Tutorial: Getting started with *Scheduling* in CPLEX for Python

This notebook introduces the basic building blocks of a scheduling model that can be solved using *Constraint Programming Optimizer* (named CP Optimizer in the following) that is included in *CPLEX for Python*.

It is part of **Prescriptive Analytics for Python**

It requires either an installation of CPLEX Optimizers or it can be run on IBM Watson Studio Cloud (Sign up for a free IBM Cloud account and you can start using Watson Studio Cloud right away).

To follow the examples in this section, some knowledge about optimization (math programming or constraint programming) and about modeling optimization problems is necessary. For beginners in optimization, following the online free Decision Optimization tutorials (here and here) might help to get a better understanding of Mathematical Optimization.

Each chapter of this notebook is a self-contained separate lesson.

Chapter 1. Introduction to Scheduling

This chapter describes the basic characteristics of a scheduling program.

Set up the model solving

Solving capabilities are required to solve example models that are given in the following. There are several ways to solve a model:

- Subscribe to the private cloud offer or Decision Optimization on Cloud solve service here.
- Use Watson Studio Cloud that contain a pre-installed version of CPLEX Community Edition
- Use a local solver, a licensed installation of CPLEX Optimization Studio to run the notebook locally.

When using a cloud solving solution, the following attributes must be set with appropriate values:

In [1]:
```
url = None
key = None
```

Scheduling building blocks

Scheduling is the act of creating a schedule, which is a timetable for planned occurrences. Scheduling may also involve allocating resources to activities over time.

A scheduling problem can be viewed as a constraint satisfaction problem or as a constrained optimization problem. Regardless of how it is viewed, a scheduling problem is defined by:

- A set of *time intervals*, to define activities, operations, or tasks to be completed
- A set of *temporal constraints*, to define possible relationships between the start and end times of the intervals
- A set of *specialized constraints*, to specify of the complex relationships on a set of intervals due to the state and finite capacity of resources.

Creation of the model

A scheduling model starts with the creation of the model container, as follows

In [2]:
```
import sys
from docplex.cp.model import *
```

In [3]:
```
mdl0 = CpoModel()
```

This code creates a CP model container that allows the use of constraints that are specific to constraint programming or to scheduling.

Declarations of decision variables

Variable declarations define the type of each variable in the model. For example, to create a variable that equals the amount of material shipped from location *i* to location *j*, a variable named *ship* can be created as follows:

```
ship = [[integer_var(min=0) for j in range(N)] for i in range(N)]
```

This code declares an *array* (list of lists in Python) of non-negative integer decision variables; `ship[i][j]` is the decision variable handling the amount of material shipped from location *i* to location *j*.

For scheduling there are specific additional decision variables, namely:

- *interval* variables
- *sequence* variables.

Activities, *operations* and*tasks* are represented as interval decision variables.

An interval has a *start*, a *end*, a *length*, and a *size*. An interval variable allows for these values to be variable within the model. The start is the lower endpoint of the interval and the end is the upper endpoint of the interval. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

An interval variable may also be optional, and its presence in the solution is represented by a decision variable. If an interval is not present in the solution, this means that any constraints on this interval acts like the interval is “not there”. The exact semantics will depend on the specific constraint.

The following example contains a dictionary of interval decision variables where the sizes of the interval variables are fixed and the keys are 2 dimensional: `itvs = {(h,t) : mdl.interval_var(size = Duration[t]) for h in Houses for t in TaskNames}`

Objective function

The objective function is an expression that has to be optimized. This function consists of variables and data that have been declared earlier in the model. The objective function is introduced by either the *minimize* or the *maximize* function.

For example: `mdl.minimize(mdl.endOf(tasks["moving"]))`
indicates that the end of the interval variable `tasks["moving"]` needs to be minimized.

Constraints

The *constraints* indicate the conditions that are necessary for a feasible solution to the model.

Several types of constraints can be placed on interval variables:

- *precedence* constraints, which ensure that relative positions of intervals in the solution (For example a precedence constraint can model a requirement that an interval a must end before interval b starts, optionally with some minimum delay z);
- *no overlap* constraints, which ensure that positions of intervals in the solution are disjointed in time;
- *span* constraints, which ensure that one interval to cover those intervals in a set of intervals;
- *alternative* constraints, which ensure that exactly one of a set of intervals be present in the solution;
- *synchronize* constraints, which ensure that a set of intervals start and end at the same time as a given interval variable if it is present in the solution;
- *cumulative expression* constraints, which restrict the bounds on the domains of cumulative expressions.

Example

This section provides a completed example model that can be tested.

The problem is a house building problem. There are ten tasks of fixed size, and each of them needs to be assigned a starting time.

The statements for creating the interval variables that represent the tasks are:

In [4]:
```
masonry = mdl0.interval_var(size=35)
carpentry = mdl0.interval_var(size=15)
plumbing = mdl0.interval_var(size=40)
ceiling = mdl0.interval_var(size=15)
roofing = mdl0.interval_var(size=5)
painting = mdl0.interval_var(size=10)
windows = mdl0.interval_var(size=5)
facade = mdl0.interval_var(size=10)
garden = mdl0.interval_var(size=5)
moving = mdl0.interval_var(size=5)
```

Adding the constraints

The constraints in this problem are precedence constraints; some tasks cannot start until other tasks have ended. For example, the *ceilings* must be completed before *painting* can begin.

