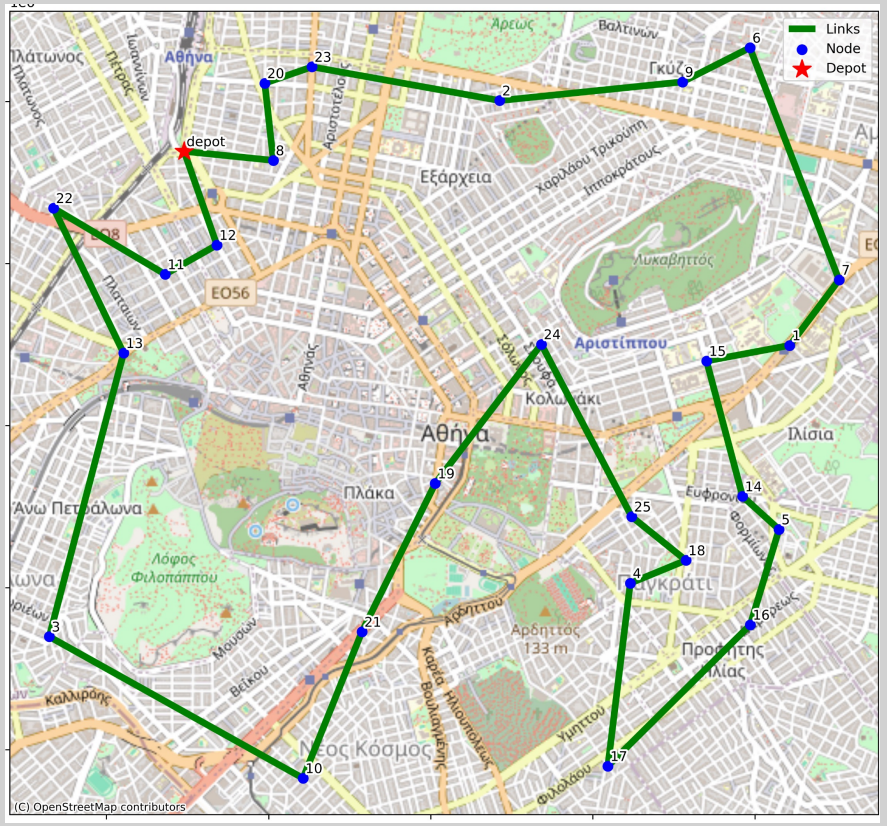
$x_i \in [1, \dots, 25, depot] \backslash i$
$y_i \in [0, \dots, 25]$: the position of the node in the route
$l_i \in \{0, 1, 2\}$: loaded parcels at node i

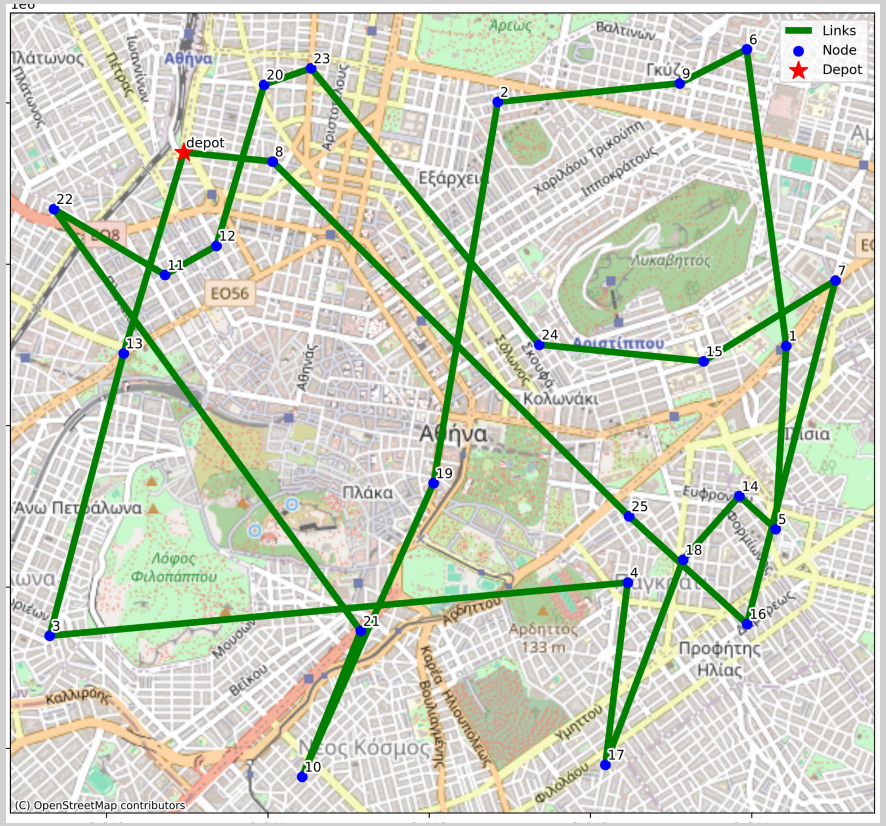
$l_{x_i} = l_i + w_i \qquad \forall i \neq depot$
$l_{depot} = 1$



Constraint Programming

Modeling in CP: Traveling Salesman Problem





Scheduling

Common variants/objectives

Annotations:

J: Jobs
M: Machines

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.

Annotations:

J: Jobs
M: Machines

- Single-stage jobs – each job is processed in a single phase
  e.g., exam scheduling – each course is assigned exactly once
- Multi-stage jobs – each job requires multiple phases of processing
  e.g., manufacturing a product usually requires the construction of multiple compartments, assembled at the end
  The compartments may impose a designated order of construction or machine eligibilities

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.

Annotations:

J: Jobs
M: Machines
p_{jm}: Processing time of job j on machine m

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.
Each job is processed for a fixed time interval called **processing time**. As long as a job is processed, then the assigned machine is unavailable (or a part of its resources is unavailable).

Annotations:

J: Jobs
M: Machines
p_{jm}: Processing time of job j on machine m

- Identical machines – all processing times are the same (p_j instead of p_{jm})
- Uniform machines – each machine has a speed parameter
- Unrelated machines – each job has different processing times per machines in an unrelated manner

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.
Each job is processed for a fixed time interval called **processing time**. As long as a job is processed, then the assigned machine is unavailable (or a part of its resources is unavailable).

Annotations:

J: Jobs
M: Machines
p_{jm}: Processing time of job j on machine m
s_{ijm}: Setup times

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.
Each job is processed for a fixed time interval called **processing time**.
Setups are tasks which take place right before the start of processing. The duration of these tasks are called **setup times**.

Annotations:

J: Jobs
M: Machines
p_{jm}: Processing time of job j on machine m
s_{ijm}: Setup times

- Sequence-dependent setup times: the duration is depended on the previous job which has been processed - s_{ijm} is the setup time of job j on machine m if the previous job was i.

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.
Each job is processed for a fixed time interval called **processing time**.
Setups are tasks which take place right before the start of processing. The duration of these tasks are called **setup times**.

Annotations:

J: Jobs
M: Machines
p_{jm}: Processing time of job j on machine m
s_{ijm}: Setup times
d_j: due-time of job j
r_j: release time of job j

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.
Each job is processed for a fixed time interval called **processing time**.
Setups are tasks which take place right before the start of processing. The duration of these tasks are called **setup times**. **Release** or **due** times may restrict the solution.

