// --------------------------------------------------------------------------

// Licensed Materials - Property of IBM

//

// 5725-A06 5725-A29 5724-Y48 5724-Y49 5724-Y54 5724-Y55

// Copyright IBM Corporation 1998, 2020. All Rights Reserved.

//

// Note to U.S. Government Users Restricted Rights:

// Use, duplication or disclosure restricted by GSA ADP Schedule

// Contract with IBM Corp.

// --------------------------------------------------------------------------

using CP;

/\*

This model solves a school time tabling problem.

Given teacher skills, room equipment and pupil course requirement,

the model generates for each course a time table specifying :

- a teacher

- a start time

- a room

constraints are used to:

- ensure the course ends after it starts

- ensure course numerotation is chronological

- ensure that a teacher is required once at any time point.

- ensure the teacher can teach the discipline

- ensure that a room is required once at any time point.

- ensure the room can support the discipline

- ensure that a class follows one course at a time

- ensure that for given class and discipline, the teacher is always the same

- ensure a course starts and end the same halfday

- insert break duration between specified disciplines

- avoid to have the same discipline taught twice a day

- ensure that the morning disciplines end in the morning

Note: To reduce the amount of decision variable, we choose to use

course start times as time points where uniqueness of resources (classes,

teachers and rooms) is enforced.

This model is greater than the size allowed in trial mode.

If you want to run this example with the large data set, you need a commercial edition of CPLEX Studio to run this example.

If you are a student or teacher, you can also get a full version through

the IBM Academic Initiative.

\*/

execute{

}

tuple Pair {

string a;

string b;

};

// requirment: which calss needs to be taught which dispiline every week.

tuple Requirement {

string Class; // a set of pupils

string discipline; // what will be taught

int Duration; // course duration

int repetition; // how many time the course is repeated

};

//

// user given model data

//

// these disciplins need break

{Pair} NeedBreak = ...; // disciplines that should not be contiguous in time

// these disciplines must start in the morning

{string} MorningDiscipline = ...; // disciplines that must be taught in the morning

// which teacher can teach which dispiline, one teacher can teach multiple discplines

{Pair} TeacherDisciplineSet = ...; // what are the teacher skills

// some rooms can only conduct these disciplines

{Pair} DedicatedRoomSet = ...; // a set of disciplines requiring special rooms

{Requirement} RequirementSet = ...; // the educational program

{string} Room = ...; // the set of available rooms

int BreakDuration = ...; // time interval between two disciplines

int DayDuration = ...; // must be even (morning duration equals afternoon duration)

// 6 days per period for planning

int NumberOfDaysPerPeriod = ...; // how many worked days per period

//

// vocabularies

//int

// get all classes

{string} Class = {c | <c,d,u,n> in RequirementSet };

// get all teachers

{string} Teacher = { t | <t,d> in TeacherDisciplineSet };

// get all dispiline

{string} Discipline = {d | <t,d> in TeacherDisciplineSet };

//

// time expressions

// 3 hours per half day

int HalfDayDuration = DayDuration div 2;

int MaxTime = DayDuration\*NumberOfDaysPerPeriod;

range Time = 0..MaxTime-1;

//

// convenience expressions for room compatibility

// given room x, go through every room and displine such that x, displine is in dedicated room set

// or z, displine in didecated room set, check if the carinality of this set is zero, if zero, it is also possible

/// what does this mean?

// for eample, given math, room R1, <math, r1> not in dedated set

// <r1, math> such that <r1,math> in dedicated false

// k = physics,s, z = r1, x = r1, <r1,physics> or <r1, math> not in dedicatd room set

// this set is zero, this room is ok

// d = math, x = stadium, the constructed set is <x,k> = <stadium, k=sports>, z = r1, <r1, stadium> is in the set

// k = sport, x= r1, z = r3,

{Pair} rooms[d in Discipline, x in Room] = {<x,k> | z in Room, k in Discipline: (<x,k> in DedicatedRoomSet)

|| (<z,d> in DedicatedRoomSet)};