The set of precedence constraints for this problem can be added to the model with the block:

In [5]:
```
mdl0.add( mdl0.end_before_start(masonry, carpentry) )
mdl0.add( mdl0.end_before_start(masonry, plumbing) )
mdl0.add( mdl0.end_before_start(masonry, ceiling) )
mdl0.add( mdl0.end_before_start(carpentry, roofing) )
mdl0.add( mdl0.end_before_start(ceiling, painting) )
mdl0.add( mdl0.end_before_start(roofing, windows) )
mdl0.add( mdl0.end_before_start(roofing, facade) )
mdl0.add( mdl0.end_before_start(plumbing, facade) )
mdl0.add( mdl0.end_before_start(roofing, garden) )
mdl0.add( mdl0.end_before_start(plumbing, garden) )
mdl0.add( mdl0.end_before_start(windows, moving) )
mdl0.add( mdl0.end_before_start(facade, moving) )
mdl0.add( mdl0.end_before_start(garden, moving) )
mdl0.add( mdl0.end_before_start(painting, moving) )
```

Here, the special constraint *end_before_start()* ensures that one interval variable ends before the other starts. If one of the interval variables is not present, the constraint is automatically satisfied.

Calling the solve

In [6]:
```
# Solve the model
print("\nSolving model....")
msol0 = mdl0.solve(url=url, key=key, TimeLimit=10)
print("done")
```

```
Solving model....
done
```

Displaying the solution

The interval variables and precedence constraints completely describe this simple problem.

Print statements display the solution, after values have been assigned to the start and end of each of the interval variables in the model.

In [7]:
```
var_sol = msol0.get_var_solution(masonry)
print("Masonry : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(carpentry)
print("Carpentry : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(plumbing)
print("Plumbing : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(ceiling)
print("Ceiling : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(roofing)
print("Roofing : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(painting)
print("Painting : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(windows)
```

```
print("Windows : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(facade)
print("Facade : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol0.get_var_solution(moving)
print("Moving : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
```

```
Masonry : 0..35
Carpentry : 35..50
Plumbing : 35..75
Ceiling : 35..50
Roofing : 50..55
Painting : 50..60
Windows : 55..60
Facade : 75..85
Moving : 85..90
```

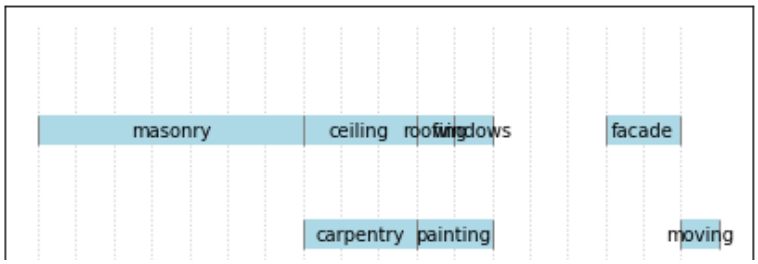
To understand the solution found by *CP Optimizer* to this satisfiability scheduling problem, consider the line: `Masonry : 0..35`

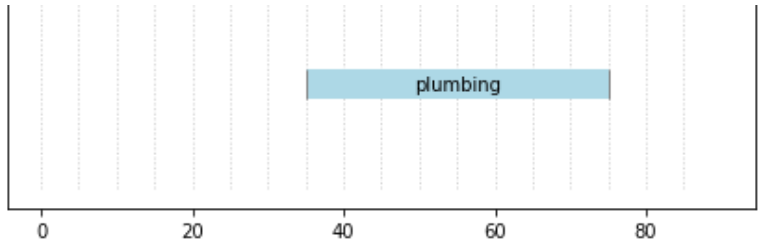
The interval variable representing the masonry task, which has size 35, has been assigned the interval [0,35). Masonry starts at time 0 and ends at the time point 35.

Graphical view of these tasks can be obtained with following additional code:

In [8]:
```
import docplex.cp.utils_visu as visu
import matplotlib.pyplot as plt
%matplotlib inline
#Change the plot size
from pylab import rcParams
rcParams['figure.figsize'] = 15, 3
```

In [9]:
```
wt = msol0.get_var_solution(masonry)
visu.interval(wt, 'lightblue', 'masonry')
wt = msol0.get_var_solution(carpentry)
visu.interval(wt, 'lightblue', 'carpentry')
wt = msol0.get_var_solution(plumbing)
visu.interval(wt, 'lightblue', 'plumbing')
wt = msol0.get_var_solution(ceiling)
visu.interval(wt, 'lightblue', 'ceiling')
wt = msol0.get_var_solution(roofing)
visu.interval(wt, 'lightblue', 'roofing')
wt = msol0.get_var_solution(painting)
visu.interval(wt, 'lightblue', 'painting')
wt = msol0.get_var_solution(windows)
visu.interval(wt, 'lightblue', 'windows')
wt = msol0.get_var_solution(facade)
visu.interval(wt, 'lightblue', 'facade')
wt = msol0.get_var_solution(moving)
visu.interval(wt, 'lightblue', 'moving')
visu.show()
```





Note on interval variables

After a time interval has been assigned a start value (say s) and an end value (say e), the interval is written as [s,e). The time interval does not include the endpoint e. If another interval variable is constrained to be placed after this interval, it can start at the time e.

