Lab: Scheduling House Building Tasks

November 10th, 2018

Author: John Chaves john.chaves@us.ibm.com
Review: Fabio Lima fabiofl@br.ibm.com

Table of contents

Contents

Overview

This Lab exercise will guide you on how to set up Decision Optimization engines, and build a constraint programming model in a Jupyter Notebook to efficiently assign construction tasks to workers of different skill levels when you build houses in different locations. You will learn how to:

- How to download the libraries
- Setup the Optimization engine
- Setup a prescriptive model and solve it.

Pre-requisites, access, and files

- Working knowledge of DSX Local and Jupyter notebooks.
- To complete this lab, you will need access to a DSX Local cluster.
- You will also need to download and unzip this GitHub repository: https://github.ibm.com/john-chaves/FastStart_DDLabs

The Business Problem

This is a problem of building five houses in different locations; the masonry, roofing, painting, etc. must be scheduled. Some tasks must necessarily take place before others and these requirements are expressed through precedence constraints.

There are three workers, and each worker has a given skill level for each task. Each task requires one worker; the worker assigned must have a non-null skill level for the task. A worker can be assigned to only one task at a time.

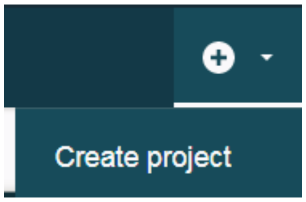
Each house has a completion deadline.

The objective is to maximize the skill levels of the workers assigned to the tasks.

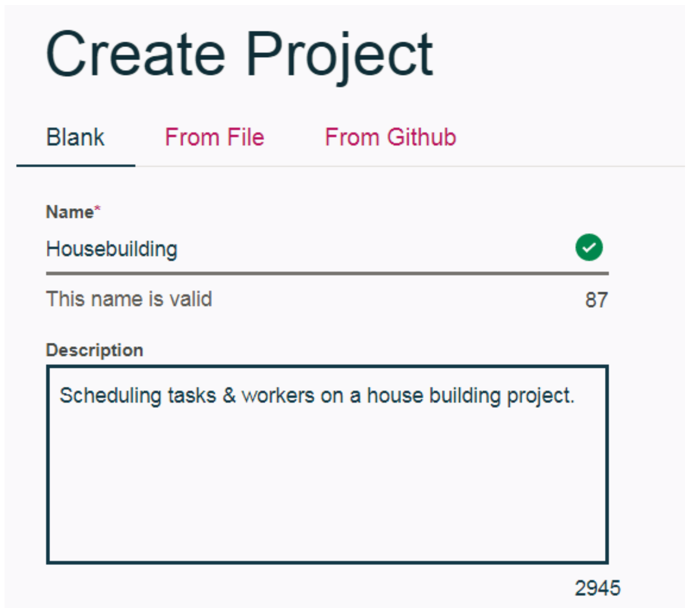
Part 1: Set up a Project

In this section we will set up a DSX Local Project. The Project is a high level container where all assets are stored or referenced.

1. Login DSX Local
2. Once in the *Community* page, click on the little plus (+) sign on the top right of the screen and select *Create Project*. →



