Discrete Optimization Specialization: Assignment 10
Banquet Preparation

1 Problem Statement

In order to celebrate Nüwa's 10000th birthday, the Great Turtle is also preparing a huge banquet, which includes all her favorite dishes. The Head Chef in the Undersea Kingdom needs to schedule the preparation of the banquet in the least time to ensure it is all done before Nüwa is too tired and needs to retire to bed.

The Head Chef has a set of cooks to use, and each dish is broken into some different steps that all have to be done in a certain order. Each step needs to be performed by one of the cooks, who can only be doing one thing at a time.

2 Data Format Specification

The input data for Banquet Preparation is contained in a file named `data/banquet`p`.dzn`, where

- p is the problem number,
- $COOK$ is an enumerated type of the cooks available,
- $DISH$ is an enumerated type of the dishes to be cooked,
- no_tasks is the total number of tasks for all dishes,
- $steps$ is an array of sets of tasks, showing the tasks that are required for each dish (note that some tasks may be shared by dishes), and
- $time$ is a 2D array giving for each task and each cook how long that cook takes to complete the task, or 0 if the cook cannot complete the task.

The aim is to generate respectively the start time s and the cook c for each task.

The data declarations and main decisions are hence:

```
enum COOK; % the set of cooks available
enum DISH; % the set of all dishes to prepare
int: no_tasks; % total number of tasks
set of int: TASK = 1..no_tasks;
array[DISH] of set of TASK: steps; % steps to making a dish
array[TASK, COOK] of int: time; % amount of time to complete task by cook

int: maxt = sum(array1d(time));
set of int: TIME = 0..maxt;
array[TASK] of var TIME:    s;  % start time for task
array[TASK] of var COOK:    c;  % cook for task
var TIME: obj;
```

The output required for every stage is of the form

```
s = [start time for each task];
c = [the cook assigned to each task];
obj = time when all dishes are completed;
```

The aim is to minimize the finish time (`obj`). Note that the steps in a dish must be completed in the order of task number.

For example given the data file

```
COOK = { BOB, CAROL, TED, ALICE };
DISH = { MAPOTOFU, FRIEDEEL, STICKYRICE };
no_tasks = 9;
steps = [1..3, {2,4,5,6}, 7..9];
time = [| 17, 25,  0,  0
        |  0, 30,  0, 40
        |  0,  0,160,150
        | 30,  0, 50,  0
        |  0, 66,  0, 55
        |  0,  0, 65, 78
        | 56, 62,  0,  0
        |  0, 70, 80,  0
        |  0,  0,100, 90 |];
```

which gives 3 dishes involving 9 tasks, one of which is shared.

An optimal solution is given by

```
s = [0, 17, 47, 73, 103, 207, 17, 73, 158];
c = [BOB, CAROL, TED, BOB, ALICE, TED, BOB, CAROL, ALICE];
obj = 272;
```

Note how task 2 is completed before both task 3 and task 4 begin. Note also how no cook is doing two things at once: BOB works on task 1 during the time interval 0..17, task 7 during 17..73, and task 4 during 73..103; CAROL works on task 2 during 17..30, and task 8 during 73..143; TED works on task 3 during 47..207 and task 6 during 207..272; and ALICE works on task 5 during 103..158 and task 9 during 158..248.

The assignment requires you to build an efficient model for the problem, and then an effective search strategy to find good schedules in no more than 30 seconds.

Note that given this is a scheduling problem you will want to use scheduling globals to make the model efficient. Note that there are other important concepts of the problem which you may want to use in the search, for example the end time for each task, or the duration of each task. You may want to explore a good restart strategy as part of your search.

In addition to setting the "Solver flags" in the IDE or relying on command line arguments, MiniZinc now supports also the following self-explanatory annotations for invoking restart search:

- `restart_luby(int: scale)`
- `restart_geometric(float: base, int: scale)`

- `restart_constant(int: scale)`
- `restart_linear(int: scale)`

Remember to include "`gecode.mzn`". You are expected to use such annotations in your model submissions. Please make sure that you have the *latest* version of MiniZinc installed.

3 Instructions

Edit `banquet.mzn` to solve the optimization problem described above. Your `banquet.mzn` implementation can be tested on the data files provided. In the MiniZinc IDE, use the *Run* icon to test your model locally. At the command line use,

```
mzn-gecode ./banquet.mzn ./data/<inputFileName>
```

to test locally. In both cases, your model is compiled with MiniZinc and then solved with the GECODE solver.

Resources You will find several problem instances in the `data` directory provided with the handout.

Handin This assignment contains 4 solution submissions and 3 model submissions. For solution submissions, we will retrieve the best/last solution the solver has found using your model and check its correctness and quality. For model submissions, we will retrieve your model file (.mzn) and run it on some hidden data to perform further tests.

From the MiniZinc IDE, the *Submit to Coursera* icon can be used to submit assignment for grading. From the command line, `submit.py` is used for submission. In both cases, follow the instructions to apply your MiniZinc model(s) on the various assignment parts. You can submit multiple times and your grade will be the best of all submissions.[1] It may take several minutes before your assignment is graded; please be patient. You can track the status of your submission on the *programming assignments* section of the course website.

4 Technical Requirements

For completing the assignment you will need MiniZinc 2.1.x and the GECODE 5.0.x solver. Both of these are included in the bundled version of the MiniZinc IDE 2.1.x (`http://www.minizinc.org`). To submit the assignment from the command line, you will need to have Python 3.5.x installed.

[1] Solution submissions can be graded an unlimited number of times. However, there is a limit on grading of **model submissions**.