Annotations:

J: Jobs
M: Machines
p_{jm}: Processing time of job j on machine m
s_{ijm}: Setup times

- Due-times impose either **strict** (the job must be completed before its due-time) or **soft** (the job could be completed after its due-time, but a penalty is charged) constraints.

d_j: due-time of job j
r_j: release time of job j

Description:

Set J contains tasks which should can be carried out by (some of the) machines M.
Each job is processed for a fixed time interval called **processing time**.
Setups are tasks which take place right before the start of processing. The duration of these tasks are called **setup times**. **Release** or **due** times may restrict the solution.

Common objectives:

Total completion times: $\min \sum_{j \in J} C_j$, C_j: completion time of job j

All jobs must be scheduled in the minimum sum of completion times.

Throughput: $\max \sum_{j \in J} \sum_{m \in M} x_{jm}$, $x_{jm} = 1$ if job j is assigned to m

Not all jobs can be scheduled – maximise the number of scheduled jobs

Makespan: $\min C_{\max}$, $C_{\max} \geq C_j \; \forall j \in J$

All jobs must be scheduled in the minimum makespan (when the last job is completed).

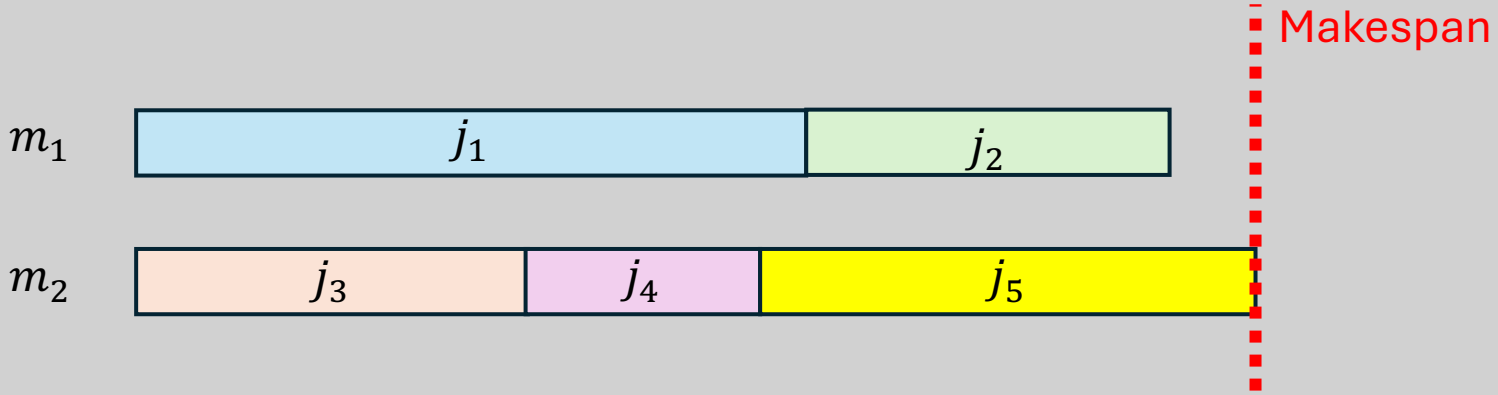
(Weighted) Tardiness: $\min \sum_{j \in J} (w_j \cdot) T_j$, $T_j \geq C_j - d_j \; \forall j \in J$

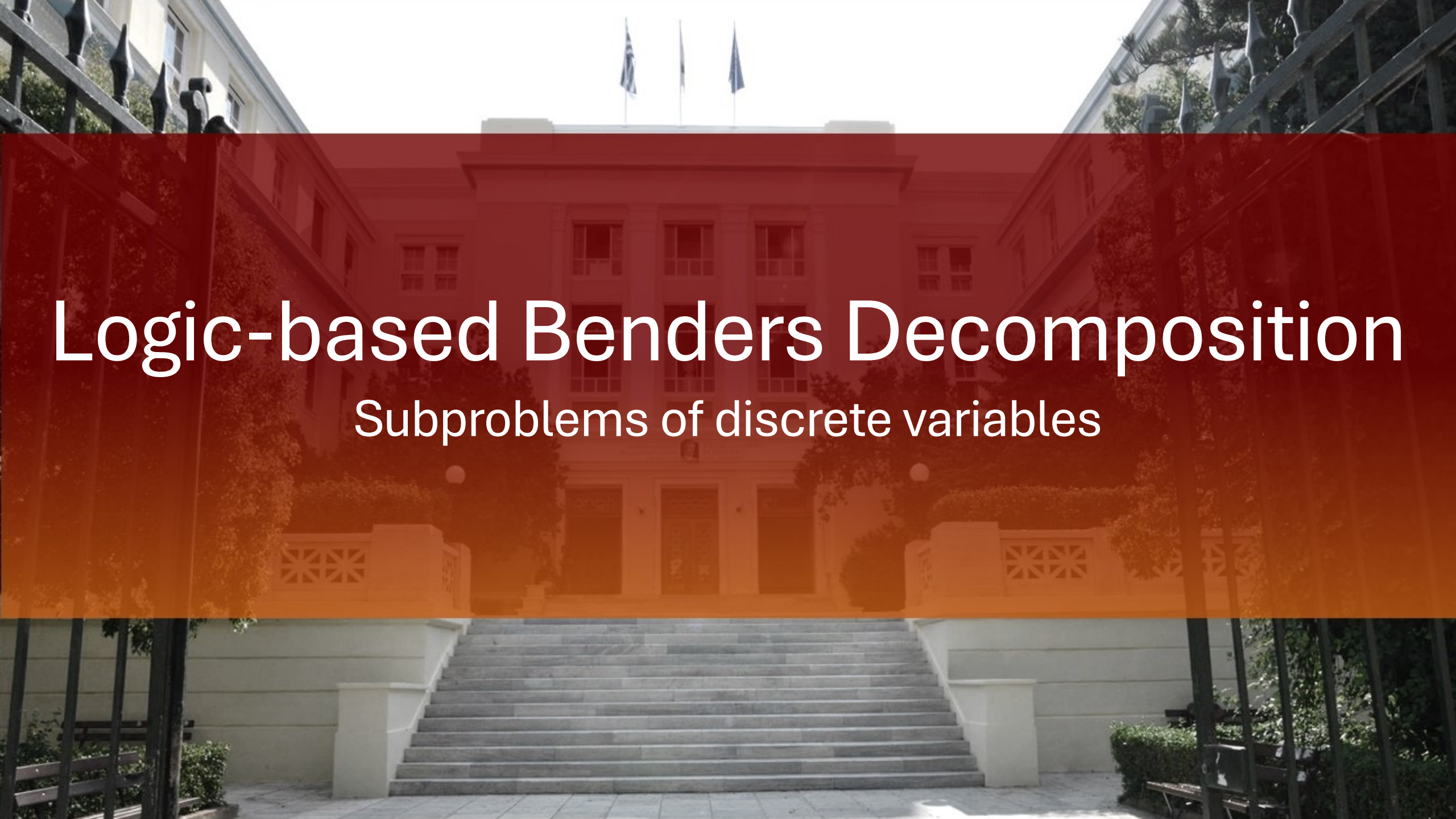
All jobs must be scheduled, but not all due-times can be respected – we minimise the delay penalties.

The simplest variant:

Single-stage jobs J of processing times p_j must be assigned to identical parallel machines M. Each machine can process one job at the time. We aim at minimising **makespan** (maximum completion times).

E.g., for 5 jobs on 2 machines:





Logic-based Benders Decomposition

Subproblems of discrete variables

Logic-Based Benders Decomposition

Beyond dual-derived cuts

In classical Benders Decomposition, the subproblem should consist of strictly **continuous variables**, to generate dual-derived cuts.

