Introduction to Google OR-Tools

Google OR-Tools

PROS Lab Members

Flavio Corradini

FULL PROFESSOR

Andrea Polini

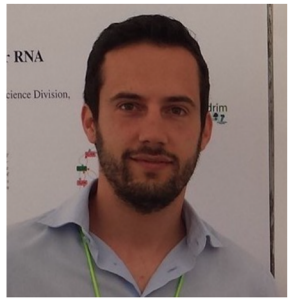
ASSOCIATE PROFESSOR

Barbara Re

ASSOCIATE PROFESSOR

Francesco Tiezzi

ASSOCIATE PROFESSOR



Andrea Morichetta

RESEARCH FELLOW

Fabrizio Fornari

POSTDOCTORAL RESEARCHER

Lorenzo Rossi

POSTDOCTORAL RESEARCHER

Marco Piangerelli

POSTDOCTORAL RESEARCHER

Alessandro Marcelletti

PHD STUDENT

Caterina Luciani

PHD STUDENT

Khalid Bourr

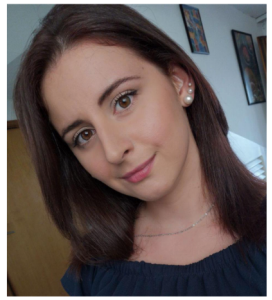
PHD STUDENT

Ivan Compagnucci

PHD STUDENT

Sara Pettinari

PHD STUDENT



Arianna Fedeli

PHD STUDENT

and…

- Morena Barboni
- Vincenzo Nucci
- Ahmad Ronaghikhameneh
- Umair Qureshi

Ivan Compagnucci

- Ph.D. student at UNICAM
- PROS Lab Member

PROS Lab
pros.unicam.it

PROS
PROcesses and Services Lab

pros@unicam.it

http://pros.unicam.it

Interests

- Business Process Management
- BPMN
- **BP** & **IoT** modeling and enactment

Ivan Compagnucci

PHD STUDENT

ivan.compagnucci@unicam.it

https://www.linkedin.com/in/ivan-compagnucci/

ivan.compagnucci@studenti.unicam.it

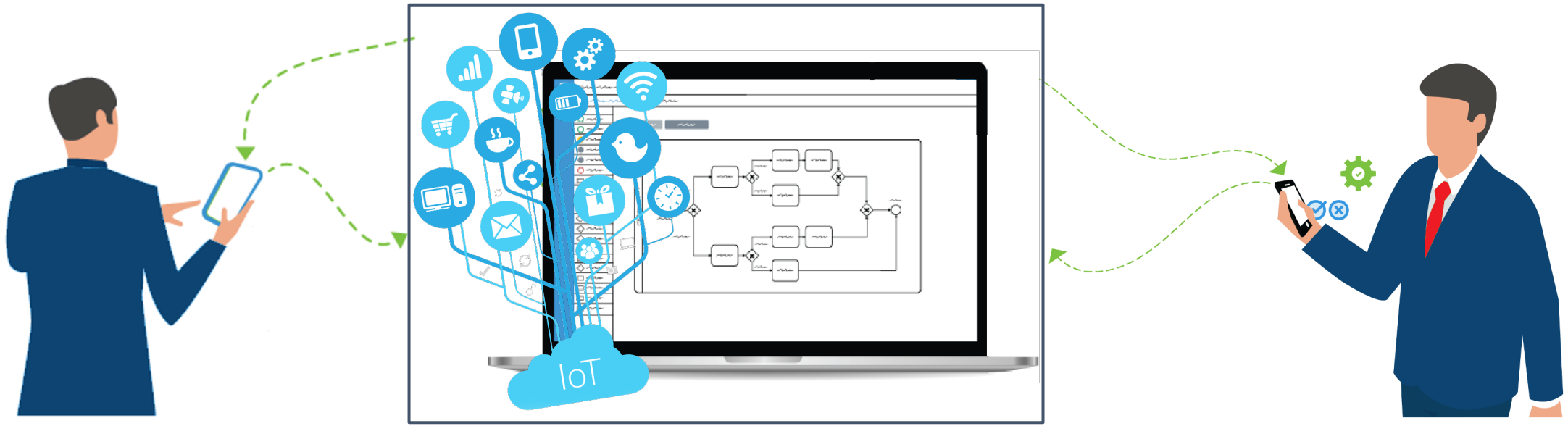
Research Topics

Internet of Things

Network of interconnected devices that collect and exchange data to monitor, control or transfer relevant information so as to be able to perform consequent intelligent actions

Business Process

A set of activities, tasks or actions to carry out a specific organizational goal such as a service or a product



Business Process Meet Internet of Things

- **Design** and **monitoring** of the smart environment for a **better execution**, **safety** and **less complexity**
- **Bridging the gap** between the high level of the Business Process and the low level of the IoT technologies
- Programming of "**dependencies between independent devices**“ in a process-oriented vision

Process-Oriented Modelling Notations for Internet of Things

Ivan Compagnucci
ivan.compagnucci@unicam.it
Computer Science Division, Science and Technology School, University of Camerino

PROS Lab
pros.unicam.it

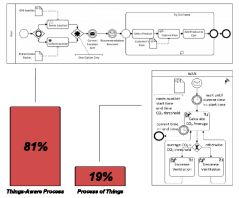
Business Process meet Internet of Things

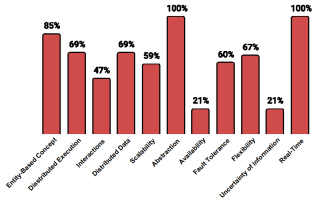
The **Internet of Things** term refers to the inter-networking of physical objects embedded with electronics hardware, software, sensors, actuators, and network connectivity.

The **Business Process Management** is a well-established discipline that deals with the analysis, design, implementation, execution, monitoring, and evolution of business processes.

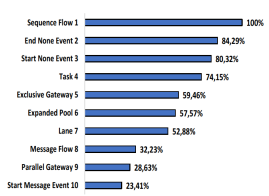
SLR Research Questions

- RQ1. Which are the **relevant modelling perspectives** to consider when modelling IoT-Aware business processes?
- RQ2. What are the **IoT requirements supported** by notations used to model IoT-Aware business processes?
- RQ3. Which are the **modelling notations proposed** and adopted to model IoT-Aware business processes?