3. Enter a project name (i.e. "Housebuilding") and a brief description. Click *Create*.



Create Project

Blank From File From Github

Name*

Housebuilding

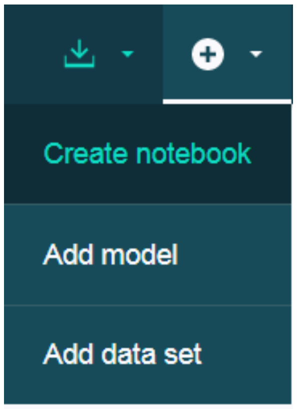
This name is valid 87

Description

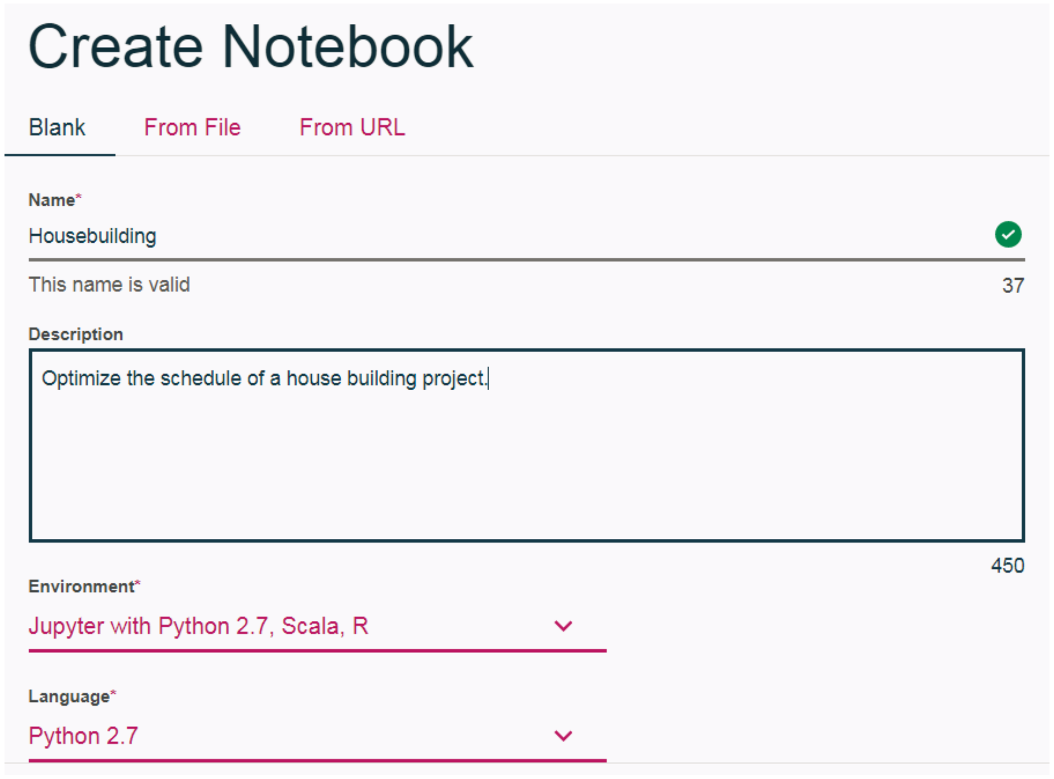
Scheduling tasks & workers on a house building project.

2945

4. On the Project Dashboard, click again on the Plus sign at the top, right side of the screen and create a new Notebook.



5. Enter a Name and (optional) brief description. Leave the default settings for Environment and Language. Click *Create*


Create Notebook

Name*
Housebuilding
This name is valid 37

Description
Optimize the schedule of a house building project.
450

Environment*
Jupyter with Python 2.7, Scala, R

Language*
Python 2.7

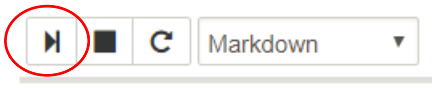
Part 2: Setting up the engine and Model

Each box below represents a Notebook cell. Copy and paste its contents in the cells in the prescribed sequence.

1. Add a brief heading text. Set the cell type as Markdown

```
# Creating and running an Optimization Model
```

Run the cell using the toolbar to see the results



2. Download the CPLEX library

   DOcplex is the Decision Optimization CPLEX Modeling library for Python. This library includes two modules, Mathematical Programming Modeling (DOcplex.MP) and Constraint Programming Modeling (DOcplex.CP).

   Run the following cell to install the library.

```
!pip install docplex
```

Import the CP package

```
from docplex.mp.model import Model, Context
```

To access the DOClexCloud solve service, perform the following steps:

- Subscribe to the Decision Optimization on Cloud (DOcplexcloud) service.
- Get the service base URL and personal API key.
- Enter the URL and API key in the cell below, enclosed in quotation marks (""), and run the cell.

```
# Instanciando o contexto, onde definimos se executaremos localment
e ou na nuvem
url = 'https://api-oaas.docloud.ibmcloud.com/job_manager/rest/v1/'
key = 'api_5b04b8e7-0561-4f4b-ae87-aded89a9bb63'

context = Context.make_default_context()
context.solver.docloud.url = url
context.solver.docloud.key = key
context.solver.agent = 'docloud'

# Instanciando o modelo
# É importante darmos um nome para podermos identificar no otimizad
or qual job está em execução!
mdl = Model(name="otimizador_os",context=context)
```

If you don’t have a DOCplexCloud account, leave it blank. The notebook will solve it using Cplex Community Edition.

For solution display purposes, ensure latest version of matplotlib is available:

```
# Make sure the indentation is exactly as shown in the guide.
try:
   import matplotlib
   if matplotlib.__version__ < "1.4.3":
        !pip install --upgrade matplotlib
except:
   !pip install --user matplotlib
```

Now, we need to import all required modeling functions that are provided by the docplex.cp package:

```
from docplex.cp.model import *
from sys import stdout
from collections import namedtuple
```

Part 3: Model the data

Planning contains the number of houses and the max amount of periods (days) for our schedule.
➔ TIP: Add a Markdown cell with this section name, i.e. “Model the Data”

```
NB_HOUSES = 5
MAX_AMOUNT_OF_PERIODS = 318
HOUSES = range(1, NB_HOUSES + 1)
# All tasks must start and end between 0 and the max amount of
periods.
period_domain = (0, MAX_AMOUNT_OF_PERIODS)
```

The following table shows the duration of the task, in days, along with the tasks that must be finished before the task can start. A worker can only work on one task at a time; each task, once started, may not be interrupted.

