Optimization of constraint computing in Automatic Time Tabling

Peter Wilke and **Johannes Ostler**
University Erlangen-Nuremberg, Informatik 2
Martensstrasse 3, 91058 Erlangen, Germany
Tel.: +49.9131.8527624 Fax: +49.9131.8528809
peter.wilke@cs.fau.de johanes.ostler@cs.fau.de

Abstract

The constraint computation occupies a great part of total computation time. So there is demand for optimization the update of cost adapted to the changes of the plan. Especially neighborhood searching techniques needs many cost evaluations after very little changes. So this offers a great optimization potential.

Introduction

The global goal of our software development is to provide a software framework which is capable of solving various timetabling problems. The definition of a time tabling problem bases on a set of resources, events and constraints. For the user is a GUI available to describe the settings of the special problem. The problem description is saved in a general XML format. The core of the system are some optimization algorithms, like *Simulated Annealing*, *Tabu Search*, *Genetic Algorithm* or *Branch & Bound*. On the following pages we want to explain the description and the computation of constraints.

Definition of a Time Tabling Problem

First of all we want to show how a time tabling problem can be defined in a general way. As example we use the well known School Time Tabling Problem. The events are school lessons. To each lesson must be assigned at least one room, one teacher, one class and a set of time slots when it happens. So we have found the resource types of this problem: Teacher, Room, Class and TimeSlot. For an event is defined how much resources of every type should take part on this. On that way the first constraints are defined, a MinimumConstraint and a MaximumConstraint, which are violated when to less respectively to much resources are assigned. If this resource type event tuple is marked as fixed the search space will be limited to solutions which comply with this constraints. Otherwise there must be defined a penalty function which specifies the costs of constraint violation. All penalty functions have this form:

$$p(x) = w(ax^b + c) \;\; with \;\; w \in \mathbb{R}^+ \;\; and \;\; a,b,c \in \mathbb{R}_0^+ \quad (1)$$

Whereby in that case $x = min - count$ if to less resources or $x = count - max$ if to much are assigned. There are also exposed two sorts of constraint compliance per limitation of search space or to optimize costs of solutions. The first type lower the computation time, but it is not always possible or often the way to a better solution leads over a worse.

Views on Time Tabling Problems

View on an event

On the definition of constraints are several views on the problem possible. First can be looked on a single event. This kind of view is used by the Min and MaxConstraints above. Or the size of room must be greater than the count of students in the corresponding class. This constraints must be checked after a change of the assignments of an event but only for this event. This type of constraints occur rather often.

View on Events of a Resource

Quite so common are constraints which belongs to the events assigned to a resource, as example the lessons of a teacher. Because a teacher can teach only one class at the same time, the time slots of these lessons must not overlap. The sum of the slots should be correspond to the weekly working time of the teacher. Greenly we can thought that the costs only must be updated after changing assignments of the resource, in this case the teacher. But the update will be also necessary if the time slots of a lesson of this teacher changes. A little bit more complex is the case that the list of events depends on a tuple of resources of different types. As example all lessons teacher t_x gives class c_y.

View on a Set of Events

In some cases there are special relationships between a set of events. As example events are in a chronological order. Event A should be terminated before event B starts. This constraint must be checked if the time slots of A or B changes. In a more general view an update is necessary if one of the events belonging to this relationship is modified. In many cases this can be limited to modification of resource assignments of a specific resource type.

Variables

Introduction

Variables are the connection between constraint definition and the resources and events. They returns a value of a determined type. The update mechanism in hierarchically organized. If the plan changes, the affected variables receive a message. Thereupon these will update there value and if this is modified, they will send a message to all constraints which uses them.

Definition versus Instance

Now take a look on the *Room Size Constraint*. For every event the capacity of every assigned room must be greater than the size of the class which belongs to this lesson. So for every event must be computed the minimum capacity of assigned rooms and the capacity of the class. The definition of the two variables is same for every event, take list of assigned rooms, select attribute value *capacity* and compute the minimum of capacity values respectively take list of assigned classes and select attribute value *size*. But the value differs depending on the state of the particular event. So the corresponding value must be saved for each event. The two variables must be instantiated for every event once. The constraint lists per event references to the variable instances and the penalty cost. If values of at least one of the variables changes, the constraint request the current values of the associated variable instances. If the review of compliance with the constraints gives an other result, i.e. different costs, the difference between new and old costs will be added to the total costs of plan.

Recapitulating must be adhered that each constraint maintains informations about every operand tuple and the costs belonging to this. For this purpose the constraint holds references to all variable instances. If a instance's value changes, only the costs of the affected tuples will be updated.