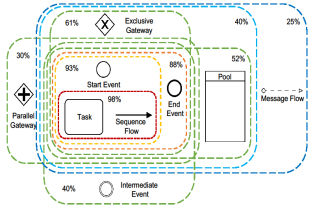




Trends on the Usage of BPMN 2.0 Standard Notation



Frequency Distribution of BPMN Elements

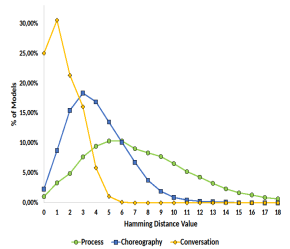


Subsets of common BPMN Elements

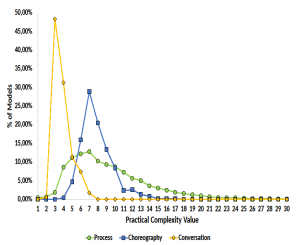
There isn't a standard *de facto* for modeling smart scenarios in a process-oriented way. Existing solutions **mainly use BPMN** or **an extension of it**.

14% Not BPMN
86% BPMN or BPMN Extension

However, there is no a well-established and mature approach that **fully meet the requirements to represent an IoT scenario**.



Hamming Distance of BPMN Elements



Practical Complexity of BPMN Elements

Element One	Element Two	ρ
Process		
Receive Task	Send Task	0.76
Sequence Flow	Task	0.75
Sequence Flow	Exclusive Gateway	0.73
Intermediate Throw Link Event	Intermediate Catch Link Event	0.72
End Event	Start Event	0.68
Start Event	Expanded Sub-Process	0.64
Intermediate Throw Message Event	Intermediate Catch Message Event	0.59
Association Data Output	Data Object	0.59
Expanded Pool	Lane	0.58
Sequence Flow	End Event	0.55
Exclusive Gateway	Task	0.53
Message Flow	Intermediate Catch Message Event	0.53
Expanded Pool	Message Flow	0.52
End Event	Expanded Sub-Process	0.50
Intermediate Throw Signal Event	Intermediate Catch Signal Event	0.50
Choreography		
Choreography Participant	Choreography Task	0.96
Choreography Task	Choreography Message	0.54
Choreography Participant	Choreography Message	0.51
Conversation		
Conversation Link	Conversation	0.98
Conversation Link	Collapsed Pool	0.54
Conversation	Collapsed Pool	0.54

BPMN Elements Pairs Correlation

References

[1] I. Compagnucci, F. Corradini, F. Fornari, A. Polini, B. Re and F. Tiezzi. Modelling Notations for IoT-Aware Business Processes: A Systematic Literature Review. BPM Workshop, Vol. 397, 108-121, 2020.
[2] I. Compagnucci, F. Corradini, F. Fornari, and B. Re. Trends on the Usage of BPMN 2.0 from Publicly Available Repositories. In: Perspectives in Business Informatics Research, Vol. 430, 84-99, 2021.

Findings

There isn't a standard de facto for modeling smart scenarios in a process- oriented way. Existing solutions (86%) **mainly use BPMN or an extension of it**

However, **there is no a well-established and mature approach** that fully meet the requirements to represent an IoT scenario.

...Constraint programming can improve IoT systems?

Constraint Programming

Constraint programming is a powerful paradigm for **solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence (AI), operations research, algorithms, and graph theory.**

The basic idea in constraint programming is that **the user states the constraints,** and a general-purpose **"constraint solver" is used to solve them.**

Constraint Programming: Everyday-life example

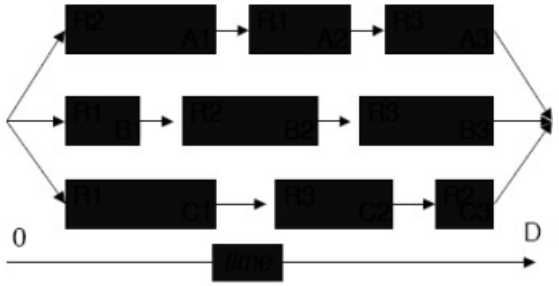
Sudoku

- Rules for inserting numbers in the table

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

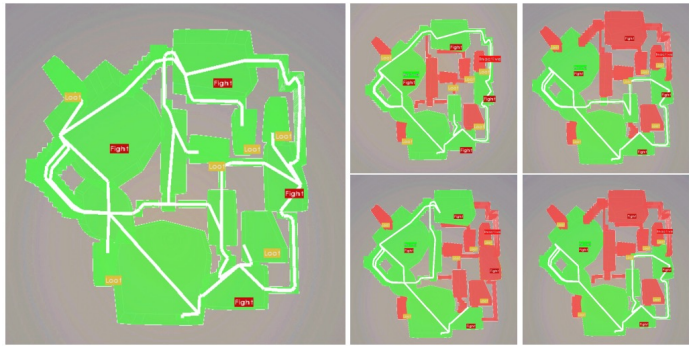
CPU's Job Scheduling

- Constraints on determing which process should be executed



Gaming: Procedural Dungeon Simulation

- Procedural dungeon generation
  in an open world/universe context



What is OR Tools?

OR-Tools is an open source software suite for **optimization, for solve problems in vehicle routing, network flows, integer and linear programming, and constraint programming.**

Google OR-Tools

OR Tools: Identify the type of Solvers

There are **many different types of optimization problems in the world**. For each type of problem, there are **different approaches and algorithms for finding an optimal solution**. Before you can start writing a program to solve an optimization problem, you need to **identify what type of problem you are dealing with, and then choose an appropriate solver** — an algorithm for finding an optimal solution.