Task	Duration (Days)	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

Tasks with duration

```
Task = (namedtuple("Task", ["name", "duration"]))
TASKS = {Task("masonry",   35),
         Task("carpentry", 15),
         Task("plumbing",  40),
         Task("ceiling",   15),
         Task("roofing",    5),
         Task("painting",  10),
         Task("windows",    5),
         Task("facade",    10),
         Task("garden",     5),
         Task("moving",     5),
        }
```

Tasks and precedences

```
TaskPrecedence = (namedtuple("TaskPrecedence", ["beforeTask",
"afterTask"]))
TASK_PRECEDENCES = {TaskPrecedence("masonry",   "carpentry"),
                    TaskPrecedence("masonry",   "plumbing"),
                    TaskPrecedence("masonry",   "ceiling"),
                    TaskPrecedence("carpentry", "roofing"),
                    TaskPrecedence("ceiling",   "painting"),
                    TaskPrecedence("roofing",   "windows"),
                    TaskPrecedence("roofing",   "facade"),
                    TaskPrecedence("plumbing",  "facade"),
                    TaskPrecedence("roofing",   "garden"),
                    TaskPrecedence("plumbing",  "garden"),
                    TaskPrecedence("windows",   "moving"),
                    TaskPrecedence("facade",    "moving"),
                    TaskPrecedence("garden",    "moving"),
                    TaskPrecedence("painting",  "moving"),
                   }
```

There are three workers with varying skill levels in regard to the ten construction tasks. If a worker has a skill level of zero for a task, he may not be assigned to the task.

Task	*Joe*	*Jack*	*Jim*
masonry	9	5	0
carpentry	7	0	5
plumbing	0	7	0
ceiling	5	8	0
roofing	6	7	0
painting	0	9	6
windows	8	0	5
façade	5	5	0
garden	5	5	9
moving	6	0	8

Adding worker's names and their skills

```
WORKERS = {"Joe", "Jack", "Jim"}
Skill = (namedtuple("Skill", ["worker", "task", "level"]))
SKILLS = {Skill("Joe",  "masonry",   9),
          Skill("Joe",  "carpentry", 7),
          Skill("Joe",  "ceiling",   5),
          Skill("Joe",  "roofing",   6),
          Skill("Joe",  "windows",   8),
          Skill("Joe",  "facade",    5),
          Skill("Joe",  "garden",    5),
          Skill("Joe",  "moving",    6),
          Skill("Jack", "masonry",   5),
          Skill("Jack", "plumbing",  7),
          Skill("Jack", "ceiling",   8),
          Skill("Jack", "roofing",   7),
          Skill("Jack", "painting",  9),
          Skill("Jack", "facade",    5),
          Skill("Jack", "garden",    5),
          Skill("Jim",  "carpentry", 5),
          Skill("Jim",  "painting",  6),
          Skill("Jim",  "windows",   5),
          Skill("Jim",  "garden",    9),
          Skill("Jim",  "moving",    8)
          }
```

Add some utility functions

```
# find_tasks: returns the task it refers to in the TASKS vector.
def find_tasks(name):
    return next(t for t in TASKS if t.name == name)

# find_skills: returns the skill it refers to in the SKILLS vector.
def find_skills(worker, task):
    return next(s for s in SKILLS if (s.worker == worker) and (s.task
== task))

# find_max_level_skill: returns the tuple "skill" where a worker's
level is the maximum for a given task.
def find_max_level_skill(task):
    st = [s for s in SKILLS if s.task == task]
    return next(sk for sk in st if sk.level == max([s.level for s in
st]))
```

Part 4: Set up the prescriptive model

1. Create the model container
The model is represented by a Python object that is filled with the available model elements (variables, constraints, objective function, etc.). The first thing to do is to create the object:

```
# Create the model container
mdl = CpoModel(name="HouseBuilding")
```

2. For each `house`, an interval variable is created for each task.
This interval must start and end inside the `period_domain` and its duration is set as the value stated in TASKS definition.

```
tasks = {}   # dict of interval variable for each house and task
for house in HOUSES:
    for task in TASKS:
        tasks[(house, task)] = interval_var(start=period_domain,
                                            end=period_domain,
                                            size=task.duration,
                                            name="house {} task
{}".format(house, task))
```

3. For each house, an **optional** interval variable is created for each skill.
With skill being a tuple (worker, task, level), this means that for each house, an **optional** interval variable is created for each couple worker-task such that the skill level of this worker for this task is > 0.