// i corrected a mistake in this, it should be <x,k> not <z,k> to include in the set

// this means if x,d not in didicated roomt set, then we see if x is didicted to any other displine, or if this displine is dedicated

// to any other room, if this set is not empty, this pair <x,d> cannot be possible, cox either x is used for some other displine,

// or this displine d is taught on some other room

int PossibleRoom[d in Discipline, x in Room] =

<x,d> in DedicatedRoomSet

|| 0 == card({<x,k> | z in Room, k in Discipline

: (<x,k> in DedicatedRoomSet)

|| (<z,d> in DedicatedRoomSet)});

// num of rooms

int NbRoom = card(Room);

range RoomId = 0..NbRoom-1;

// get possible rooms ids for this displine

{int} PossibleRoomIds[d in Discipline] =

{i | i in RoomId, z in Room

: (PossibleRoom[d,z] == 1) && (i == ord(Room,z))};

//

// convenience expressions for teacher skills

//

// possible teacher disciplines

// for each teacher, get all possible displine

{string} PossibleTeacherDiscipline[x in Teacher] = {d | <x,d> in TeacherDisciplineSet };

int NbTeacher = card(Teacher);

range TeacherId = 0..NbTeacher-1;

// possible teacher ids

{int} PossibleTeacherIds[d in Discipline] =

{i | i in TeacherId, z in Teacher

: i == ord(Teacher, z)

&& d in PossibleTeacherDiscipline[z] };

//

// convenience expressions for requirement instantiation

//

// for a given requirement, an instance is one course occurrence

tuple Instance {

string Class;

string discipline;

int Duration;

int repetition;

int id;

int requirementId;

};

{Instance} InstanceSet = {

<c,d,t,r,i,z> | <c,d,t,r> in RequirementSet

, z in ord(RequirementSet,<c,d,t,r>) .. ord(RequirementSet,<c,d,t,r>)

, i in 1..r

};

//

// decision variables

//

dvar int Start[InstanceSet] in Time; // the course starting point

dvar int room[InstanceSet] in RoomId; // the room in which the course is held

dvar int teacher[InstanceSet] in TeacherId; // the teacher in charge of the course

//

// helper variables

//

dvar int End[InstanceSet] in Time; // the course end time

dvar int classTeacher[Class,Discipline] in TeacherId; // teacher working once per time point

dvar int makespan in Time; // ending date of last course

//

// search setup

//

execute {

writeln("MaxTime = ", MaxTime);

writeln("DayDuration = ", DayDuration);

writeln("Teacher = ", Teacher);

writeln("Discipline = ", Discipline);

writeln("Class = ", Class);

writeln("rooms = ", rooms)

var f = cp.factory;

var selectVar = f.selectSmallest(f.domainSize());

var selectValue = f.selectRandomValue();

var assignRoom = f.searchPhase(room, selectVar, selectValue);

var assignTeacher = f.searchPhase(teacher, selectVar, selectValue);

var assignStart = f.searchPhase(Start, selectVar, selectValue);

cp.setSearchPhases(assignTeacher, assignStart, assignRoom);

var p = cp.param;

p.logPeriod = 10000;

p.searchType = "DepthFirst";

p.timeLimit = 600;

}

// minimize makespan

minimize makespan;

subject to {

makespan == max(r in InstanceSet) End[r];

// help proving optimality

makespan >= max(c in Class) sum(r in InstanceSet : r.Class == c) r.Duration;

// ensure the course ends after it starts

forall(r in InstanceSet)

End[r] == r.Duration + Start[r];

// ensure course numerotation is chronological

forall(i, j in InstanceSet

: i.id < j.id

&& i.requirementId == j.requirementId)

Start[i] < Start[j];

// ensure that a teacher is required once at any time point.

forall(r in InstanceSet, x in Teacher) {

if(r.discipline in PossibleTeacherDiscipline[x])

(sum(o in InstanceSet

: o.discipline in PossibleTeacherDiscipline[x])

(Start[o] >= Start[r])

\*(Start[o] < End[r])

\*(teacher[o] == ord(Teacher,x))) < 2;

}

// ensure the teacher can teach the discipline

forall(r in InstanceSet)

teacher[r] in PossibleTeacherIds[r.discipline];

// ensure that a room is required once at any time point.