Chapter 2. Modeling and solving house building with an objective

This chapter presents the same *house building* example in such a manner that minimizes an objective.

It intends to present how to:

- use the *interval variable*,
- use the constraint *endBeforeStart*,
- use the expressions *startOf* and *endOf*.

The objective to minimize is here the cost associated with performing specific tasks before a preferred earliest start date or after a preferred latest end date. Some tasks must necessarily take place before other tasks, and each task has a given duration.

To find a solution to this problem, a three-stage method is used: *describe*, *model*, and *solve*.

Problem to be solved

The problem consists of assigning start dates to tasks in such a way that the resulting schedule satisfies precedence constraints and minimizes a criterion. The criterion for this problem is to minimize the earliness costs associated with starting certain tasks earlier than a given date, and tardiness costs associated with completing certain tasks later than a given date.

For each task in the house building project, the following table shows the duration (measured in days) of the task along with the tasks that must finish before the task can start.

Note: The unit of time represented by an interval variable is not defined. As a result, the size of the masonry task in this problem could be 35 hours or 35 weeks or 35 months.

House construction tasks:

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry

roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

The other information for the problem includes the earliness and tardiness costs associated with some tasks.

House construction task earliness costs:

Task	Preferred earliest start date	Cost per day for starting early
masonry	25	200.0
carpentry	75	300.0
ceiling	75	100.0

House construction task tardiness costs:

Task	Preferred latest end date	Cost per day for ending late
moving	100	400.0

Solving the problem consists of identifying starting dates for the tasks such that the total cost, determined by the earliness and lateness costs, is minimized.

Step 1: Describe the problem

The first step in modeling the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Writing a natural language description of this problem requires to answer these questions:

- What is the known information in this problem ?
- What are the decision variables or unknowns in this problem ?
- What are the constraints on these variables ?
- What is the objective ?

- *What is the known information in this problem ?*

  There are ten house building tasks, each with a given duration. For each task, there is a list of tasks that must be completed before the task can start. Some tasks also have costs associated with an early start date or late end date.

- *What are the decision variables or unknowns in this problem ?*

  The unknowns are the date that each task will start. The cost is determined by the assigned start dates.

- *What are the constraints on these variables ?*

In this case, each constraint specifies that a particular task may not begin until one or more given tasks have been completed.

- *What is the objective ?*

  The objective is to minimize the cost incurred through earliness and tardiness costs.

Step 2: Declare the interval variables

In the model, each task is represented by an interval variables. Each variable represents the unknown information, the scheduled interval for each activity. After the model is executed, the values assigned to these interval variables will represent the solution to the problem.

Declaration of engine

A scheduling model starts with the declaration of the engine as follows:

In [10]:
```
import sys
from docplex.cp.model import *

mdl1 = CpoModel()
```

The declaration of necessary interval variables is done as follows:

In [11]:
```
masonry = mdl1.interval_var(size=35)
carpentry = mdl1.interval_var(size=15)
plumbing = mdl1.interval_var(size=40)
ceiling = mdl1.interval_var(size=15)
roofing = mdl1.interval_var(size=5)
painting = mdl1.interval_var(size=10)
windows = mdl1.interval_var(size=5)
facade = mdl1.interval_var(size=10)
garden = mdl1.interval_var(size=5)
moving = mdl1.interval_var(size=5)
```

Step 3: Add the precedence constraints

In this example, certain tasks can start only after other tasks have been completed. *CP Optimizer* allows to express constraints involving temporal relationships between pairs of interval variables using *precedence constraints*.

Precedence constraints are used to specify when an interval variable must start or end with respect to the start or end time of another interval variable.

The following types of precedence constraints are available; if *a* and *b* denote interval variables, both interval variables are present, and *delay* is a number or integer expression (0 by default), then:

- *end_before_end(a, b, delay)* constrains at least the given delay to elapse between the end of a and the end of b. It imposes the inequality endTime(a) + delay <= endTime(b).
- *end_before_start(a, b, delay)* constrains at least the given delay to elapse between the end of a and the start of b. It imposes the inequality endTime(a) + delay <= startTime(b).

- *end_at_end(a, b, delay)* constrains the given delay to separate the end of a and the end of ab. It imposes the equality endTime(a) + delay == endTime(b).
- *end_at_start(a, b, delay)* constrains the given delay to separate the end of a and the start of b. It imposes the equality endTime(a) + delay == startTime(b).
- *start_before_end(a, b, delay)* constrains at least the given delay to elapse between the start of a and the end of b. It imposes the inequality startTime(a) + delay <= endTime(b).
- *start_before_start(a, b, delay)* constrains at least the given delay to elapse between the start of act1 and the start of act2. It imposes the inequality startTime(a) + delay <= startTime(b).
- *start_at_end(a, b, delay)* constrains the given delay to separate the start of a and the end of b. It imposes the equality startTime(a) + delay == endTime(b).
- *start_at_start(a, b, delay)* constrains the given delay to separate the start of a and the start of b. It imposes the equality startTime(a) + delay == startTime(b).