This may be too restrictive: most practical problems require a partitioning which brings forward subproblems of integer variables.

Logic-Based Benders Decomposition

Beyond dual-derived cuts

In classical Benders Decomposition, the subproblem should consist of strictly **continuous variables**, to generate dual-derived cuts.

This may be too restrictive: most practical problems require a partitioning which brings forward subproblems of integer variables.

In 2000, J.F.Hooker extended the method, so that subproblems of integer variables are implemented. The extension is named **Logic-Based Benders Decomposition**. The method is efficient, as:

- Any complicated MILP can be partitioned into more easily solved subproblems.
- Each subproblem can be consisted of any variable and solved by any method (e.g., the subproblem can be solved by an exact algorithm or a Constraint Programming model, not strictly as a MILP).

Logic-Based Benders Decomposition

Beyond dual-derived cuts

In classical Benders Decomposition, the subproblem should consist of strictly **continuous variables**, to generate dual-derived cuts.

This may be too restrictive: most practical problems require a partitioning which brings forward subproblems of integer variables.

In 2000, J.F.Hooker extended the method, so that subproblems of integer variables are implemented. The extension is named **Logic-Based Benders Decomposition**. The method is efficient, as:

- Any complicated MILP can be partitioned into more easily solved subproblems.
- Each subproblem can be consisted of any variable and solved by any method (e.g., the subproblem can be solved by an exact algorithm or a Constraint Programming model, not strictly as a MILP).

In LBBD, **combinatorial cuts** are used to eliminate previous solutions.

Logic-Based Benders Decomposition

Example: Scheduling

J: Jobs {1, 2, 3}

M: Machines {1, 2}

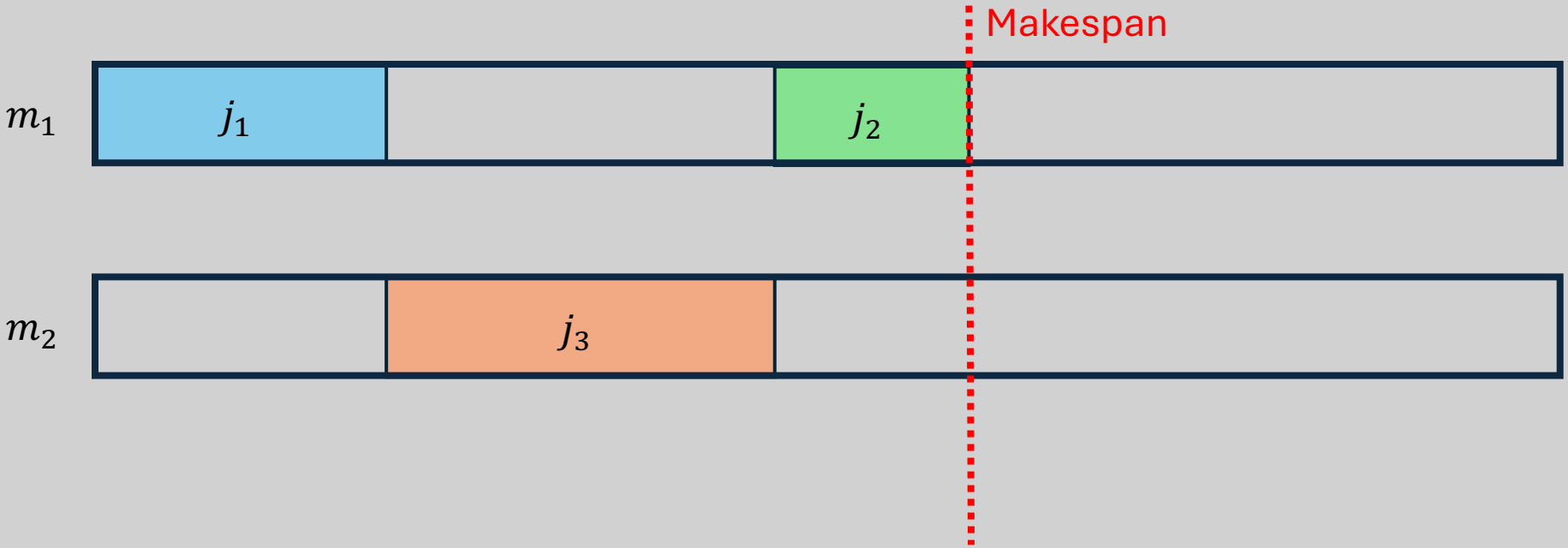
p_{jm}: Processing time of job j on machine m *1:* {3, 2, 5}, *2*: {1, 5, 4}

All jobs must be assigned to one machine. Only one job can be processed at the same time over all machines.

Objective function: minimization of makespan

Logic-Based Benders Decomposition

Example: Scheduling



Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{P}$:

$$\min \quad C_{max}$$

$$\begin{aligned}
&\textstyle\sum_{m \in M} x_{jmt} = 1 && \forall j \in J \\
&\textstyle\sum_{t'=t}^{t+p_{jm}} y_{jmt'} \geq p_{jm} \cdot x_{jmt} && \forall j \in J, m \in M, t \\
&\textstyle\sum_{j \in J} \sum_{m \in M} y_{jmt} \leq 1 && \forall t \\
&C_{max} \geq t \cdot y_{jmt} && \forall j \in J, m \in M, t
\end{aligned}$$

$$\begin{aligned}
&x_{jmt} \in \{0,1\} && \forall j \in J, m \in M, t \\
&y_{jmt} \in \{0,1\} && \forall j \in J, m \in M, t \\
&C_{max} \geq 0
\end{aligned}$$

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{P}$:

$$\min \quad C_{max}$$

$$\sum_{m \in M} x_{jmt} = 1 \qquad \forall j \in J$$

$$\sum_{t'=t}^{t+p_{jm}} y_{jmt'} \geq p_{jm} \cdot x_{jmt} \qquad \forall j \in J, m \in M, t$$

$$\sum_{j \in J} \sum_{m \in M} y_{jmt} \leq 1 \qquad \forall t$$

$$C_{max} \geq t \cdot y_{jmt} \qquad \forall j \in J, m \in M, t$$

$$x_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

$$y_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

$$C_{max} \geq 0$$

Set to 1 if job *j* start at time *t* on machine *m*.

Set to 1 if job *j* is processed at time *t* on machine *m*.

