Constraint Based Compensation Scheduler

Ahmed Amr [37-5919] and Omar Emad [37-3037]

German University in Cairo, Egypt

1 Introduction

Compensation Scheduling problems are real life combinatorial problems concerned with scheduling a number of events within a specific time slots. If solved manually, it is hard to guarantee to assign all events in an optimal way such that maximum number of staff members satisfied in terms of there own preferences.

In this paper, we present a model to develop a constraint-based system to solve the compensation scheduler problem. Constraint programming is a problem-solving technique that works by incorporating constraints into a programming environment. Constraints are relations which specify the domain of solutions by shrinking combinations of values.

In this paper, we show how the compenstation scheduler problem can be formalized as a constraint satisfaction problem (CSP) and implemented by means of specialized constraint solving techniques that are available in a constraint logic programming language (CLP) . In solving the problem, we define the variables, domain of variables , all constraints which satisfies the problem and calculate the objective function to get optimal solution based on preferences.

The paper is organized as follows. The next section introduces the Compensation Scheduler problem and its requirements. Then we show how the problem can be modeled as a CSP problem and how we implemented the different constraints using clpfd library in Prolog. Finally, we conclude with a summary.

2 Problem Description and Requirements

At the beginning of every semester, the academic calendar is announced. The academic calendar includes the dates of the start and end of each semester. Thus includes information about when the lectures start and when the tutorials start. The academic calendar also includes the planned weeks when the midterm exams, final exams and revision weeks are to be held. All the public holidays and the days without teaching activities are also announced. Whenever any day is announced to either be a holiday or a no teaching activity day, all the events (lectures, tutorials, labs) that were scheduled on that day have to be compensated.

The Compensation Scheduler problems aims at re-schedule all events whenever these days is announced as no teaching activity day with an autonomous process for all staff members given in a particular term of staff members' preferences.
The Compensation Scheduler problem consists of a set of schedules or information related to slot event. The slot event contains information about staff member,semester,course, set of tutorials (if staff member is TA then size of tutorials is one), set of common days off between staff member and students , room type , and which room is free in specific slot.Moreover, a set of free rooms that is applicable to compensate in a specific time and set of preferences from staff members.The task is to assign free room in a specific slot to all events that have to be compensated.

The task can be performed into three parts.In first part, we will figure out all events that need to be assigned to time slot and room. In second part, Constraints must be considered related to conflicts between staff members,groups and time slots. In third part, the optimization must satisfy all staff preferences and system get optimal schedule based on preferences by inserting the slots in which prefer/ do not prefer to have the compensations in.

When assigning an event to a certain room and time slot, several constraints must be considered.

1. **Events Clash Constraint:** The event schedule cannot have a more than one teaching event at the same time slot and room.

2. **Free Rooms Assignment Constraint:** The event schedule should assign time slot and room from set of free rooms.

3. **Room Type constraint:** All events schedules must be assigned to room with same room type as required in set of schedules.

4. **Tutorial Group Clash Constraint:** A group can not be assigned to multiple meetings at the same time.

5. **Staff Members Clash Constraint:** A staff member can not be assigned to multiple meetings at the same time.

3 Constraint Satisfaction Problem (CSP) Model

A constraint satisfaction problem is defined as a set of variables where each variable can be assigned between values on nonempty domain and a set of constraints each restricting values that variables can take to insure consistency of problem.

3.1 Description for input

Before we go throw to CSP model, we should mention all input in main predicate to understand how CSP work.

1. **Schedules:** is defined as a list of tuple ⟨ StaffID , Semester , Course , Tutorials, RoomType , DaysOFF , Event_Room_Slot ⟩

   - StaffID : The identification number of staff member with 0-based index.
   - Semester: The identification for year of current semester eg. 0 is identification for semsester 1.
   - Course: The identification of course in a specific semester with 0-based index.
   - Tutorials : A list of tutorial numbers that are included in same group event.
   - RoomType : The type of the room that should be assigned for teaching event.
   - *Event_Room_Slot:* It is a key value or hash value that indicate for a tuple which it represents free room in specific time slot ⟨ room_ID, RoomType,Day,Slot ⟩ Since the maximum number of rooms is 50 rooms per room type and number of room types is 4 and total number of slots for 6 days are 30 slots then the total number of combinations = $50 \times 4 \times 6 \times 5 = 6000$
     The key value can be calculated using this equation
     Event_Room_Slot = room_ID $\times 4 \times 6 \times 5$ + RoomType $\times 6 \times 5$ + Day $\times 5$ + Slot.

     We can prove our intuition by stating a similar idea and figure out the general rule that satisfy *Event_Room_Slot* and guaranteed by the uniqueness of the equation.For example,if we have a tuple that represent time for a specific month ⟨Day,Hour,Minute⟩.Now, to know the total number of combinations to represent time in minutes = 30 (days) $\times$ 24 (hours) $\times$ 60 (minutes) = 43200 (minutes).

and to represent the tuple in terms of a single variable(minutes)
Time_in_minutes = day × 24 × 60 + hour × 60 + minute.