If either interval *a* or *b* is not present in the solution, the constraint is automatically satisfied, and it is as if the constraint was never imposed.

For our model, precedence constraints can be added with the following code:

In [12]:
```
mdl1.add( mdl1.end_before_start(masonry, carpentry) )
mdl1.add( mdl1.end_before_start(masonry, plumbing) )
mdl1.add( mdl1.end_before_start(masonry, ceiling) )
mdl1.add( mdl1.end_before_start(carpentry, roofing) )
mdl1.add( mdl1.end_before_start(ceiling, painting) )
mdl1.add( mdl1.end_before_start(roofing, windows) )
mdl1.add( mdl1.end_before_start(roofing, facade) )
mdl1.add( mdl1.end_before_start(plumbing, facade) )
mdl1.add( mdl1.end_before_start(roofing, garden) )
mdl1.add( mdl1.end_before_start(plumbing, garden) )
mdl1.add( mdl1.end_before_start(windows, moving) )
mdl1.add( mdl1.end_before_start(facade, moving) )
mdl1.add( mdl1.end_before_start(garden, moving) )
mdl1.add( mdl1.end_before_start(painting, moving) )
```

To model the cost for starting a task earlier than the preferred starting date, the expression *start_of()* can be used. It represents the start of an interval variable as an integer expression.

For each task that has an earliest preferred start date, the number of days before the preferred date it is scheduled to start can be determined using the expression *start_of()*. This expression can be negative if the task starts after the preferred date. Taking the maximum of this value and 0 using *max()* allows to determine how many days early the task is scheduled to start. Weighting this value with the cost per day of starting early determines the cost associated with the task.

The cost for ending a task later than the preferred date is modeled in a similar manner using the expression *endOf()*. The earliness and lateness costs can be summed to determine the total cost.

Step 4: Add the objective

The objective function to be minimized can be written as follows:

In [13]:
```
obj = mdl1.minimize(  400 * mdl1.max([mdl1.end_of(moving) - 100, 0])
                    + 200 * mdl1.max([25 - mdl1.start_of(masonry), 0])
                    + 300 * mdl1.max([75 - mdl1.start_of(carpentry), 0])
                    + 100 * mdl1.max([75 - mdl1.start_of(ceiling), 0]) )
mdl1.add(obj)
```

Solving a problem consists of finding a value for each decision variable so that all constraints are satisfied. It is not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

Step 5: Solve the model and display the solution

In [14]:
```
# Solve the model
print("\nSolving model....")
msol1 = mdl1.solve(url=url, key=key, TimeLimit=20)
print("done")
```

```
Solving model....
done
```

In [15]:
```
print("Cost will be " + str(msol1.get_objective_values()[0]))
```

```
Cost will be 5000
```

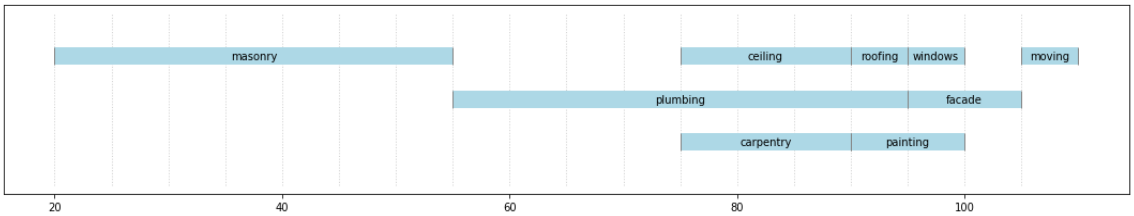
In [16]:
```
var_sol = msol1.get_var_solution(masonry)
print("Masonry : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(carpentry)
print("Carpentry : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(plumbing)
print("Plumbing : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(ceiling)
print("Ceiling : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(roofing)
print("Roofing : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(painting)
print("Painting : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(windows)
print("Windows : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(facade)
print("Facade : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
var_sol = msol1.get_var_solution(moving)
print("Moving : {}..{}".format(var_sol.get_start(), var_sol.get_end()))
```

```
Masonry : 20..55
Carpentry : 75..90
Plumbing : 55..95
Ceiling : 75..90
Roofing : 90..95
Painting : 90..100
Windows : 95..100
Facade : 95..105
Moving : 105..110
```

Graphical display of the same result is available with:

In [17]:
```
import docplex.cp.utils_visu as visu
import matplotlib.pyplot as plt
%matplotlib inline
#Change the plot size
from pylab import rcParams
rcParams['figure.figsize'] = 15, 3
```