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{P}$:

$$min \quad C_{max}$$

$\sum_{m \in M} x_{jmt} = 1$	$\forall j \in J$	Each job is assigned to one machine.
$\sum_{t'=t}^{t+p_{jm}} y_{jmt'} \geq p_{jm} \cdot x_{jmt}$	$\forall j \in J, m \in M, t$	
$\sum_{j \in J} \sum_{m \in M} y_{jmt} \leq 1$	$\forall t$	
$C_{max} \geq t \cdot y_{jmt}$	$\forall j \in J, m \in M, t$	
$x_{jmt} \in \{0,1\}$	$\forall j \in J, m \in M, t$	Set to 1 if job *j* start at time *t* on machine *m*.
$y_{jmt} \in \{0,1\}$	$\forall j \in J, m \in M, t$	Set to 1 if job *j* is processed at time *t* on machine *m*.
$C_{max} \geq 0$		

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{P}$:

$$min \quad C_{max}$$

$$\sum_{m \in M} x_{jmt} = 1 \qquad \forall j \in J$$

$$\sum_{t'=t}^{t+p_{jm}} y_{jmt'} \geq p_{jm} \cdot x_{jmt} \qquad \forall j \in J, m \in M, t$$

Each job which starts at time t is processed at all time instances from t to $t + p_{jm}$.

$$\sum_{j \in J} \sum_{m \in M} y_{jmt} \leq 1 \qquad \forall t$$

$$C_{max} \geq t \cdot y_{jmt} \qquad \forall j \in J, m \in M, t$$

$$x_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

Set to 1 if job j start at time t on machine m.

$$y_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

Set to 1 if job j is processed at time t on machine m.

$$C_{max} \geq 0$$

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{P}$:

$$min \quad C_{max}$$

$$\sum_{m \in M} x_{jmt} = 1 \qquad \forall j \in J$$

$$\sum_{t'=t}^{t+p_{jm}} y_{jmt'} \geq p_{jm} \cdot x_{jmt} \qquad \forall j \in J, m \in M, t$$

$$\sum_{j \in J} \sum_{m \in M} y_{jmt} \leq 1 \qquad \forall t$$

$$C_{max} \geq t \cdot y_{jmt} \qquad \forall j \in J, m \in M, t$$

Each time instance can be assigned to no more than one job.

$$x_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

Set to 1 if job *j* start at time *t* on machine *m*.

$$y_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

Set to 1 if job *j* is processed at time *t* on machine *m*.

$$C_{max} \geq 0$$

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{P}$:

$$min \quad C_{max}$$

$$\sum_{m \in M} x_{jmt} = 1 \qquad \forall j \in J$$

$$\sum_{t'=t}^{t+p_{jm}} y_{jmt'} \geq p_{jm} \cdot x_{jmt} \qquad \forall j \in J, m \in M, t$$

$$\sum_{j \in J} \sum_{m \in M} y_{jmt} \leq 1 \qquad \forall t$$

$$C_{max} \geq t \cdot y_{jmt} \qquad \forall j \in J, m \in M, t$$

Makespan is set to the maximum completion time.

$$x_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

Set to 1 if job *j* start at time *t* on machine *m*.

$$y_{jmt} \in \{0,1\} \qquad \forall j \in J, m \in M, t$$

Set to 1 if job *j* is processed at time *t* on machine *m*.

$$C_{max} \geq 0$$

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{M}$:

$$min \quad C_{max}$$

$$\sum_{m \in M} x_{jm} = 1 \qquad \forall j \in J$$
$$C_{max} \geq \sum_{j \in J} p_{jm} \cdot x_{jm} \qquad \forall m \in M$$

$$x_{jm} \in \{0, 1\} \qquad \forall j \in J, m \in M$$
$$C_{max} \geq 0$$

Logic-Based Benders Decomposition

Example: Scheduling

MILP formulation for $\mathcal{M}$:

$$min \quad C_{max}$$

$$\sum_{m \in M} x_{jm} = 1 \qquad \forall j \in J$$

$$C_{max} \geq \sum_{j \in J} p_{jm} \cdot x_{jm} \qquad \forall m \in M$$

$$x_{jm} \in \{0,1\} \qquad \forall j \in J, m \in M$$

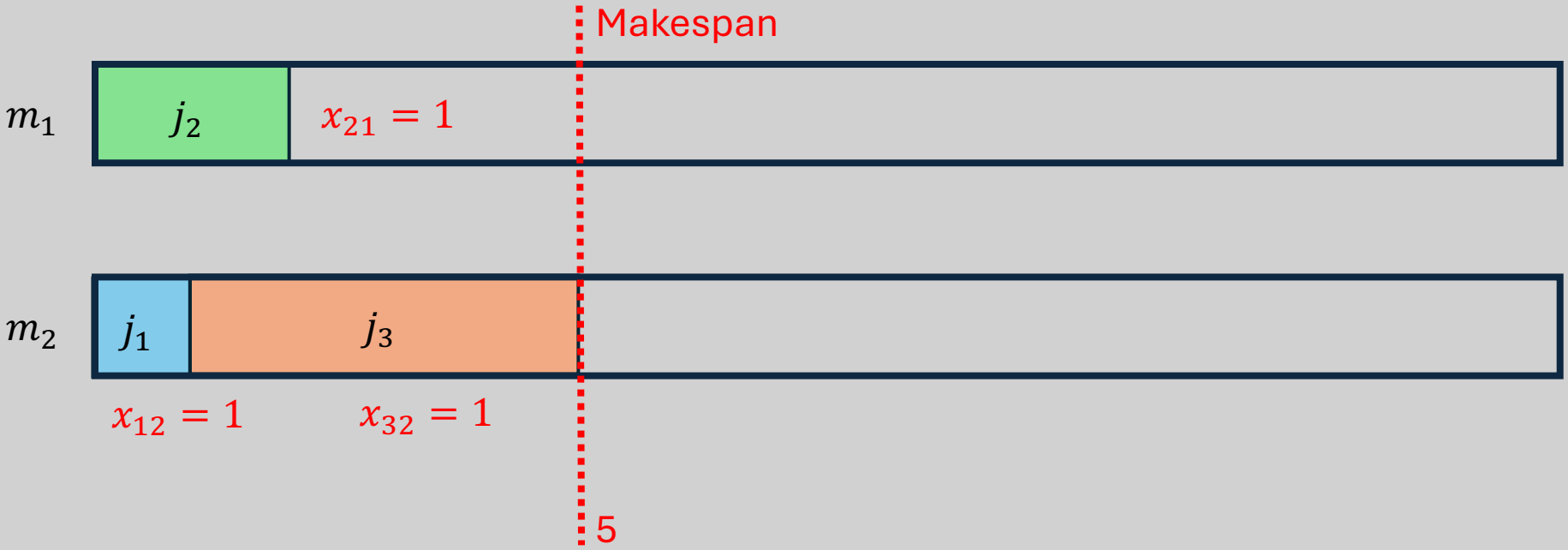
$$C_{max} \geq 0$$

A **relaxation** of $\mathcal{P}$:

- Each job is assigned to exactly one machine.
- The constraints which enforce the processing of no more than one job at the same time are removed.

Logic-Based Benders Decomposition

Example: Scheduling



Logic-Based Benders Decomposition

Example: Scheduling

CP formulation for $\mathcal{S}$:

Given the assignments of jobs to machines, $\bar{p}_j = p_{jm}$ if $x_{jm} = 1$.

$$min \quad C_{max}$$

$$Cumulative(s_j, \bar{p}_j, 1, 1 | j \in J)$$

$$C_{max} \geq s_j + \bar{p}_j \qquad \forall j \in J$$

$$s_j \geq 0 \; integer \qquad \forall j \in J$$

$$C_{max} \geq 0$$

Logic-Based Benders Decomposition

Example: Scheduling

Logic-Based Benders Decomposition

Example: Scheduling

Iteration 1:

- After solving $\mathcal{M} \rightarrow x_{21} = 1, x_{12} = 1, x_{32} = 1$, LB = 5.
- Given the solution of $\mathcal{M}$, we solve $\mathcal{S} \rightarrow$ UB = 7.
- Since LB < UB, the optimal solution is not found yet → a cut is added:

$$C_{max} \geq UB - UB \cdot \left(\sum_{(j,m):\bar{x}_{jm}=1} (1 - x_{jm})\right)$$

Logic-Based Benders Decomposition

Example: Scheduling

Iteration 1:

- After solving $\mathcal{M} \rightarrow x_{21} = 1, x_{12} = 1, x_{32} = 1$, LB = 5.
- Given the solution of $\mathcal{M}$, we solve $\mathcal{S} \rightarrow$ UB = 7.
- Since LB < UB, the optimal solution is not found yet → a cut is added:

$$C_{max} \geq UB - UB \cdot \left(\sum_{(j,m):\bar{x}_{jm}=1}(1 - x_{jm})\right)$$

If all jobs are assigned to the same machine (i.e., $\sum_{(j,m):\bar{x}_{jm}=1}(1 - x_{jm}) = 0$), then the value of makespan is set to the **upper bound**.