We can generalize equation of hash value for any tuple of $\langle T_1, T_2, \ldots, T_N \rangle$ and domain of values $\langle D_1, D_2, \ldots, D_N \rangle$ such that T_i between 0 to $D_i - 1$
Key/Hash Value = $T_1 \times (D_2 \times D_3 \ldots D_N) + T_2 \times (D_3 \times D_4 \ldots D_N) + \ldots + T_{N-1} \times D_N + T_N$.

For extracting the tuple from key value

- room_id = $\lfloor \frac{Event\_Room\_Slot}{4 \times 6 \times 5} \rfloor$ mod 50.
- RoomType = $\lfloor \frac{Event\_Room\_Slot}{6 \times 5} \rfloor$ mod 4.
- Day = $\lfloor \frac{Event\_Room\_Slot}{5} \rfloor$ mod 6.
- Slot = Event_Room_Slot mod 5.

2. **Free Rooms:** is defined as a list of key values to identify free rooms using the same equation of *Event_Room_Slot*

3. **Preferable and NonPreferable:** is defined as a list of key values that indicate for a tuple which it represents preference for a staff member to a specific time slot ⟨ Staff_ID , Day , Slot ⟩.If we assume maximum number of staff members is 60 then total number of combinations for the tuple = 60 × 6 × 5 = 1800. Therefore, We can represent key value for all preferences by an equation = Staff_ID × 6 × 5 + Day × 5 + Slot.

3.2 Description for *CSP* Model

In modeling the Compensation Scheduler problem as a CSP we use the following notations:

- n is number of events to be compensated.
- m is maximum number of key values to identify event time slot as we mentioned in section 3.1. in this problem we state that m = 6000.
- $E = \{ E_1 \ldots E_N \}$ represents the Event_Room_Slot variables that should be assigned to an event group,where E_i indicates to event time slot to an event i in list of *Schedules*.
- $D = \{0 \ldots m - 1\}$ indicates a set of m key values for events such that domain of each variable E_i is between 0 to m-1.

Different constraints are applied on E_i eliminating the different values each variable can take and thus reducing the domains of every variable to improve the

efficiency of the search thus reaching a solution in an optimal time and reach the consistency for the problem.
We describe the requirements of our problem in terms of prolog using the constraint solver over finite domains (clpfd).

1. The Events Clash Constraint stating that no schedule event can have more than one room at the same time slot can be enforced using *all_distinct/1* global constraint. We have to constraint list of Event_Room_Slot variables E by applying *all_distinct/1* on this list would ensure that all events are taken in different rooms with respect to every time slot.

2. The Free Rooms Assignment Constraint has to be fulfilled in order to assign *Event_Room_Slot* from a valid free room with respect to a specific time slot in list of *FreeRooms*.We implement predicate *all_in_free_rooms/2* is true iff each variable E_i is in FreeRooms using *element/3* global constraint

```
all_in_free_rooms([] , _).
all_in_free_rooms([E | T] ,  FreeRooms):-
                element(_,FreeRooms , E),
                all_in_free_rooms(T , FreeRooms).

```

3. The Room Type constraint compares RoomType in list of *Schedules* as mentioned in section 3.1 with the room type in E_i. Basically,We compare using equation for extracting RoomType from key value E_i as we mentioned in section 3.1. We compare all values using reification #<==> which is used to reified constraints to relate a list of finite domain variables E to the number of elements that satisfy reification RoomType #= (E_i // 30) mod 4 #<==> B where B is 1 iff equation satisfies otherwise B is 0 and then accumulate all Bs using predicate *sum/3* global constraint and check summation is equivalent to length of list E.

```
same_RoomType_compare(E , RoomType , B):-
    Z #= (E // 30) mod 4 , Z #= E #<==> B.

same_RoomType_constraint(Schedules):-
  get_RoomSlots(Schedules , E) ,
  get_RoomType(Schedules , RoomTypes) ,
  maplist(same_RoomType_compare , E , RoomTypes , Check),
  sum(Check, #=, Num),
  length(Schedules , Num).

```

4. The tutorial group clash constraint stating that no group can have more than one time slot. First of all, we should extract tuple of $\langle$ $Semester_i$, $Tutorial_{i,j}$, E_i mod 30 $\rangle$ from list of *Schedules* and hash tuple with an equation

Hash = $Semester_i \times 20 \times 30 + Tutorial_{i,j} \times 30 + E_i \bmod 30$
in a list called *HashList* as we assume maximum number of tutorials in one semester is 20 then we have to constraint list by applying *all_distinct/1* to ensure that no group have multiple time slots.

```
get_hash_group_slot([] , []).
get_hash_group_slot([(_ , _ , _ , [] , _ ,_,_ ) | T] ,L):-
  get_hash_group_slot(T , L).
get_hash_group_slot([(_ , Semester , _ , [Tutorial|
    Tutorials] , _ ,_,E ) | T] ,[Hash | L]):-
    Hash #= Semester  * 20 * 30  + Tutorial * 30 + (E mod
    30),
get_hash_group_slot([(_,Semester,_,Tutorials , _,_,E)|T],L
    ).
group_slot_constraint(Schedules):-
  get_hash_group_slot(Schedules , HashList),
  all_distinct(HashList).
```