In [18]:
```
wt = msol1.get_var_solution(masonry)
visu.interval(wt, 'lightblue', 'masonry')
wt = msol1.get_var_solution(carpentry)
visu.interval(wt, 'lightblue', 'carpentry')
wt = msol1.get_var_solution(plumbing)
visu.interval(wt, 'lightblue', 'plumbing')
wt = msol1.get_var_solution(ceiling)
visu.interval(wt, 'lightblue', 'ceiling')
wt = msol1.get_var_solution(roofing)
visu.interval(wt, 'lightblue', 'roofing')
wt = msol1.get_var_solution(painting)
visu.interval(wt, 'lightblue', 'painting')
wt = msol1.get_var_solution(windows)
visu.interval(wt, 'lightblue', 'windows')
wt = msol1.get_var_solution(facade)
visu.interval(wt, 'lightblue', 'facade')
wt = msol1.get_var_solution(moving)
visu.interval(wt, 'lightblue', 'moving')
visu.show()
```



The overall cost is 5000 and moving will be completed by day 110.

Chapter 3. Adding workers and transition times to the house building problem

This chapter introduces workers and transition times to the house building problem described in the previous chapters. It allows to learn the following concepts:

- use the interval variable *sequence*;
- use the constraints *span* and *no_overlap*;
- use the expression *length_of*.

The problem to be solved is the scheduling of tasks involved in building *multiple* houses in a manner that minimizes the costs associated with completing each house after a given due date and with the length of time it takes to build each house. Some tasks must necessarily take place before other tasks, and each task has a predefined duration. Each house has an earliest starting date. Moreover, there are two workers, each of whom must perform a given subset of the necessary tasks, and there is a transition time associated with a worker transferring from one house to another house. A task, once started, cannot be interrupted.

The objective is to minimize the cost, which is composed of tardiness costs for certain tasks as well as a cost associated with the length of time it takes to complete each house.

Problem to be solved

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies

temporal constraints and minimizes a criterion. The criterion for this problem is to minimize the tardiness costs associated with completing each house later than its specified due date and the cost associated with the length of time it takes to complete each house.

For each type of task, the following table shows the duration of the task in days along with the tasks that must be finished before the task can start. In addition, each type of task must be performed by a specific worker, Jim or Joe. A worker can only work on one task at a time. A task, once started, may not be interrupted. The time required to transfer from one house to another house is determined by a function based on the location of the two houses.

The following table indicates these details for each task:

Task	Duration	Worker	Preceding tasks
masonry	35	Joe	
carpentry	15	Joe	masonry
plumbing	40	Jim	masonry
ceiling	15	Jim	masonry
roofing	5	Joe	carpentry
painting	10	Jim	ceiling
windows	5	Jim	roofing
facade	10	Joe	roofing, plumbing
garden	5	Joe	roofing, plumbing
moving	5	Jim	windows, facade,garden, painting

For each of the five houses that must be built, there is an earliest starting date, a due date and a cost per day of completing the house later than the preferred due date.

The house construction tardiness costs is indicated in the following table:

House	Earliest start date	Preferred latest end date	Cost per day for ending late
0	0	120	100.0
1	0	212	100.0
2	151	304	100.0
3	59	181	200.0
4	243	425	100.0

Solving the problem consists of determining starting dates for the tasks such that the cost, where the cost is determined by the lateness costs and length costs, is minimized.

Step 1: Describe the problem

- What is the known information in this problem ?

  There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given duration, or size. Each house also has a given earliest starting date. For each task, there is a list of tasks that must be completed before the task can start. Each task must be performed by a given worker, and there is a transition time associated with a worker transferring

from one house to another house. There are costs associated with completing eachhouse after its preferred due date and with the length of time it takes to complete each house.

- What are the decision variables or unknowns in this problem ?

  The unknowns are the start and end dates of the interval variables associated with the tasks. Once fixed, these interval variables also determine the cost of the solution. For some of the interval variables, there is a fixed minimum start date.

- What are the constraints on these variables ?

  There are constraints that specify a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that a worker can be assigned to only one task at a time and that it takes time for a worker to travel from one house to the other.

- What is the objective ?

  The objective is to minimize the cost incurred through tardiness and length costs.

Step2: Prepare data

First coding step is to prepare model data:

In [19]:
```
NbHouses = 5

WorkerNames = ["Joe", "Jim"]

TaskNames = ["masonry", "carpentry", "plumbing",
             "ceiling", "roofing", "painting",
             "windows", "facade", "garden", "moving"]

Duration =  [35, 15, 40, 15, 5, 10, 5, 10, 5, 5]

Worker = {"masonry"  : "Joe" ,
          "carpentry": "Joe" ,
          "plumbing" : "Jim" ,
          "ceiling"  : "Jim" ,
          "roofing"  : "Joe" ,
          "painting" : "Jim" ,
          "windows"  : "Jim" ,
          "facade"   : "Joe" ,
          "garden"   : "Joe" ,
          "moving"   : "Jim"}

ReleaseDate = [  0,     0,   151,    59,   243]
DueDate     = [120,   212,   304,   181,   425]
Weight      = [100.0, 100.0, 100.0, 200.0, 100.0]

Precedences = [("masonry", "carpentry"),("masonry", "plumbing"),
               ("masonry", "ceiling"), ("carpentry", "roofing"),
               ("ceiling", "painting"), ("roofing", "windows"),
               ("roofing", "facade"), ("plumbing", "facade"),
               ("roofing", "garden"), ("plumbing", "garden"),
               ("windows", "moving"), ("facade", "moving"),
               ("garden", "moving"), ("painting", "moving")]

Houses = range(NbHouses)
```

One part of the objective is based on the time it takes to build a house. To model this, one interval variable is used for each house, and is later constrained to span the tasks associated with the given house. As each house has an earliest starting date, and each house interval variable is declared to have a start date no earlier than that release date. The ending date of the task is not constrained, so the upper value of the range for the variable is maxint.