Logic-Based Benders Decomposition

Example: Scheduling

Iteration 1:

- After solving $\mathcal{M} \rightarrow x_{21} = 1, x_{12} = 1, x_{32} = 1$, LB = 5.
- Given the solution of $\mathcal{M}$, we solve $\mathcal{S} \rightarrow$ UB = 7 – the best found.
- Since LB < UB, the optimal solution is not found yet → a cut is added:

$$C_{max} \geq 7 - 7 \cdot ((1 - x_{21}) + (1 - x_{12}) + (1 - x_{32}))$$

If all jobs are assigned to the same machine (i.e., $\sum_{(j,m):\bar{x}_{jm}=1}(1 - x_{jm}) = 0$), then the value of makespan is set to the **upper bound**.

Logic-Based Benders Decomposition

Example: Scheduling

Iteration 2:

- After solving $\mathcal{M} \rightarrow x_{11} = 1, x_{21} = 1, x_{32} = 1$, LB = 5.
- Given the solution of $\mathcal{M}$, we solve $\mathcal{S} \rightarrow$ UB = 9 – not improving the best found (7).
- Since LB < UB, the optimal solution is not found yet → a cut is added:

$$C_{max} \geq 9 - 9 \cdot ((1 - x_{11}) + (1 - x_{21}) + (1 - x_{32}))$$

Logic-Based Benders Decomposition

Example: Scheduling

Iteration 3:

- After solving $\mathcal{M} \rightarrow x_{12} = 1, x_{22} = 1, x_{31} = 1$, LB = 6.
- Given the solution of $\mathcal{M}$, we solve $\mathcal{S} \rightarrow$ UB = 11 – not improving the best found (7).
- Since LB < UB, the optimal solution is not found yet → a cut is added:

$$C_{max} \geq 11 - 11 \cdot ((1 - x_{12}) + (1 - x_{22}) + (1 - x_{31}))$$

Logic-Based Benders Decomposition

Example: Scheduling

Iteration 4:

- After solving $\mathcal{M} \rightarrow x_{12} = 1, x_{21} = 1, x_{31} = 1$, LB = 7.
- We notice that LB = best found UB (7).
- Since LB = UB, the optimal solution has been found, and the algorithm terminates.

Logic-Based Benders Decomposition

Example: Scheduling

```
master = masterProblem(jobs, machines, processingTimes) # Construct the master problem 'M'
lowerBound, upperBound, iteration = 0, inf, 1 # Initialize lower bound, upper bound, number of iterations
while(lowerBound < upperBound): # The procedure is repeated until a convergence is reached.
    results = opt.solve(master, tee = False) # Solve 'M'
    x = {}
    for j in master.Jobs:
        for m in master.Machines:
            if value(master.x[j, m]) > 0.9:
                x[j] = m
    lowerBound = value(master.obj) # Update the value of the lower bound
    startTimes, ub = subProblem(jobs, [processingTimes[j][x[j]] for j in master.Jobs]) # Construct the subproblem 'S'
    if ub < upperBound: # Update the value of the upper bound, in case of improvement
        upperBound = ub
    master.constraints.add(master.Cmax >= ub - ub*(sum((1 - master.x[j, x[j]]) for j in master.Jobs))) # Add cut to 'M'
    print(f"{iteration:<12}{lowerBound:<16}{upperBound:<16}{round(100*(upperBound - lowerBound)/upperBound, 2)}%")
    plotRoute(iteration, jobs, machines, startTimes, processingTimes, x)
    iteration += 1 # Increase the number of iterations by 1
```

Industrial case

Scheduling in Textile industry

Industrial case

Problem description

Scheduling in Textile industry:

Sets

- A set of orders J to be scheduled
- A set of parallel machines (weaving looms) M

Parameters

- Each job j has a machine-dependent processing time p_{jm}
- Before processing, each job j must be set up in the weaving loom m – the duration of the setup operation s_{ijm} is sequence-and-machine-dependent.
- Setup operations occupy a working group – only R groups are available.

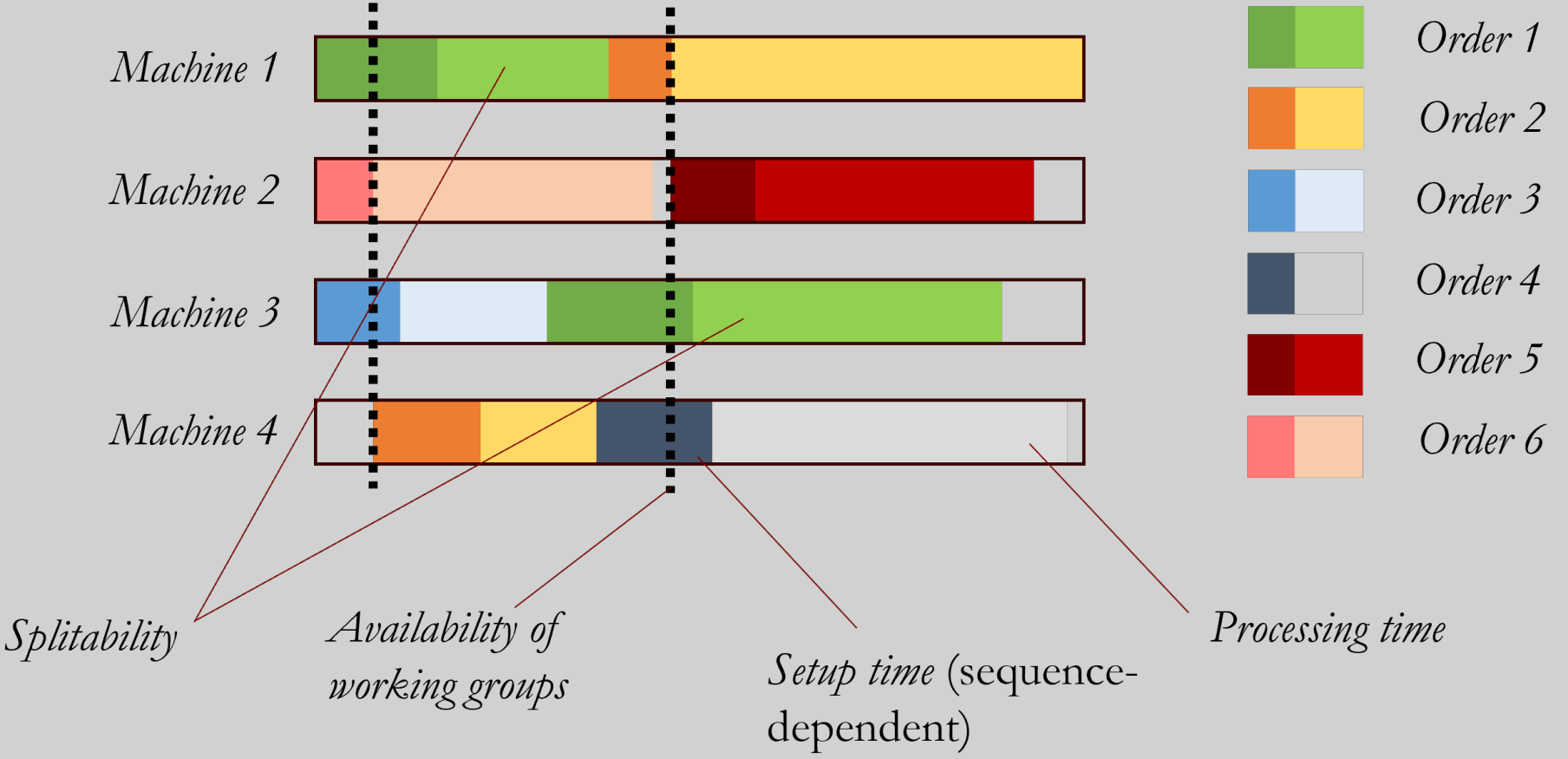
Constraints

- Jobs can be split – parts of the same job can be processed on different machines
- All jobs must be scheduled.
- No more than 1 job can be set up or processed on each machine.
- No more than R setup operations can take place in parallel.