These are the types of problems that OR-Tools solves:

- Linear optimization
- Mixed-Integer optimization
- Constraint optimization
- Network flows/ Routing
- Assignment
- Scheduling

OR Tools: Installation

Download the following resources:

- **“Constraints”:**
  - You must have the **Microsoft Visual C++ Redistributable for Visual Studio 2019.** https://visualstudio.microsoft.com/downloads/?q=Visual+C%2B%2B+Redistributable+for+Visual+Studio
  - You must also have a **Java JDK 64 bit, version 8.0 or later installed**. https://java.com/en/download/help/download_options.html
  - You must also have a **Maven 64 bit installed**. https://maven.apache.org/download.cgi

- **Visual Studio Code 2022:** As IDE.
- **OR-Tools Binary Distribution:**
  1. Visit: **https://developers.google.com/optimization/install**
  2. Select the distribution depending on your Operating System.
  3. Unzip

OR Tools: Installation

Testing the installation (Inside the unzipped folder):

```
> tools\make test_java
```

You should be able to have this output:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] --- maven-install-plugin:3.0.0-M1:install-file (default-cli) @ standalone-pom ---
[INFO] Installing C:\OR-Tools\ortools-win32-x86-64-9.3.10497.jar to C:\Users\User\.m2\rep
ols-win32-x86-64\9.3.10497\ortools-win32-x86-64-9.3.10497.jar
[INFO] Installing C:\Users\User\AppData\Local\Temp\ortools-win32-x86-64-9.3.1049748104416
r\.m2\repository\com\google\ortools\ortools-win32-x86-64\9.3.10497\ortools-win32-x86-64-9
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.864 s
[INFO] Finished at: 2022-04-14T16:07:39+02:00
[INFO] ------------------------------------------------------------------------
```

Possible errors

Make sure you:

- **Defined environmental variables (PATH):**
  - **For JDK, Maven, cmd view, …**
- Microsoft Visual C++ Redistributable **is up to date!**
- Install MinGW (If "*make*" command doesn't work)
- …

OR Tools: What is an optimization problem?

The goal of **optimization** is to find the *best* solution to a problem out of a large set of possible solutions. (Sometimes you'll be satisfied with finding any feasible solution; OR-Tools can do that as well.)

A typical optimization problem:

Here's a typical optimization problem. Suppose that a **shipping company delivers packages to its customers using a fleet of trucks**. Every day, the company must assign packages to trucks, and then choose a route for each truck to deliver its packages. Each possible assignment of packages and routes has a cost, based on the total travel distance for the trucks, and possibly other factors as well. The problem is to choose the assignments of packages and routes that has the least cost.

OR Tools: What is an optimization problem?

Like all optimization problems, this problem has the following 2 elements:

- **The Objective:** The quantity you want to optimize. Here, the objective is to **minimize cost**. To set up an optimization problem, you need to **define a function that calculates the value of the objective for any possible solution**. This is called the ***objective function***. **In the preceding example, the objective function would calculate the total cost of any assignment of packages and routes**.

*An **optimal solution** is one for which the value of the objective function is the best.*

- **The Constraints:** Restrictions on the **set of possible solutions, based on the specific requirements of the problem**. For example, if the shipping company can't assign packages above a given weight to trucks, this would impose a constraint on the solutions.

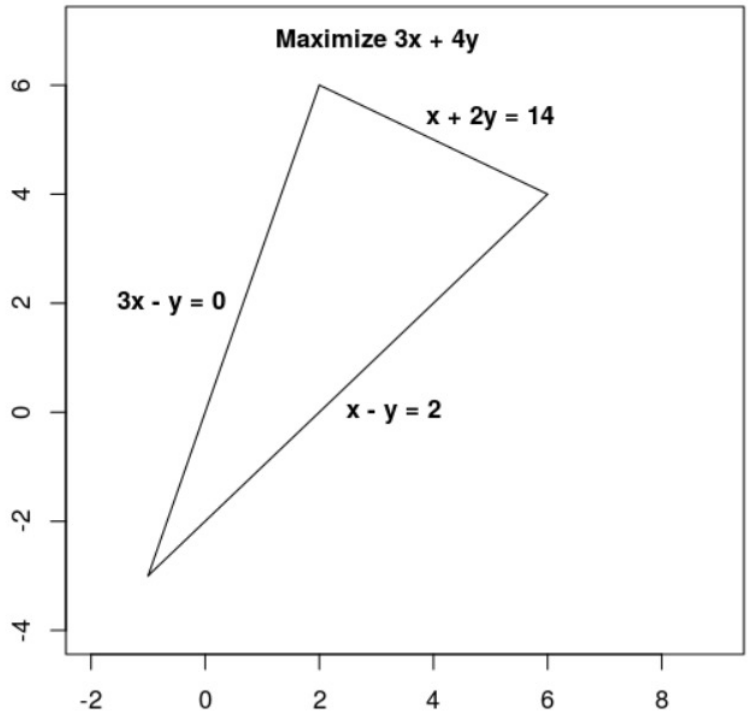
*A **feasible solution** is one that satisfies all the given constraints for the problem, without necessarily being optimal.*

Another linear optimization problem in Java

A linear optimization example:

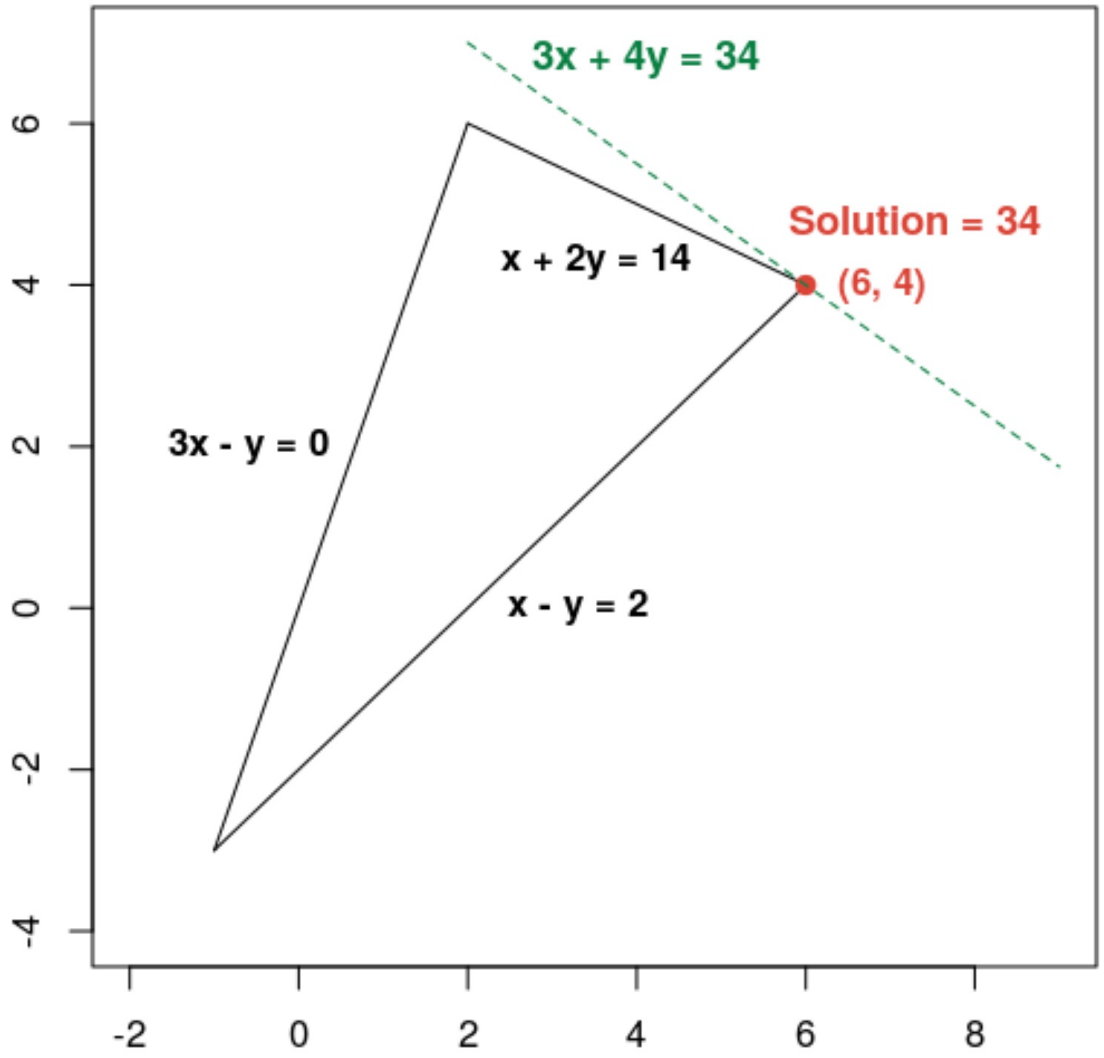
- **Maximize** $3x + 4y$ subject to the following constraints:
- Both the objective function and the constraints are given by linear expressions, which makes this a linear problem.

$x + 2y$	$\leq$	14
$3x - y$	$\geq$	0
$x - y$	$\leq$	2



The constraints define the feasible region, which is the triangle shown below, including its interior.

Solving a linear optimization problem in Java

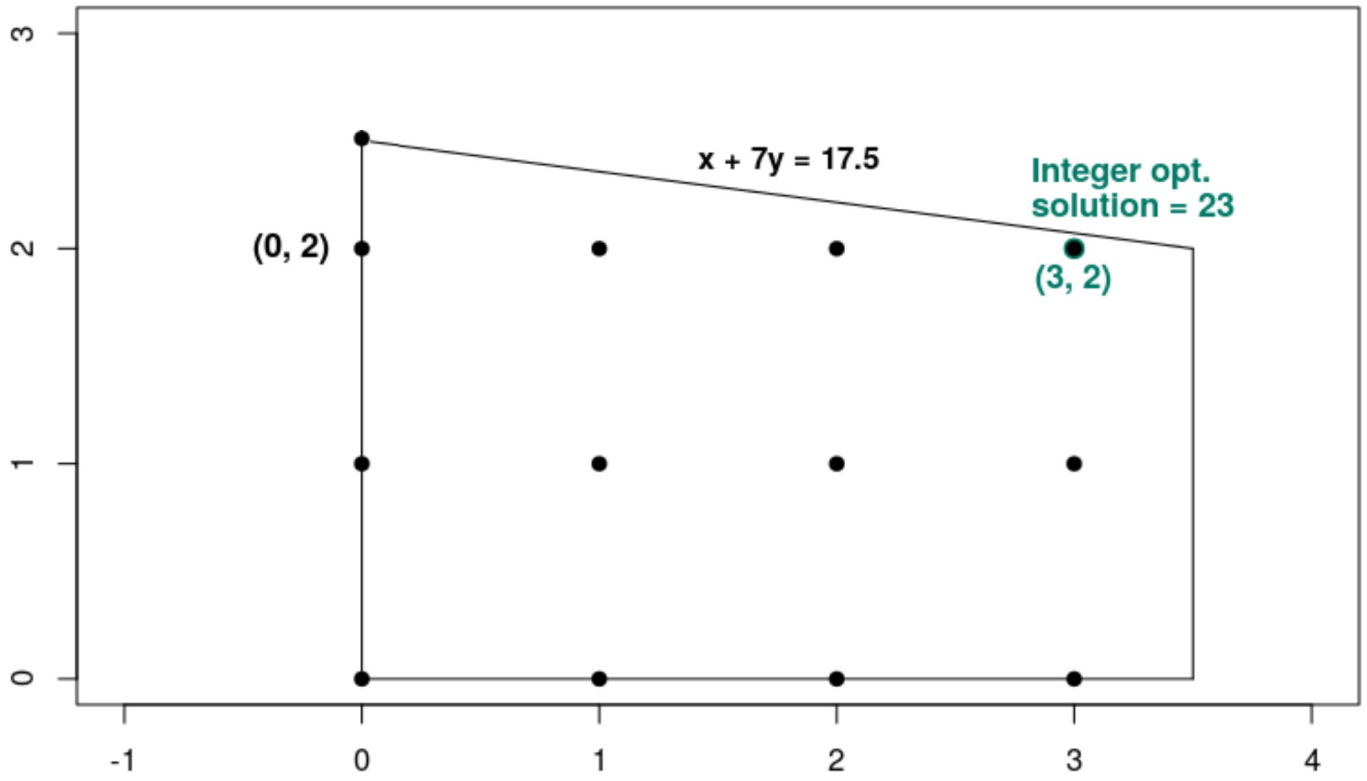


The dashed green line is defined by setting the objective function equal to its optimal value of 34.

Any line whose equation has the form $3x + 4y = c$ is parallel to the dashed line, and 34 is the largest value of c for which the line intersects the feasible region