Step 3: Create the house interval variables

In [20]:
```
import sys
from docplex.cp.model import *

mdl2 = CpoModel()
```

In [21]:
```
houses = [mdl2.interval_var(start=(ReleaseDate[i], INTERVAL_MAX), name="hous
e"+str(i)) for i in Houses]
```

Step 4: Create the task interval variables

Each house has a list of tasks that must be scheduled. The duration, or size, of each task t is Duration[t]. This information allows to build the matrix *itvs* of interval variables.

In [22]:
```
TaskNames_ids = {}
itvs = {}
for h in Houses:
    for i,t in enumerate(TaskNames):
        _name = str(h)+"_"+str(t)
        itvs[(h,t)] = mdl2.interval_var(size=Duration[i], name=_name)
        TaskNames_ids[_name] = i
```

Step 5: Add the precedence constraints

The tasks of the house building project have precedence constraints that are added to the model.

In [23]:
```
for h in Houses:
    for p in Precedences:
        mdl2.add(mdl2.end_before_start(itvs[(h,p[0])], itvs[(h,p[1])]) )
```

To model the cost associated with the length of time it takes to build a single house, the interval variable associated with the house is constrained to start at the start of the first task of the house and end at the end of the last task. This interval variable must span the tasks.

Step 6: Add the span constraints

The constraint *span* allows to specify that one interval variable must exactly cover a set of interval variables. In other words, the spanning interval is present in the solution if and only if at least one of the spanned interval variables is present and, in this case, the spanning interval variable starts at the start of the interval variable scheduled earliest in the set and ends at the end of the interval variable scheduled latest in the set.

For house *h*, the interval variable *houses[h]* is constrained to cover the interval variables in *itvs* that are associated with the tasks of the given house.

In [24]:
```
for h in Houses:
    mdl2.add( mdl2.span(houses[h], [itvs[(h,t)] for t in TaskNames] ) )
```

Step 7: Create the transition times

Transition times can be modeled using tuples with three elements. The first element is the interval variable type of one task, the second is the interval variable type of the other task and the third element of the tuple is the transition time from the first to the second. An integer interval variable type can be associated with each interval variable.

Given an interval variable *a1* that precedes (not necessarily directly) an interval variable *a2* in a sequence of non-overlapping interval variables, the transition time between *a1* and *a2* is an amount of time that must elapse between the end of *a1* and the beginning of *a2*.

In [25]:
```
transitionTimes = transition_matrix(NbHouses)
for i in Houses:
    for j in Houses:
        transitionTimes.set_value(i, j, int(abs(i - j)))
```

Each of the tasks requires a particular worker. As a worker can perform only one task at a time, it is necessary to know all of the tasks that a worker must perform and then constrain that these intervals not overlap and respect the transition times. A sequence variable represents the order in which the workers perform the tasks.

Note that the sequence variable does not force the tasks to not overlap or the order of tasks. In a later step, a constraint is created that enforces these relations on the sequence of interval variables.

Step 8: Create the sequence variables

Using the decision variable type sequence, variable can be created to represent a sequence of interval variables. The sequence can contain a subset of the variables or be empty. In a solution, the sequence will represent a total order over all the intervals in the set that are present in the solution. The assigned order of interval variables in the sequence does not necessarily determine their relative positions in time in the schedule. The sequence variable takes an array of interval variables as well as the transition types for each of those variables. Interval sequence variables are created for Jim and Joe, using the arrays of their tasks and the task locations.

In [26]:
```
workers = {w : mdl2.sequence_var([ itvs[(h,t)] for h in Houses for t in Task
Names if Worker[t]==w ],
                                 types=[h for h in Houses for t in TaskNames
if Worker[t]==w ], name="workers_"+w)
           for w in WorkerNames}
```

Step 9: Add the no overlap constraint

Now that the sequence variables have been created, each sequence must be constrained such that the interval variables do not overlap in the solution, that the transition times are respected, and that the

sequence represents the relations of the interval variables in time.

The constraint *no_overlap* allows to constrain an interval sequence variable to define a chain of non-overlapping intervals that are present in the solution. If a set of transition tuples is specified, it defines the minimal time that must elapse between two intervals in the chain. Note that intervals which are not present in the solution are automatically removed from the sequence. One no overlap constraint is created for the sequence interval variable for each worker.