Industrial case

Problem description

Scheduling in Textile industry:



Industrial case

Problem description

Scheduling in Textile industry:

Following the same modeling approach, we should has a set of variables:

$x_{ijmt} \in \{0,1\} \qquad \forall i \in J, j \in J \backslash \{i\}, m \in M, t \in T$

set to 1 if job j is processed at time instance t on machine m, succeeding job i.

For a moderate scale (~100 jobs), a set of 5 looms and for a daily planning horizon (i.e., 1440 minutes):

$|J| \times |J| \times |M| \times |T| = 100 \times 100 \times 5 \times 1440 = 72.000.000$ variables

This scale cannot be handled – a MILP would not provide any feasible solution in reasonable time.

Industrial case

Problem description

Scheduling in Textile industry:

Instead, a relaxation which considers the same setting **without the resource constraints** would be a simpler problem, consisted of variables:

$x_{ijm} \in \{0,1\} \qquad \forall i \in J, j \in J \backslash \{i\}, m \in M,$

set to 1 if job j is processed on machine m, succeeding job i.

For a moderate scale (~100 jobs), a set of 5 looms:

$|J| \times |J| \times |M| = 100 \times 100 \times 5 = 50.000$ variables

An easily handled scale for modern solvers.

Industrial case

Master problem

For the master problem $\mathcal{M}$:

Sets

- A set of orders J to be scheduled
- A set of parallel machines (weaving looms) M

Parameters

- Each job j has a machine-dependent processing time p_{jm}
- Before processing, each job j must be set up in the weaving loom m – the duration of the setup operation s_{ijm} is sequence-and-machine-dependent.
- ~~Setup operations occupy a working group – only R groups are available.~~

Constraints

- Jobs can be split – parts of the same job can be processed on different machines
- All jobs must be scheduled.
- No more than 1 job can be set up or processed on each machine.
- ~~No more than R setup operations can take place in parallel.~~

For the master problem $\mathcal{M}$:

$\min z$

subject to:

Assignment of jobs to machines

$$\sum_{m \in M} W_{i,m}^{k-1} = 100 \qquad \forall i \in J^*$$

$$100 \cdot y_{i,m}^{k-1} \geq W_{i,m}^{k-1} \qquad \forall i \in J^*, m \in M$$

Sequencing of jobs

$$y_{i,m}^{k-1} = \sum_{j \in J, j \neq i} x_{i,j,m}^{k-1} \qquad \forall i \in J^*, m \in M$$

$$y_{i,m}^{k-1} = \sum_{j \in J, j \neq i} x_{j,i,m}^{k-1} \qquad \forall i \in J^*, m \in M$$

$$\sum_{j \in J} x_{0,j,m}^{k-1} \leq 1 \qquad \forall m \in M$$

$$n_{i,m}^{k-1} - n_{j,m}^{k-1} + |J| \cdot x_{i,j,m}^{k-1} \leq |J| - 1 \qquad \forall i, j \in J^*, m \in M$$

$$n_{i,m}^{k-1} \leq |J| - 1 \qquad \forall i \in J^*, m \in M$$

Objective function

$$z \geq \sum_{j \in J^*} \frac{p_{j,m}}{100} \cdot W_{j,m}^{k-1} + \sum_{i \in J} \sum_{j \in J^*} s_{i,j,m} \cdot x_{i,j,m}^{k-1} \qquad \forall m \in M$$

Variables

$$x_{i,j,m}^{k-1} \in \{0,1\} \qquad \forall j \in J, i \in J \setminus \{j\}, m \in M$$

$$y_{i,m}^{k-1} \in \{0,1\} \qquad \forall i \in J, m \in M$$

$$n_{i,m}^{k-1}, W_{i,m}^{k-1} \in \mathbb{Z}_+ \qquad \forall i \in J, m \in M$$

Industrial case

Master problem

For the master problem $\mathcal{M}$:

Assignment of jobs to machines

$$\sum_{m \in M} W_{i,m}^{k-1} = 100 \qquad \forall i \in J^*$$

$$100 \cdot y_{i,m}^{k-1} \geq W_{i,m}^{k-1} \qquad \forall i \in J^*, m \in M$$

$W_{i,m}$: Percentage of job $i \in J$ assigned to machine $m \in M$, integer

$y_{i,m}$: Binary variable; set to 1 if any part of job $i \in J$ is assigned to machine $m \in M$

- 100% of each job must be assigned to the machines.
- If $W_{i,m}$ is greater than 0, then $y_{i,m}$ is set to 1.

Industrial case

Master problem

For the master problem $\mathcal{M}$:

Sequencing of jobs

$$y_{i,m}^{k-1} = \sum_{j \in J, j \neq i} x_{i,j,m}^{k-1} \qquad \forall i \in J^*, m \in M$$

$$y_{i,m}^{k-1} = \sum_{j \in J, j \neq i} x_{j,i,m}^{k-1} \qquad \forall i \in J^*, m \in M$$

$$\sum_{j \in J} x_{0,j,m}^{k-1} \leq 1 \qquad \forall m \in M$$

$$n_{i,m}^{k-1} - n_{j,m}^{k-1} + |J| \cdot x_{i,j,m}^{k-1} \leq |J| - 1 \qquad \forall i, j \in J^*, m \in M$$

$$n_{i,m}^{k-1} \leq |J| - 1 \qquad \forall i \in J^*, m \in M$$

$x_{i,j,m}$: Binary variable; set to 1 if j succeeds i on machine m

$n_{i,m}$: Order of processing of job i on machine m

- If any part of i is assigned to m, it will succeed and precede of exactly one job.
- An imaginary job 0 is defined as a starting point.
- Subtour elimination constraints – each machine has a sequence of jobs; the properties are similar with the case of TSP.