Selectors and Operations

Selectors extract a part of information of an input value. As example we need the minimal size of the assigned rooms to a lesson, so first of all we need a selector which returns a list of the assigned rooms of this lesson. The input value lesson contains information about the teachers, the time slots etc., but only the rooms are of interest.

An *Event Selector* represents a direct connection to an event. As a rule its result is a list of all assigned resources of a determined resource type. So the value of the selector must be updated if the event' s assignments of this type changes. For every event must be a selector instantiated. A Event Selector is specified by a resource type and in most cases an attribute. The input is an event, the output a list of values of the attribute's type or a list of resources if no attribute is determined.

If the view is on a single resource, it is a little bit more complex. As a rule the associated events are the items of interest, however usually not the list of events in principle, but the list of assigned resources to this events. In lot of cases lists of attribute values of these resources are needed. First of all the list of events is specified by a resource, the list of assigned events. The transformation from a list of events to a list of resources or attribute values can be determined by a Event Selector applied on every entry of this list.

This issue seems easier than it is. As example let look at the time slots associated with a teacher. There are some possibilities of creating lists of time slots. First of all add all time slots assigned to event associated with a teacher to one list. Secondly duplicates can be eliminated on creating this list. Thirdly for every event can be generated a separate list of time slots and then this list can be added to the result list. Elimination of duplicates is senseless on the third version.

More complex will be the list creation if the view is not only on a single resource but on a tuple of resources of different resource types, as example all events a teacher t_x and a class c_y are assigned. An *Event List Selector* is determined by set RT of resource types . The input is a tuple of resources one of each resource type of RT. If RT is empty, a list of all events is the result. Is $|RT| = 1$ a list of the events assigned to the resource will be returned. If the dimension of the tuple is n then the intersection of the n lists associated with the n resources is the result.

The constraint must be checked for every list generated by the Event List Selector. The count of lists is the count of possible tuples. If there are x teachers and y classes, there will be x*y lists.

Operations transform a value of a primitive type[1] into a value of same or another primitive type. Operations represents mathematical functions like the minimum or the sum of a list of values or the projection of a time stamp to the corresponding daytime. This is the part of our framework which needs in some cases programming effort. The framework provides a set of operations which can be selected by the GUI, but there is also the possibility to import new operations. The variety of different operations is nearly unlimited. Which approach can be selected to optimize the computation time independent of the function the operation applies respectively the code of the operation's implementation.

The first approach is based on the acceptance that the result of a function is same if the input is same. So it is possible to hold a map which saves the result for a determined input. But usually the encoding of input value and finding of the corresponding result is more expensive than the computation of the result. Therefore, this is a optional optimization, which use must be determined for each operation individually.

The second approach is to have a look only on the changes of the input value. This will be only useful if input is a list of values. As example there is a sum of a integer values list. Value a was removed from the list and value b added. So is the new result $result_{new} = result_{old} - a + b$. This approach can be implemented by committing the whole list, the list of added elements and the list of removed elements to the operation. This can then decide how to compute the result on the fastest way. On that way the call of the operation is independent of the operations implementation, but in many cases there will be an overhead on generating the three lists.

[1] Possible primitive types are integer, double, bool, string, time stamp or time slot respectively list of values of these types

On that way the interface of a operation class is partly determined by the optimization approaches above.

```
public interface Operation{

   public Object compute(
             Object inputValue};

   public Object computeChange(
             Collection inputList,
             Collection addedElements,
             Collection removedElements,
             Object oldValue);

   public OPType getResultType(
             OPType inputType);

}
```

At the first computation *compute* after changes *computeChanges* will be called. The parameters *addedElements* and *removedElements* must be cleared after setting the new value and get updated after changes of *inputList*. This overhead is in some cases more expensive than the profit. The method *getResultType* returns the type of the operations result dependent on the type of the input value. On that way the same *Operation* class can provide computation for different input types, as example the sum of a integer or a double list.

Event Variable

A special type of variables produces a transformation from an event into an operand. Such variables are called *Event Variables*. The input is an event and the associated resources, the value is an operand value. For calculation of the operand value the Event Variable uses an Event Selector followed by a sequence of operations o_1, .. o_n. Whereby the result of operation o_i is the input of operation $o_{i+1} \forall i < n$. The result of the Event Selector is the input of o_1 and that of o_n is the operand of the constraint.

So an Event Variable is totally determined by an Event Selector and a list of operations. For each event the variable must be instantiated.

Event List Variable

The variety of generation of event lists is shown in the section about selectors. Creating a list of events by a selector is defined by a tuple of resources of given resource types. Because a Event List Variable uses a Event List Selector to create the list of events, the input of this variable is such a tuple. The event list is transformed by an Event Selector into a list of primitive types. This list is converted by a sequence of operations into the result.