In [27]:
```
for w in WorkerNames:
    mdl2.add( mdl2.no_overlap(workers[w], transitionTimes) )
```

The cost for building a house is the sum of the tardiness cost and the number of days it takes from start to finish building the house. To model the cost associated with a task being completed later than its preferred latest end date, the expression *endOf()* can be used to determine the end date of the house interval variable. To model the cost of the length of time it takes to build the house, the expression *lengthOf()* can be used, which returns an expression representing the length of an interval variable.

Step 10: Add the objective

The objective of this problem is to minimize the cost as represented by the cost expression.

In [28]:
```
# create the obj and add it.
mdl2.add(
    mdl2.minimize(
        mdl2.sum(Weight[h] * mdl2.max([0, mdl2.end_of(houses[h])-DueDate[h
]]) + mdl2.length_of(houses[h]) for h in Houses)
    )
)
```

Step 11: Solve the model

The search for an optimal solution in this problem can potentiality take a long time. A fail limit can be placed on the solve process to limit the search process. The search stops when the fail limit is reached, even if optimality of the current best solution is not guaranteed. The code for limiting the solve process is provided below:

In [29]:
```
# Solve the model
print("\nSolving model....")
msol2 = mdl2.solve(url=url, key=key, FailLimit=30000)
print("done")
```

```
Solving model....
done
```

In [30]:
```
print("Cost will be " + str(msol2.get_objective_values()[0]))
```

```
Cost will be 18741
```

In [31]:
```
# Viewing the results of sequencing problems in a Gantt chart
# (double click on the gantt to see details)
import docplex.cp.utils_visu as visu
import matplotlib.pyplot as plt
```

```
%matplotlib inline
#Change the plot size
from pylab import rcParams
rcParams['figure.figsize'] = 15, 3

def showsequence(msol, s, setup, tp):
    seq = msol.get_var_solution(s)
    visu.sequence(name=s.get_name())
    vs = seq.get_value()
    for v in vs:
        nm = v.get_name()
        visu.interval(v, tp[TaskNames_ids[nm]], nm)
    for i in range(len(vs) - 1):
        end = vs[i].get_end()
        tp1 = tp[TaskNames_ids[vs[i].get_name()]]
        tp2 = tp[TaskNames_ids[vs[i + 1].get_name()]]
        visu.transition(end, end + setup.get_value(tp1, tp2))

visu.timeline("Solution for SchedSetup")
for w in WorkerNames:
    types=[h for h in Houses for t in TaskNames if Worker[t]==w]
    showsequence(msol2, workers[w], transitionTimes, types)
visu.show()
```

Chapter 4. Adding calendars to the house building problem

This chapter introduces calendars into the house building problem, a problem of scheduling the tasks involved in building multiple houses in such a manner that minimizes the overall completion date of the houses.

There are two workers, each of whom must perform a given subset of the necessary tasks. Each worker has a calendar detailing on which days he does not work, such as weekends and holidays. On a worker's day off, he does no work on his tasks, and his tasks may not be scheduled to start or end on these days. Tasks that are in process by the worker are suspended during his days off.

Following concepts are demonstrated:

- use of the *step functions*,
- use an alternative version of the constraint *no_overlap*,
- use *intensity* expression,
- use the constraints *forbid_start* and *forbid_end*,
- use the *length* and *size* of an interval variable.

Problem to be solved

The problem consists of assigning start dates to a set of tasks in such a way that the schedule satisfies temporal constraints and minimizes a criterion. The criterion for this problem is to minimize the overall completion date. For each task type in the house building project, the following table shows the size of the task in days along with the tasks that must be finished before the task can start. In addition, each type of task can be performed by a given one of the two workers, Jim and Joe. A worker can only work on one task at a time. Once started, Problem to be solveda task may be suspended during a worker's days off, but may not be interrupted by another task.

House construction tasks are detailed in the folowing table:

Task	Duration	Worker	Preceding tasks
masonry	35	Joe	
carpentry	15	Joe	masonry
plumbing	40	Jim	masonry
ceiling	15	Jim	masonry
roofing	5	Joe	carpentry
painting	10	Jim	ceiling
windows	5	Jim	roofing
facade	10	Joe	roofing, plumbing
garden	5	Joe	roofing, plumbing
moving	5	Jim	windows, facade, garden, painting

Solving the problem consists of determining starting dates for the tasks such that the overall completion date is minimized.

Step 1: Describe the problem

The first step in modeling the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

- What is the known information in this problem ?

  There are five houses to be built by two workers. For each house, there are ten house building tasks, each with a given size. For each task, there is a list of tasks that must be completed before the task can start. Each task must be performed by a given worker, and each worker has a calendar listing his days off.

- What are the decision variables or unknowns in this problem ?

  The unknowns are the start and end times of tasks which also determine the overall completion time. The actual length of a task depends on its position in time and on the calendar of the associated worker.

- What are the constraints on these variables ?

  There are constraints that specify that a particular task may not begin until one or more given tasks have been completed. In addition, there are constraints that specify that a worker can be assigned to only one task at a time. A task cannot start or end during the associated worker's days off.

- What is the objective ?

  The objective is to minimize the overall completion date.