Industrial case

Master problem

For the master problem $\mathcal{M}$:

Objective function

$$z \geq \sum_{j \in J^*} \frac{p_{j,m}}{100} \cdot W_{j,m}^{k-1} + \sum_{i \in J} \sum_{j \in J^*} s_{i,j,m} \cdot x_{i,j,m}^{k-1} \qquad \forall m \in M$$

Makespan objective: the maximum completion time of the schedule of the machines

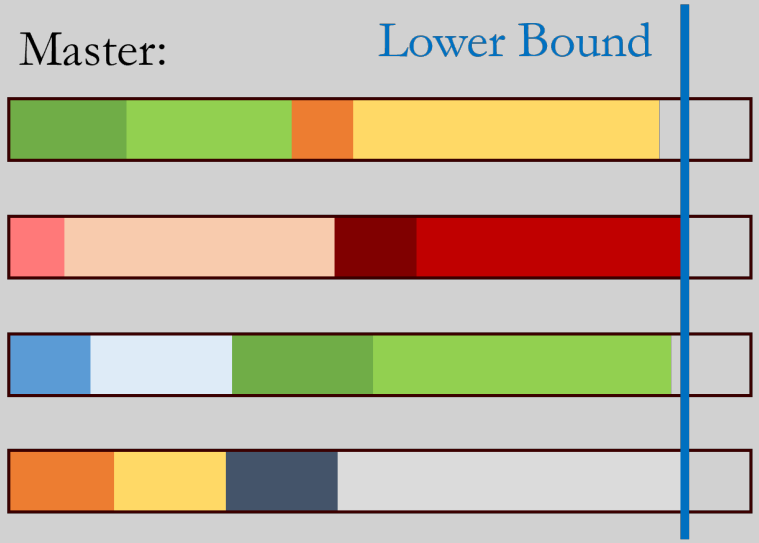
- Sum of processing times and setup times of the assigned parts of jobs

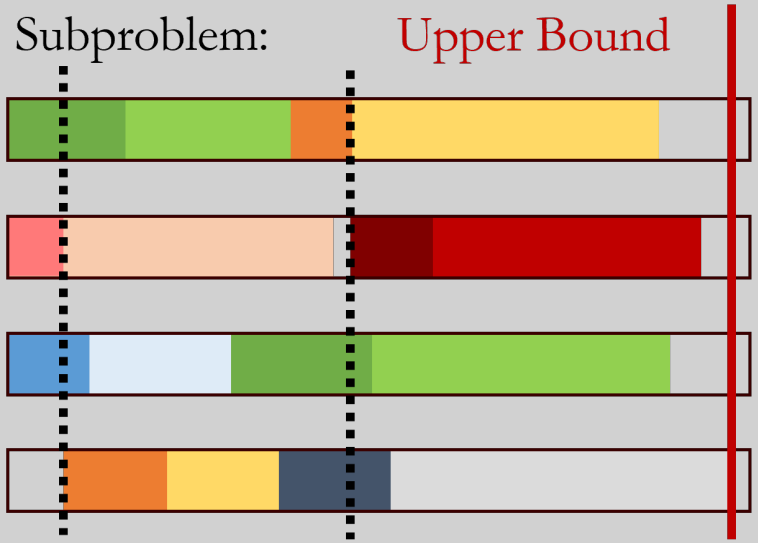
Industrial case

Subproblem

For the subproblem $\mathcal{S}$:

- For each machine, a sequence of parts of orders is known.
- The subproblem is not responsible for assigning orders to machines or for sequencing. It must simply adjust the start times of the setup operations, to ensure that no more than R working groups are occupied at the same time.





For the subproblem $\mathcal{S}$:

- For each machine, a sequence of parts of orders is known.
- The subproblem is not responsible for assigning orders to machines or for sequencing. It must simply adjust the start times of the setup operations, to ensure that no more than R working groups are occupied at the same time.

$$\min \zeta^k$$

subject to:

$$\texttt{Cumulative}((\texttt{start\_of}(\bar{\mu}^k_{i,m})), (\bar{s}^{k-1}_{i,m}), 1, R)$$

$$\texttt{start\_of}(\bar{\mu}^k_{i,m}) \geq \texttt{end\_of}(\bar{\mu}^k_{i-1,m}) + \bar{p}^{k-1}_{i-1,m} \qquad \forall m \in \bar{M}_{k-1}, i = 2, ..., |\bar{\nu}^{k-1}_m|$$

$$\zeta^k \geq \texttt{end\_of}(\bar{\mu}^k_{i,m}) + \bar{p}^{k-1}_{i,m} \qquad \forall m \in \bar{M}, i = 1, ..., |\bar{\nu}^{k-1}_m|$$

$$\bar{\mu}^k_{i,m} : \texttt{interval}(\text{size} = \bar{s}^{k-1}_{i,m}) \qquad \forall m \in \bar{M}, i = 1, ..., |\bar{\nu}^{k-1}_m|$$

For the cuts:

- Each iteration should define a different allocation of parts of jobs to machines.

$$\hat{W}_p^{k-1} - W_p^k + v \leq V \cdot \lambda_p^{\leq k} \qquad \forall p = (i,m) \in P^{k-1}$$

$$W_p^k - \hat{W}_p^{k-1} + v \leq V \cdot \lambda_p^{\geq k} \qquad \forall p = (i,m) \in P^{k-1}$$

$$\rho_p^k + 1 \geq \lambda_p^{\geq k} + \lambda_p^{\leq k} \qquad \forall p = (i,m) \in P^{k-1}$$

$$z \geq \zeta^k - \zeta^k \cdot \left[(|A^{k-1}| - \sum_{a \in A^{k-1}} x_a^k) + (|P^{k-1}| - \sum_{p \in P^{k-1}} \rho_p^k) \right]$$

Industrial case

Results

Instance		Machines														
		2			5			10			15			20		
		Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA	Gap	Time	GHA
J	10	0.89	< 1	4.64	1.23	5	10.65	3.82	1488	11.00	6.30	3507	17.02	10.86	6026	14.99
	20	0.51	< 1	2.41	1.98	11	6.57	3.96	5755	18.25	9.87	6940	20.59	10.94	7338	31.48
	30	0.39	1	2.20	1.73	83	8.71	5.16	4430	26.25	10.29	7035	30.20	10.63	7781	39.39
	40	0.57	< 1	1.04	2.13	260	6.31	5.72	5113	20.62	11.47	6809	32.57	15.42	7065	47.20
	50	0.49	9	0.98	1.48	3479	5.84	3.81	8740	17.86	9.47	7537	36.30	19.69	7274	54.54
	80	0.29	6	1.26	1.39	232	3.06	2.87	7545	18.06	9.35	8019	35.50	18.98	8221	53.82
	100	0.39	11	0.91	1.52	196	3.10	2.64	6879	14.64	7.03	6578	34.33	17.57	7952	48.99
	150	0.40	41	0.83	1.27	334	3.09	2.58	7836	13.99	5.93	8541	27.90	13.67	10109	46.17
	200	0.46	392	0.58	1.26	2645	2.85	2.33	8679	11.58	5.47	8925	28.64	17.95	7955	44.28
	300	0.44	597	0.56	1.18	5312	2.79	7.40	8185	14.97	-	-	-	-	-	-
	400	0.43	559	0.66	-	-	-	-	-	-	-	-	-	-	-	-
	500	0.44	689	0.54	-	-	-	-	-	-	-	-	-	-	-	-
	700	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Solvers

MILP/CP tools

Solvers

MILP solvers


```python
def masterProblem(): # Master problem 'M'
    model = ConcreteModel()

    model.y = Var(within = NonNegativeIntegers, bounds = (0, 10)) # Variable 'y'
    model.θ = Var(within = NonNegativeReals) # Auxiliary variable 'θ'

    model.obj = Objective(rule = model.y + model.θ, sense = minimize) # min y + θ
    model.constraints = ConstraintList()

    return model

def subProblem(y): # Subproblem 'S', given the value of variable 'y'
    model = ConcreteModel()

    model.u = Var(within = NonNegativeReals) # Variable 'u'

    model.obj = Objective(rule = model.u*(5 - y), sense = maximize) # max u*(5 - y)
    model.constraints = ConstraintList()

    model.constraints.add(model.u <= 2) # u <= 2

    return model
```

- Open-source library in Python interface
- Modeling (not solving) Mixed-Integer Linear Programs
- Compatible with most solvers – solve the constructed model
- Syntax which resembles with the natural language

Commercial solvers

- IBM CPLEX Optimization Studio 
- Gurobi Optimizer 

Open-source solvers

- GNU Linear Programming Kit - GLPK

https://www.gnu.org/software/glpk/

IBM CPLEX CP Optimizer.