5. The staff members clash constraint stating that no staff member can have more than one time slot. First of all, we should extract tuple of $\langle\ StaffID_i$, $E_i \bmod 30\ \rangle$ from list of *Schedules* and hash tuple with an equation
Hash = $StaffID_i \times 30 + E_i \bmod 30$
in a list called *HashList* then we have to constraint list by applying *all_distinct/1* to ensure that no staff member assigned to multiple time slots.

```
get_hash_staff_slot([] , []).
get_hash_staff_slot([(StaffID , _ , _ , _ , _ ,_,E ) | T]
    , [Hash | L]):-
  Hash #= StaffID * 30 + (E mod 30),
  get_hash_staff_slot(T , L).
staff_slot_constraint(Schedules):-
  get_hash_staff_slot(Schedules , HashList),
  all_distinct(HashList).
```

3.3 Constraint optimization problem

Constraint Optimization Problem (COP) is the most significant generalizations of the Constraint Satisfaction Problems (CSP) model. Actually, COP is a CSP solve with an objective function. An optimal solution to a minimization COP is a solution that minimizes the value of the objective function.

The Compensation Scheduler problem should be optimized based on preferences of staff members in two lists *Preferable* and *NonPreferable*.In addition, we consider in optimization problem list of *DaysOFF* where it is mentioned in section 3.1 as days off is not preferable for students and staff members.

We compute objective function in three stages:-

1. If the current time slot is preferable for staff member then the objective function will give plenty 1.

2. If the current time slot is not preferable for staff member then the objective function will give plenty 2.
3. If the current time slot in a day off then the objective function will give plenty 3.

Finally, the optimal solution is obtained by minimizing the *Cost* where
Cost $= \sum_{i=1}^{N} PreferableObjectiveFunction(Schedule_i)$
$+ \sum_{i=1}^{N} NonPreferableObjectiveFunction(Schedule_i)$
$+ \sum_{i=1}^{N} DaysOffObjectiveFunction(Schedule_i)$.

```
calculatePrefrences(Schedules ,Preferable , NonPreferable, Cost):-
    calculatePrefrences_helper(Schedules , Preferable , 1 , Score1),
    calculatePrefrences_helper(Schedules , NonPreferable , 2 , Score2),
    calculate_days_off(Schedules  , Score3),
    Cost #=  Score1 + Score2 + Score3.

calculatePrefrences_helper([] , Prefrences, Factor , 0).
calculatePrefrences_helper([Schedule | T] , Prefrences, Factor , Score):-
  maplist(compare_preference(Schedule) , Prefrences , TotalCosts),
  sum(TotalCosts , #= , Cost),
  calculatePrefrences_helper(T , Prefrences , Factor, Score1) ,
  Score #= Score1 + Cost * Factor.

compare_preference((StaffID , _ , _ , _ , _, _, E),Pref,B) :-
  Hash #= StaffID * 30 + (E mod 30) ,
  Pref #= Hash #<==> B.

compare_days_off(E , Day , B) :-
  Hash #= (E // 5)  mod 6 ,
  Day #= Hash #<==> B.

calculatePrefrences_helper([],Prefrences, Factor , 0).

calculatePrefrences_helper([Schedule | T],Prefrences,Factor, Score):-
  maplist(compare_preference(Schedule),Prefrences,TotalCosts) ,
  sum(TotalCosts , #= , Cost),
  calculatePrefrences_helper(T , Prefrences , Factor, Score1) ,
  Score #= Score1 + Cost * Factor.

calculate_days_off([] , 0).
```

```
calculate_days_off([Schedule | T]  , Score):-
  Schedule = (_ , _ , _ , _ , _, DaysOFF, E),
  maplist(compare_days_off(E) , DaysOFF , TotalCosts),
  sum(TotalCosts , #= , Cost),
  calculate_days_off(T  , Score1),
  Score #= Score1 + Cost * 3.
```

3.4 Main Predicate

In a nutshell, This is the main predicate including all CSP and COP model.

```
solve(Schedules , FreeRooms , Preferable , NonPreferable):-
%Extract all Varibles from Schedules that need to compensate
  get_RoomSlots_Var(Schedules , E),
  %Domain of varibles from 0 to total number of combinations
  E ins 0 .. 5999,
  %All constraints
  free_room_constraint(E , FreeRooms),
  all_distinct(E),
  same_RoomType_constraint(Schedules),
  group_slot_constraint(Schedules),
  staff_slot_constraint(Schedules),
  %Objective Funciton
  calculatePrefrences(Schedules , Preferable , NonPreferable
    , Cost),
  %COP by minimizing the CSP model using objective function
  labeling([min(Cost)] , E).
```

4 Conclusion

To sum up, the Compensation Scheduler problem has been discussed and a possible solution using constraint logic programming has been presented with optimizing the problem using objective function based on preferences.