```
Number of variables = 2
Number of constraints = 3
Solution:
x = 6.0
y = 4.0
Optimal objective value = 34.0
```

Mixed-Integer optimization problem in Java



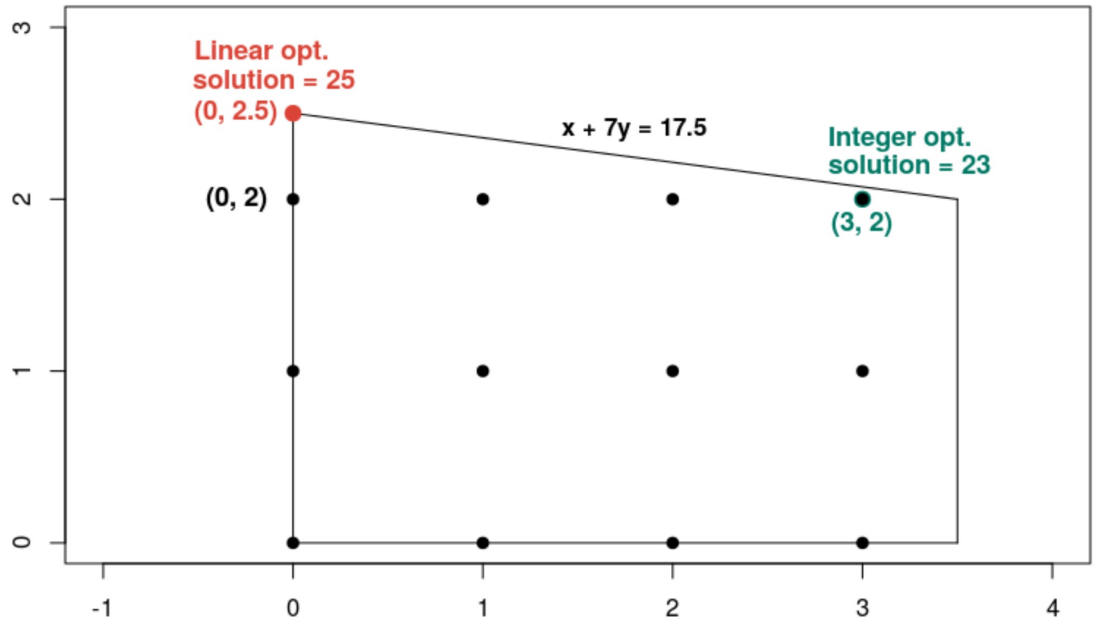
The optimal value of the objective function is **23**, which occurs at the point x = 3, y = 2.

```
Number of variables = 2
Number of constraints = 2
Solution:
Objective value = 23
x = 3
y = 2
```

Comparing Linear and Integer Optimization

Let's **compare** the solution to the integer optimization problem, with the solution to the corresponding linear optimization problem, **in which integer constraints are removed.**

1. Replace the **MIP Solver** with **LP Solver;** (SCIP -> GLOP)
2. Replace **Integer Variables** with **Continuous Variables;** (makeIntVar -> makeNumVar)
3. Check differences!



Results:

MIP:

```
Objective value = 23
x = 3
y = 2
```

LP:

```
Objective value = 25.000000
x = 0.000000
y = 2.500000
```

The integer solution is not close to the linear solution. The solutions to a linear optimization problem and the corresponding integer optimization **problems can be far apart**. Because of this, the two types of problems require different methods for their solution.

OR-Tools: Constraints Programming

Constraint optimization, or *constraint programming* (CP), is the name given **to identifying feasible solutions out of a very large set of candidates**, where the problem can be modeled in terms of arbitrary constraints.

CP is based on:

- **Feasibility** (finding a feasible solution) rather than **Optimization** (finding an optimal solution)
- Focuses on **constraints** and variables rather than the **objective function**

A CP Problem **may not have an objective function!** Because the idea is to **find a vary large set of possible solutions to a more manageable subset by adding constraints to the problem.**

OR-Tools: Constraints Programming

Employee Scheduling example:

The problem arises when companies that operate continuously — such as factories — need to create weekly schedules for their employees.

The company runs three 8-hour shifts per day and assigns three of its four employees to different shifts each day, while giving the fourth the day off.

Actors:

- Company
- 3 runs every day (8-hours)
- 4 Employees
- Every day a different employee has a day off

OR-Tools: Constraints Programming

Even in such a small case, the number of possible schedules is huge:

- On each day there are $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ possible employee assignments;
- The number of possible weekly schedules is 247, which is over 4.5 billion.

Note that, **usually, there will be other constraints that reduce the number of feasible solutions**:

- In example: *Each employee should work at least a minimum number of days per week*

However, the **CP method keeps track of which solutions remain feasible when you add new constraints,** making it a powerful tool for solving large, real-world scheduling problems.

OR-Tools: Constraints Programming

Even in such a small case, the number of possible schedules is huge:

- On each day there are $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ possible employee assignments;
- The number of possible weekly schedules is 247, which is over 4.5 billion.

Note that, **usually, there will be other constraints that reduce the number of feasible solutions**:

- In example: *Each employee should work at least a minimum number of days per week*

However, the **CP method keeps track of which solutions remain feasible when you add new constraints,** making it a powerful tool for solving large, real-world scheduling problems.

The Employee Scheduling problem

Organizations whose employees work multiple shifts need to schedule sufficient workers for each daily shift. Typically, the schedules will have constraints, such as:

“No employee should work two shifts in a row“

Finding a schedule that satisfies all constraints can be computationally difficult.

OR-Tools provide the CP-SAT Solver for solve such problems:

```
CpSolver solver = new CpSolver();
CpSolverStatus status = solver.solve(model);
```

The Employee Scheduling problem:CP-SAT Solver

The CP-SAT Solver returns one of the status values shown in the table below:

Status	Description
`OPTIMAL`	An optimal feasible solution was found.
`FEASIBLE`	A feasible solution was found, but we don't know if it's optimal.
`INFEASIBLE`	The problem was proven infeasible.
`MODEL_INVALID`	The given CpModelProto didn't pass the validation step. You can get a detailed error by calling `ValidateCpModel(model_proto)`.
`UNKNOWN`	The status of the model is unknown because no solution was found (or the problem was not proven INFEASIBLE) before something caused the solver to stop, such as a time limit, a memory limit, or a custom limit set by the user.

The Employee Scheduling problem

In particular the example is about a **nurse scheduling problem.** A **hospital supervisor needs to create a schedule** for **four nurses** over a **three-day period**, subject to the following conditions:

- **Each day is divided into three 8-hour shifts**
- **Every day, each shift is assigned to a single nurse, and no nurse works more than one shift.**
- **Each nurse is assigned to at least two shifts during the three-day period.**

The Employee Scheduling problem: step by step

1. Import the required libraries:

```
import com.google.ortools.Loader;
import com.google.ortools.sat.CpModel;
import com.google.ortools.sat.CpSolver;
import com.google.ortools.sat.CpSolverSolutionCallback;
import com.google.ortools.sat.CpSolverStatus;
import com.google.ortools.sat.LinearExpr;
import com.google.ortools.sat.LinearExprBuilder;
import com.google.ortools.sat.Literal;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;
```

2. Create Variables

```
final int numNurses = 4;
final int numDays = 3;
final int numShifts = 3;

final int[] allNurses = IntStream.range(0, numNurses).toArray();
final int[] allDays = IntStream.range(0, numDays).toArray();
final int[] allShifts = IntStream.range(0, numShifts).toArray();
```

The Employee Scheduling problem: step by step

3. Create the CP Model:

```
CpModel model = new CpModel();
```

4. Create an array of variables:

```
Literal[][][] shifts = new Literal[numNurses][numDays][numShifts];
for (int n : allNurses) {
  for (int d : allDays) {
    for (int s : allShifts) {
      shifts[n][d][s] = model.newBoolVar("shifts_n" + n + "d" + d + "s" + s);
    }
  }
}
```

The array defines assignments for shifts to nurses as follows:

shifts[(n, d, s)] equals **1 if shift s is assigned to nurse n on day d**, and **0 otherwise**.

The Employee Scheduling problem: step by step

5. Next, we assign nurses to shifts subject to the following constraints:

- Each shift is assigned to a single nurse per day

```
for (int d : allDays) {
  for (int s : allShifts) {
    List<Literal> nurses = new ArrayList<>();
    for (int n : allNurses) {
      nurses.add(shifts[n][d][s]);
    }
    model.addExactlyOne(nurses);
  }
}
```

The last line says **that for each shift, the sum of the nurses assigned to that shift is 1.**

- Each nurse works at most one shift per day

```
for (int n : allNurses) {
  for (int d : allDays) {
    List<Literal> work = new ArrayList<>();
    for (int s : allShifts) {
      work.add(shifts[n][d][s]);
    }
    model.addAtMostOne(work);
  }
}
```

For each nurse, **the sum of shifts assigned to that nurse is at most 1** ("at most" because a nurse might have the day off).

The Employee Scheduling problem: step by step

Next, we need to **assign shifts to nurses as evenly as possible:**

Since there are **nine shifts over the three-day period**, we can assign **two shifts to each of the four nurses**. After that there will be **one shift left over**, which can be assigned to any nurse.

```
int minShiftsPerNurse = (numShifts * numDays) / numNurses;
int maxShiftsPerNurse;
if ((numShifts * numDays) % numNurses == 0) {
  maxShiftsPerNurse = minShiftsPerNurse;
} else {
  maxShiftsPerNurse = minShiftsPerNurse + 1;
}
for (int n : allNurses) {
  LinearExprBuilder numShiftsWorked = LinearExpr.newBuilder();
  for (int d : allDays) {
    for (int s : allShifts) {
      numShiftsWorked.add(shifts[n][d][s]);
    }
  }
  model.addLinearConstraint(numShiftsWorked, minShiftsPerNurse, maxShiftsPerNurse);
}
```

minShiftsPerNurse: used to distribute the shifts evenly.

Note: *If this is not possible, because the total number of shifts is not divisible by the number of nurses, some nurses will be assigned one more shift.*

The Employee Scheduling problem: step by step

Since there are num_shifts * num_days total shifts in the schedule period, you can assign at least

```
(num_shifts * num_days) // num_nurses
```

Shifts to each nurse, but some shifts may be left over. For the given values of:

num_nurses = 4,

num_shifts = 3,

num_days = 3,

The expression min_shifts_per_nurse has the value (3 * 3 / 4) = 2, so you can assign at least two shifts to each nurse. **This is guaranteed by the constraint.**

```
model.Add(min_shifts_per_nurse <= num_shifts_worked)
```

The extra shift can be assigned to any nurse:

```
model.Add(num_shifts_worked <= max_shifts_per_nurse)
```

(Ensures that no nurse is assigned more than one extra shift. The constraint isn't necessary in this case, because there's only one extra shift)

The Employee Scheduling problem: step by step

6. Update the solver parameters:

```
CpSolver solver = new CpSolver();
solver.getParameters().setLinearizationLevel(0);
// Tell the solver to enumerate all solutions.
solver.getParameters().setEnumerateAllSolutions(true);
```

7. Invoke the solver and display solution:

```
CpSolverStatus status = solver.solve(model, cb);
System.out.println("Status: " + status);
System.out.println(cb.getSolutionCount() + " solutions found.");
```

OR-Tools has found **5184** possible solutions!! But... how?

The Employee Scheduling problem: Solutions

```
Solution 0
Day 0
  Nurse 0 does not work
  Nurse 1 works shift 0
  Nurse 2 works shift 1
  Nurse 3 works shift 2
Day 1
  Nurse 0 works shift 2
  Nurse 1 does not work
  Nurse 2 works shift 1
  Nurse 3 works shift 0
Day 2
  Nurse 0 works shift 2
  Nurse 1 works shift 1
  Nurse 2 works shift 0
  Nurse 3 does not work
```

```
Solution 1
Day 0
  Nurse 0 works shift 0
  Nurse 1 does not work
  Nurse 2 works shift 1
  Nurse 3 works shift 2
Day 1
  Nurse 0 does not work
  Nurse 1 works shift 2
  Nurse 2 works shift 1
  Nurse 3 works shift 0
Day 2
  Nurse 0 works shift 2
  Nurse 1 works shift 1
  Nurse 2 works shift 0
  Nurse 3 does not work
```