- A **commercial solver** – not for free
- Academic licenses available

Modeling with IBM ILOG

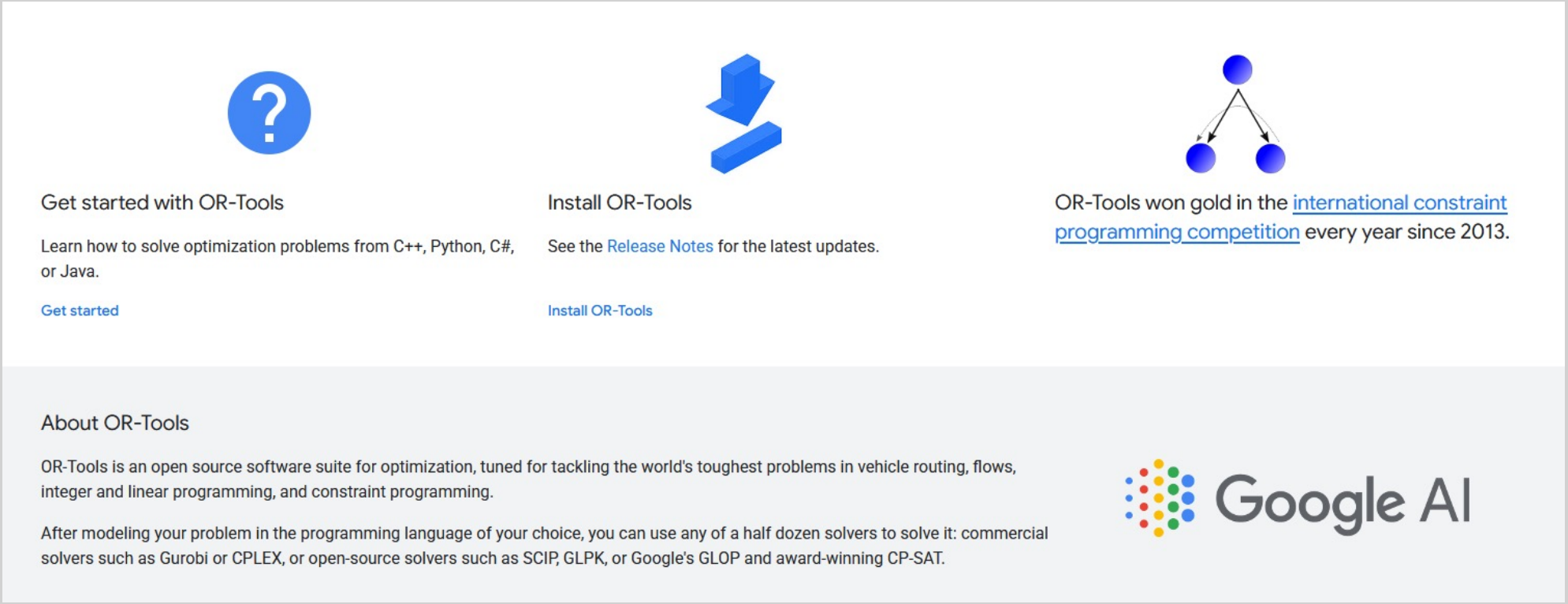
```
using CP;
int n = ...;
int m = ...;
int rd[1..n] = ...;
int dd[1..n] = ...;
float w[1..n] = ...;
int pt[1..n][1..m] = ...;
float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
dexpr int C[i in 1..n] = endOf(op[i][m]);
minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
subject to {
  forall(i in 1..n) {
    rd[i] <= startOf(op[i][1]);
    forall(j in 1..m-1)
      endBeforeStart(op[i][j],op[i][j+1]);
  }
  forall(j in 1..m)
    noOverlap(all(i in 1..n) op[i][j]);
}
```

Modeling with DOCplex Python API

```
model = CpoModel()

x = [model.integer_var(0, len(nodes)-1, name = f'nextOf_{numericIndices[i]}') for i in numericIndices.keys()]
y = [model.integer_var(0, len(nodes)-1, name = f'positionOf_{numericIndices[i]}') for i in numericIndices.keys()]

model.add(model.all_diff(x))
for i in numericIndices.keys():
    if numericIndices[i] != 'depot':
        model.add(model.element(y, x[i]) == y[i]+1)

model.add(model.minimize(sum(model.element(distanceMatrix[i], x[i]) for i in numericIndices.keys())))

sol = model.solve(TimeLimit = 60, trace_log = True)
```

CP modeler documentation

Open-source CP solvers: **Google OR-Tools**



https://developers.google.com/optimization

Open-source CP solvers: **Google OR-Tools**

```
cp = cp_model.CpModel()

startRoll = [cp.NewIntVar(rolling_state, big_M, f"startRoll_{j}") for j in instance]
endRoll = [cp.NewIntVar(rolling_state, big_M, f"endRoll_{j}") for j in instance]
rolling = [cp.NewIntervalVar(start = startRoll[j], size = math.ceil(rollDur[instance[j]]), end = endRoll[j],
cp.AddNoOverlap([rolling[j] for j in range(len(instance))])
for j in range(1, len(instance)):
    cp.Add(startRoll[j] >= endRoll[j-1])
    if instance[j] in strict:
        cp.Add(startRoll[j] == endRoll[j-1])
```

Modeling languages: **MiniZinc**

MiniZinc

MiniZinc is a high-level constraint modelling language that allows you to easily express and solve discrete optimisation problems.

Get started ▶ | Windows 10 or later

Latest release: 2.9.2 (changelog)

Packages | Source code | License information

MiniZinc is developed at Monash University with support from OPTIMA.

	Mon	Tue	Wed	Thu	Fri
Aimee		Day			Night
Beula	Night	Evening	Day	Day	Day
Ciara		Day	Night		
Darby	Evening	Evening	Day	Night	Evening
Ernst	Night	Evening	Evening	Day	Evening
Green	Day	Day	Night	Evening	Evening
Jesse	Evening	Night		Day	Day
Katie			Day	Night	
Lloyd	Day	Evening		Evening	Night
Mable	Day	Night	Evening	Evening	Day
Nevin	Evening	Day	Evening	Day	Evening
Paula	Evening		Evening	Evening	Day

Rostering

https://www.minizinc.org/

References

J.F.Benders,
Partitioning procedures for solving mixed-variables programming problems.
Numerische Mathematik (4:3), 1962

J.N.Hooker,
Logic, Optimization, and Constraint Programming.
INFORMS Journal on Computing (14:4), 2002

H.Simonis,
Sudoku as a Constraint Problem.
https://ai.dmi.unibas.ch/_files/teaching/fs21/ai/material/ai26-simonis-cp2005ws.pdf

G.Codato, M.Fischetti,
Combinatorial Benders' Cuts.
Operations Research (55:3), 2007

J.N.Hooker,
Planning and Scheduling by Logic-Based Benders Decomposition.
Integer Programming and Combinatorial Optimization, 2004


Thank you!