// only consider instances that happen after current r, so we dont repeat constraints

// consider events before r and after r at the same time is not correct, events before and after can happen at same time

forall(r in InstanceSet, x in Room) {

if(PossibleRoom[r.discipline,x] == 1)

(sum(o in InstanceSet : 1 == PossibleRoom[o.discipline,x])

(Start[o] >= Start[r])

\*(Start[o] < End[r])

\*(room[o] == ord(Room,x))) < 2;

}

// ensure the room can support the discipline

forall(r in InstanceSet)

room[r] in PossibleRoomIds[r.discipline];

// ensure that a class follows one course at a time

forall(r in InstanceSet, x in Class) {

if(r.Class == x)

(sum(o in InstanceSet : o.Class == x)

(1 == (Start[o] >= Start[r])\*(Start[o] < End[r]))) < 2;

}

// ensure that for given class and discipline, the teacher is always the same

forall(c in Class, d in Discipline, r in InstanceSet

: r.Class == c && r.discipline == d)

teacher[r] == classTeacher[c, d];

// ensure a course starts and end the same halfday

forall(i in InstanceSet : i.Duration > 1)

(Start[i] div HalfDayDuration) == ((End[i]-1) div HalfDayDuration);

// insert break duration between specified disciplines

forall(ordered i, j in InstanceSet, a,b in Discipline

: (<b,a> in NeedBreak || <a,b> in NeedBreak)

&& i != j

&& i.Class == j.Class

&& ((i.discipline == a && j.discipline == b)

|| (i.discipline == b && j.discipline == a)))

// courses do not belong to the same day

((Start[i] div DayDuration) != (Start[j] div DayDuration)) ||

// courses do not belong to the same halfday

((Start[i] div HalfDayDuration) != (Start[j] div HalfDayDuration)) ||

// courses are separated by BreakDuration

((Start[i] > End[j])\*(Start[i] - End[j]) +

(Start[j] > End[i])\*(Start[j] - End[i])) >= BreakDuration;

// avoid to have the same discipline taught twice a day

forall(ordered i,j in InstanceSet: i.discipline == j.discipline && i.Class == j.Class)

(Start[i] div DayDuration) != (Start[j] div DayDuration);

// ensure that the morning disciplines end in the morning

forall(d in MorningDiscipline, i in InstanceSet

: i.discipline == d)

(Start[i] % DayDuration) < HalfDayDuration;

};

//

// generate time table

//

tuple Course {

string teacher;

string discipline;

string room;

int id;

int repetition;

};

{Course} timetable[t in Time][c in Class] = {

<p,d,r,i,n>

| d in Discipline

, r in Room

, x in InstanceSet

, n in x.repetition..x.repetition

, p in Teacher

, i in x.id..x.id

: (t >= Start[x])

&& (t < End[x])

&& (x.Class == c)

&& (room[x] == ord(Room,r))

&& (ord(Teacher,p) == teacher[x])

&& (d == x.discipline)

};

// force execution of postprocessing expressions

execute POST\_PROCESS {

timetable;

for(var c in Class) {

writeln("Class ", c);

var day = 0;

for(var t = 0; t < makespan; t++) {

if(t % DayDuration == 0) {

day++;

writeln("Day ", day);

}

if(t % DayDuration == HalfDayDuration)

writeln("Lunch break");

var activity = 0;

for(var x in timetable[t][c]) {

activity++;

writeln((t % DayDuration)+1, "\t",

x.room, "\t",

x.discipline, "\t",

x.id, "/",

x.repetition, "\t",

x.teacher);

}

if(activity == 0)

writeln((t % DayDuration)+1, "\tFree time");

}

}

}