4. The "**set_optional()**" specifier allows a choice between different variables, thus between different couples house-skill. This means that the engine decides if the interval will be present or absent in the solution.

```
wtasks = {}  # dict of interval variable for each house and skill
for house in HOUSES:
    for skill in SKILLS:
        iv = interval_var(name='H' + str(house) + '-' + skill.task +
'(' + skill.worker + ')')
        iv.set_optional()
        wtasks[(house, skill)] = iv
```

5. The tasks have precedence constraints that are added to the model

```
for h in HOUSES:
    for p in TASK_PRECEDENCES:
        mdl.add(end_before_start(tasks[(h, find_tasks(p.beforeTask))],
tasks[(h, find_tasks(p.afterTask))]))
```

6. To constrain the solution so that exactly one of the interval variables wtasks associated with a given task of a given house is to be present in the solution, an "**alternative**" constraint is used.

```
for h in HOUSES:
    for t in TASKS:
        mdl.add(alternative(tasks[(h, t)], [wtasks[(h, s)] for s in
SKILLS if (s.task == t.name)], 1))
```

7. To add the constraints that a given worker can be assigned only one task at a given moment in time, a **noOverlap** constraint is used.

```
for w in WORKERS:
    mdl.add(no_overlap([wtasks[(h, s)] for h in HOUSES for s in SKILLS
if s.worker == w]))
```

8. Express the objective

The objective of this problem is to maximize the skill level used for all the tasks.

```
obj = sum([s.level * presence_of(wtasks[(h, s)]) for s in SKILLS for h
in HOUSES])
mdl.add(maximize(obj))
```

Part 5: Solving and Viewing the Solution

1. Let's execute the solve

```
# Solve the model
print("\nSolving model....")
msol = mdl.solve(url=SVC_URL, key=SVC_KEY, TimeLimit=10)
```

2. View the solution

```
print("Solve status: " + msol.get_solve_status())
if msol.is_solution():
    stdout.write("Solve time: " + str(msol.get_solve_time()) + "\n")
    # Sort tasks in increasing begin order
    ltasks = []
    for hs in HOUSES:
        for tsk in TASKS:
            (beg, end, dur) = msol[tasks[(hs, tsk)]]
            ltasks.append((hs, tsk, beg, end, dur))
    ltasks = sorted(ltasks, key = lambda x : x[2])
    # Print solution
    print("\nList of tasks in increasing start order:")
    for tsk in ltasks:
        print("From " + str(tsk[2]) + " to " + str(tsk[3]) + ", " +
tsk[1].name + " in house " + str(tsk[0]))
else:
    stdout.write("No solution found\n")
```

CONGRATULATIONS!
You have successfully completed the Lab1 Scheduling Housebuilding Tasks.

Bonus Section – Visualizing the Results

Let’s import some graphical tools and turn the pop up off.

```
POP_UP_GRAPHIC=False

import docplex.cp.utils_visu as visu
import matplotlib.pyplot as plt
if not POP_UP_GRAPHIC:
    %matplotlib inline
#Change the plot size
from pylab import rcParams
rcParams['figure.figsize'] = 15, 3
```

Now, we can draw the solution

```
# Get just the first letter of each Task for easy display
def compact_name(name,n): return name[:n]

if msol and visu.is_visu_enabled():
    workers_colors = {}
    workers_colors["Joe"] = 'lightblue'
    workers_colors["Jack"] = 'violet'
    workers_colors["Jim"] = 'lightgreen'
    visu.timeline('Solution per houses', 0, MAX_AMOUNT_OF_PERIODS)
    for h in HOUSES:
        visu.sequence(name="house " + str(h))
        for s in SKILLS:
            wt = msol.get_var_solution(wtasks[(h,s)])
            if wt.is_present():
                color = workers_colors[s.worker]
                wtname = compact_name(s.task,2)
                visu.interval(wt, color, wtname)
    visu.show()
```

Let’s do the same for the workers

```
def compact_house_task(name):
    loc, task = name[1:].split('-', 1)
    return task[0].upper() + loc
if msol and visu.is_visu_enabled():
    visu.timeline('Solution per workers', 0, MAX_AMOUNT_OF_PERIODS)
    for w in WORKERS:
        visu.sequence(name=w)
        for h in HOUSES:
            for s in SKILLS:
                if s.worker == w:
                    wt = msol.get_var_solution(wtasks[(h,s)])
                    if wt.is_present():
                        ml = find_max_level_skill(s.task).level
                        if s.level == ml:
                            color = 'lightgreen'
                        else:
                            color = 'salmon'
                        wtname = compact_house_task(wt.get_name())
                        visu.interval(wt, color, wtname)
    visu.show()
```