Step 2: Prepare data

A scheduling model starts with the declaration of the engine as follows:

In [32]:
```
import sys
from docplex.cp.model import *

mdl3 = CpoModel()

NbHouses = 5;

WorkerNames = ["Joe", "Jim" ]

TaskNames = ["masonry","carpentry","plumbing","ceiling","roofing","painting"
,"windows","facade","garden","moving"]

Duration =  [35,15,40,15,5,10,5,10,5,5]

Worker = {"masonry":"Joe","carpentry":"Joe","plumbing":"Jim","ceiling":"Jim"
,
          "roofing":"Joe","painting":"Jim","windows":"Jim","facade":"Joe",
          "garden":"Joe","moving":"Jim"}


Precedences = {  ("masonry","carpentry"),("masonry","plumbing"),
                 ("masonry","ceiling"),("carpentry","roofing"),
                 ("ceiling","painting"),("roofing","windows"),
                 ("roofing","facade"),("plumbing","facade"),
                 ("roofing","garden"),("plumbing","garden"),
                 ("windows","moving"),("facade","moving"),
                 ("garden","moving"),("painting","moving")  }
```

Step 3: Add the intensity step functions

To model the availability of a worker with respect to his days off, a step function is created to represents his intensity over time. This function has a range of [0..100], where the value 0 represents that the worker is not available and the value 100 represents that the worker is available with regard to his calendar.

Step functions are created by the method *step_function()*. Each interval [x1, x2) on which the function has the same value is called a step. When two consecutive steps of the function have the same value, these steps are merged so that the function is always represented with the minimal number of steps.

For each worker, a sorted tupleset is created. At each point in time where the worker's availability changes, a tuple is created. The tuple has two elements; the first element is an integer value that represents the worker's availability (0 for on a break, 100 for fully available to work, 50 for a half-day), and the other element represents the date at which the availability changes to this value. This tupleset, sorted by date, is then used to create a step function to represent the worker's intensity over time. The value of the function after the final step is set to 100.

In [33]:
```
Breaks ={
  "Joe" : [
      (5,14),(19,21),(26,28),(33,35),(40,42),(47,49),(54,56),(61,63),
      (68,70),(75,77),(82,84),(89,91),(96,98),(103,105),(110,112),(117,119),
      (124,133),(138,140),(145,147),(152,154),(159,161),(166,168),(173,175),
      (180,182),(187,189),(194,196),(201,203),(208,210),(215,238),(243,245),(
250,252),
      (257,259),(264,266),(271,273),(278,280),(285,287),(292,294),(299,301),
      (306,308),(313,315),(320,322),(327,329),(334,336),(341,343),(348,350),
      (355,357),(362,364),(369,378),(383,385),(390,392),(397,399),(404,406),(
411,413),
      (418,420),(425,427),(432,434),(439,441),(446,448),(453,455),(460,462),(
467,469),
      (474,476),(481,483),(488,490),(495,504),(509,511),(516,518),(523,525),(
530,532),
      (537,539),(544,546),(551,553),(558,560),(565,567),(572,574),(579,602),(
607,609),
      (614,616),(621,623),(628,630),(635,637),(642,644),(649,651),(656,658),(
663,665),
      (670,672),(677,679),(684,686),(691,693),(698,700),(705,707),(712,714),
      (719,721),(726,728)
  ],
  "Jim" : [
      (5,7),(12,14),(19,21),(26,42),(47,49),(54,56),(61,63),(68,70),(75,77),
      (82,84),(89,91),(96,98),(103,105),(110,112),(117,119),(124,126),(131,13
3),
      (138,140),(145,147),(152,154),(159,161),(166,168),(173,175),(180,182),(
187,189),
      (194,196),(201,225),(229,231),(236,238),(243,245),(250,252),(257,259),
      (264,266),(271,273),(278,280),(285,287),(292,294),(299,301),(306,315),
      (320,322),(327,329),(334,336),(341,343),(348,350),(355,357),(362,364),(
369,371),
      (376,378),(383,385),(390,392),(397,413),(418,420),(425,427),(432,434),(
439,441),
      (446,448),(453,455),(460,462),(467,469),(474,476),(481,483),(488,490),(
495,497),
      (502,504),(509,511),(516,518),(523,525),(530,532),(537,539),(544,546),
      (551,553),(558,560),(565,581),(586,588),(593,595),(600,602),(607,609),
      (614,616),(621,623),(628,630),(635,637),(642,644),(649,651),(656,658),
      (663,665),(670,672),(677,679),(684,686),(691,693),(698,700),(705,707),
      (712,714),(719,721),(726,728)]
  }
```

In [34]:
```
from collections import namedtuple
Break = namedtuple('Break', ['start', 'end'])
```

In [35]:
```
Calendar = {}
mymax = max(max(v for k,v in Breaks[w]) for w in WorkerNames)
for w in WorkerNames:
    step = CpoStepFunction()
    step.set_value(0, mymax, 100)
    for b in Breaks[w]:
        t = Break(*b)
        step.set_value(t.start, t.end, 0)
    Calendar[w] = step
```

This intensity function is used in creating the task variables for the workers. The intensity step function of the appropriate worker is passed to the creation of each interval variable. The size of the interval variable is the time spent at the house to process the task, not including the worker's day off. The length is the difference between the start and the end of the interval.