```
Solution 2
Day 0
  Nurse 0 works shift 0
  Nurse 1 does not work
  Nurse 2 works shift 1
  Nurse 3 works shift 2
Day 1
  Nurse 0 works shift 1
  Nurse 1 works shift 2
  Nurse 2 does not work
  Nurse 3 works shift 0
Day 2
  Nurse 0 works shift 2
  Nurse 1 works shift 1
  Nurse 2 works shift 0
  Nurse 3 does not work
```

```
Solution 3
Day 0
  Nurse 0 does not work
  Nurse 1 works shift 0
  Nurse 2 works shift 1
  Nurse 3 works shift 2
Day 1
  Nurse 0 works shift 1
  Nurse 1 works shift 2
  Nurse 2 does not work
  Nurse 3 works shift 0
Day 2
  Nurse 0 works shift 2
  Nurse 1 works shift 1
  Nurse 2 works shift 0
  Nurse 3 does not work
```

```
Solution 4
Day 0
  Nurse 0 does not work
  Nurse 1 works shift 0
  Nurse 2 works shift 1
  Nurse 3 works shift 2
Day 1
  Nurse 0 works shift 2
  Nurse 1 works shift 1
  Nurse 2 does not work
  Nurse 3 works shift 0
Day 2
  Nurse 0 works shift 2
  Nurse 1 works shift 1
  Nurse 2 works shift 0
  Nurse 3 does not work
```

```
Statistics
  - conflicts      : 5
  - branches       : 142
  - wall time      : 0.002484 s
  - solutions found: 5
```

There are 4 choices for the one nurse who works an extra shift. Having chosen that nurse, there are 3 shifts the nurse can be assigned to on each of the 3 days, **so the number of possible ways to assign the nurse with the extra shift is $4 \cdot 33 = 108$**. After assigning this nurse, there are two remaining unassigned shifts on each day..

The Employee Scheduling problem: Solutions

Of the remaining three nurses, one works days 0 and 1, one works days 0 and 2, and one works days 1 and 2. There are 3! = 6 ways to assign the nurses to these days:

Day 0	Day 1	Day 2
A B	A C	B C
A B	B C	A C
A C	A B	B C
A C	B C	A B
B C	A B	A C
B C	A C	A B

(The three nurses are labeled A, B, and C, and we have not yet assigned them to shifts.)

For each row in the above diagram, there are $2^3 = 8$ possible ways to assign the remaining shifts to the nurses (two choices on each day). So the total number of possible assignments is **108·6·8 = 5184**.

The Job Shop problem

One common scheduling problem is the ***job shop***, in which **multiple jobs are processed on several machines**. **Each job consists of a sequence of tasks, which must be performed in a given order, and each task must be processed on a specific machine**.

The problem is to schedule the tasks on the machines so **as to minimize the *length* of the schedule**—the time it takes for all the jobs to be completed.

There are several constraints for the job shop problem:

- **No task for a job can be started until the previous task for that job is completed;**
- **A machine can only work on one task at a time;**
- **A task, once started, must run to completion.**

The Job Shop problem

Each task is labeled by a pair of numbers **(m,p)** where:

- **m** is the number of the machine the task must be processed on;
- **p** is the processing time of the task (the amount of time it requires).

In the example, job 0 has three tasks. The first, (0, 3), must be processed on machine 0 in 3 units of time. The second, (1, 2), must be processed on machine 1 in 2 units of time, and so on.

Altogether, there are eight tasks:

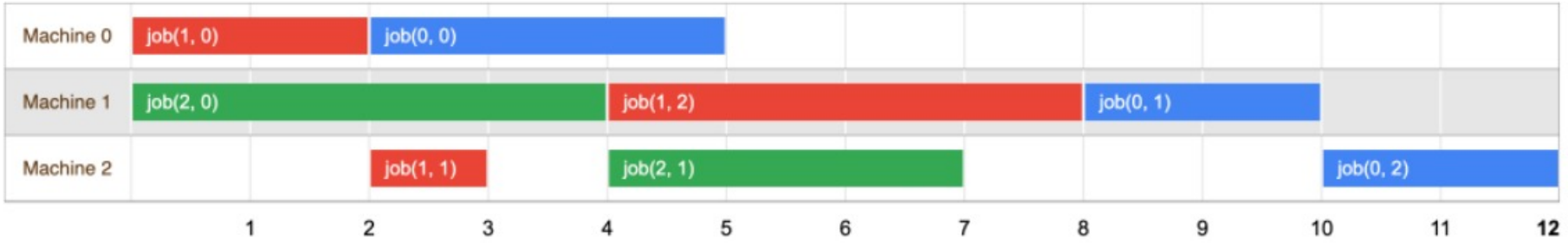
job 0 = [(0, 3), (1, 2), (2, 2)]

job 1 = [(0, 2), (2, 1), (1, 4)]

job 2 = [(1, 4), (2, 3)]

The Job Shop problem: A solution for the problem

A solution to the job shop problem is an assignment of a start time for each task, which meets the constraints given above. The diagram below shows one possible solution for the problem:



job 0 = [(0, 3), (1, 2), (2, 2)]

job 1 = [(0, 2), (2, 1), (1, 4)]

job 2 = [(1, 4), (2, 3)]

All the tasks for each job are scheduled at non-overlapping time intervals, in the order given by the problem.

The length of this solution is 12, which is the first time when all three jobs are complete.

However, as you will see below, **this is not the optimal solution to the problem!!**

The Job Shop problem: Variables and constraints for the problem

First, let task(i, j) denote the jth task in the sequence for job i.

For example, task(0, 2) denotes the second task for job 0, which corresponds to the pair (1, 2) in the problem description.

Next, define ti, j to be the start time for task(i,j). The $t_{i,j}$ are the variables in the job shop problem. Finding a solution involves determining values for these variables that meet the requirement of the problem.

There are two types of constraints for the job shop problem:

- ***Precedence constraints****: These arise from the condition that for any two consecutive tasks in the same job, the first must be completed before the second can be started.*

For example, task(0, 2) and task(0, 3) are consecutive tasks for job 0. Since the processing time for task(0, 2) is 2, the start time for task(0, 3) must be at least 2 units of time after the start time for task 2. (Perhaps task 2 is painting a door, and it takes two hours for the paint to dry.) As a result, you get the following constraint:

$$t_{0,2} + 2 \leq t_{0,3}$$

The Job Shop problem: Variables and constraints for the problem

- ***No overlap constraints***: *These arise from the restriction that a machine can't work on two tasks at the same time. For example, task(0, 2) and task(2, 1) are both processed on machine 1. Since their processing times are 2 and 4, respectively, one of the following constraints must hold:*

$$t_{0,2} + 2 \leq t_{2,1} \quad \textit{if(task(0,2) is scheduled before task(2,1))} \qquad \textbf{OR}$$

$$t_{2,1} + 4 \leq t_{0,2} \quad \textit{if(task(2,1) is scheduled before task(0,2))}$$

Objective for the problem:

The objective of the job shop problem is to minimize the *makespan*: the length of time from the earliest start time of the jobs to the latest end time.

The Employee Scheduling problem: step by step

1. Import the required libraries:

```
import static java.lang.Math.max;

import com.google.ortools.Loader;
import com.google.ortools.sat.CpModel;
import com.google.ortools.sat.CpSolver;
import com.google.ortools.sat.CpSolverStatus;
import com.google.ortools.sat.IntVar;
import com.google.ortools.sat.IntervalVar;
import com.google.ortools.sat.LinearExpr;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.IntStream;
```

2. Define the data

```
class Task {
  int machine;
  int duration;
  Task(int machine, int duration) {
    this.machine = machine;
    this.duration = duration;
  }
}

final List<List<Task>> allJobs =
    Arrays.asList(Arrays.asList(new Task(0, 3), new Task(1, 2), new Task(2, 2)), // Job0
        Arrays.asList(new Task(0, 2), new Task(2, 1), new Task(1, 4)), // Job1
        Arrays.asList(new Task(1, 4), new Task(2, 3)) // Job2
    );

int numMachines = 1;
for (List<Task> job : allJobs) {
  for (Task task : job) {
    numMachines = max(numMachines, 1 + task.machine);
  }
}
final int[] allMachines = IntStream.range(0, numMachines).toArray();

// Computes horizon dynamically as the sum of all durations.
int horizon = 0;
for (List<Task> job : allJobs) {
  for (Task task : job) {
    horizon += task.duration;
  }
}
```

3. Declare the model

```
CpModel model = new CpModel();
```

The Employee Scheduling problem: step by step

4. Define the variables:

```
class TaskType {
  IntVar start;
  IntVar end;
  IntervalVar interval;
}
Map<List<Integer>, TaskType> allTasks = new HashMap<>();
Map<Integer, List<IntervalVar>> machineToIntervals = new HashMap<>();

for (int jobID = 0; jobID < allJobs.size(); ++jobID) {
  List<Task> job = allJobs.get(jobID);
  for (int taskID = 0; taskID < job.size(); ++taskID) {
    Task task = job.get(taskID);
    String suffix = "_" + jobID + "_" + taskID;

    TaskType taskType = new TaskType();
    taskType.start = model.newIntVar(0, horizon, "start" + suffix);
    taskType.end = model.newIntVar(0, horizon, "end" + suffix);
    taskType.interval = model.newIntervalVar(
        taskType.start, LinearExpr.constant(task.duration), taskType.end, "interval" + suffix);

    List<Integer> key = Arrays.asList(jobID, taskID);
    allTasks.put(key, taskType);
    machineToIntervals.computeIfAbsent(task.machine, (Integer k) -> new ArrayList<>());
    machineToIntervals.get(task.machine).add(taskType.interval);
  }
}
```

For each job and task, the program uses the solver's **NewIntVar** method to create the variables:

- **Start_var**: Start time of the task;
- **End_var**: End time of the task.

The Employee Scheduling problem: step by step

5. Define the constraints:

```
// Create and add disjunctive constraints.
for (int machine : allMachines) {
  List<IntervalVar> list = machineToIntervals.get(machine);
  model.addNoOverlap(list);
}

// Precedences inside a job.
for (int jobID = 0; jobID < allJobs.size(); ++jobID) {
  List<Task> job = allJobs.get(jobID);
  for (int taskID = 0; taskID < job.size() - 1; ++taskID) {
    List<Integer> prevKey = Arrays.asList(jobID, taskID);
    List<Integer> nextKey = Arrays.asList(jobID, taskID + 1);
    model.addGreaterOrEqual(allTasks.get(nextKey).start, allTasks.get(prevKey).end);
  }
}
```

The program uses the solver's **AddNoOverlap** method to create the **no overlap constraints, which prevent tasks for the same machine from overlapping in time**.

Next, the program adds the precedence constraints, **which prevent consecutive tasks for the same job from overlapping in time.**

```
model.Add(
        all_tasks[job, task_id + 1].start >= all_tasks[job, task_id].end)
```

For each job, the line requires the end time of a task to occur before the start time of the next task in the job.

The Employee Scheduling problem: step by step

6. Define the objective:

```
// Makespan objective.
IntVar objVar = model.newIntVar(0, horizon, "makespan");
List<IntVar> ends = new ArrayList<>();
for (int jobID = 0; jobID < allJobs.size(); ++jobID) {
  List<Task> job = allJobs.get(jobID);
  List<Integer> key = Arrays.asList(jobID, job.size() - 1);
  ends.add(allTasks.get(key).end);
}
model.addMaxEquality(objVar, ends);
model.minimize(objVar);
```

7. Invoke the solver:

```
CpSolver solver = new CpSolver();
CpSolverStatus status = solver.solve(model);
```

The expression creates a variable ***obj_var*** whose values is the maximum of the end times for all jobs (that is the **makespan)**

The Employee Scheduling problem: Result

8. Display the results:

```
Optimal Schedule Length: 11
Machine 0: job_0_0   job_1_0
           [0,3]     [3,5]
Machine 1: job_2_0   job_0_1   job_1_2
           [0,4]     [4,6]     [7,11]
Machine 2: job_1_1   job_0_2   job_2_1
           [5,6]     [6,8]     [8,11]
```

Machine 1 might wonder why job_1_2 was scheduled at time 7 instead of time 6. Both are valid solutions, but **the objective is to minimize the makespan**. Moving job_1_2 earlier wouldn't reduce the makespan, **so the two solutions are equal from the solver's perspective**.