Therefore, an Event List Variable is specified by an Event List Selector to create the list of events, an Event Selector to convert every event into a list of primitive types or resources, a predefinition how the list per event should be unified and after all a sequence of operations.

Constraints

Introduction

A constraint is a triple of a relational operator and two operands. An operand is either a variable, another constraint or a constant value. A relational operator compares two values of primitive types and returns an integer value. The result will be zero, when the operands complies with the operator. In the other case the result will be an integer x greater than zero. This x will be used in the penalty function to compute costs of this constraint violation.

Event Constraint

Event Constraints describe a restriction based on a view on a single event. The operands are Event Constraints, Event Variables or constant values. The scope of the constraint is determined by a group of events. For all events of this group must be checked whether that complies with this constraint. The constraint must collect information about every event, a reference to the corresponding operands, as example to Event Variable instances and the current costs. Figure 1 shows the structure of an Event Constraint.

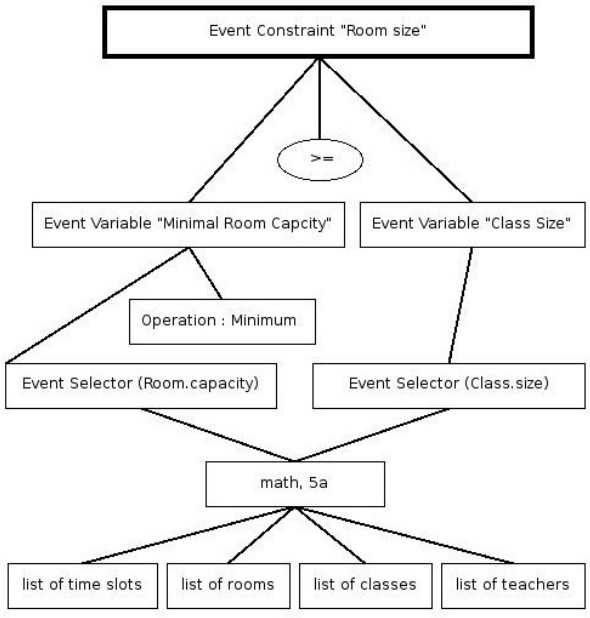

Figure 1: Example of an Event Constraint: Room Size Constraint

Event List Constraint

Constraints based on view on lists of events are defined by an Event List Constraint. The operands are other Event List Constraints, Event List Variables or constant values. The input is a tuple of resources which correlates to the union of resource types needed as input by the variables or constraints. As Example the constraint has two variables A and B. A needs a tuple of resources of types rt_1 and rt_3 and B a

resource tuple of types rt_2 and rt_3. Then the input of the constraint must be a tuple of types rt_1, rt_2 and rt_3. The Structure of an Event List Constraint is described in Figure 2.

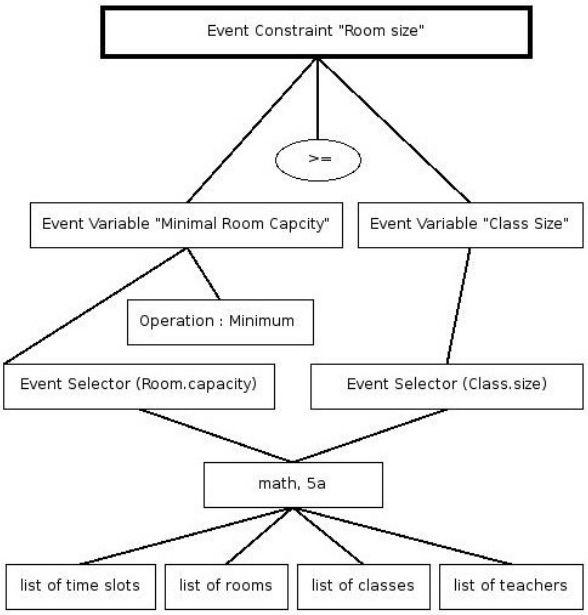

Figure 2: Example of an Event List Constraint: No Time Clash for Teacher

Optimization Potential on Cost Evaluation

Reuse of Results

Many constraints uses the same informations. So the variables and there values can be reused. Therefore, the variable must be computed only at most once on an update cycle. In the section about operations is described how the result of a operation can be reused respectively the computation can with information about old value and changes get optimized.

Evaluate only Changes

As a rule after every change the costs of plan can be evaluated by checking all constraints. But as example at the *Simulated Annealing* after every change of assignments of resources of one event the current costs must be evaluated.

Indeed it will be enough to check the constraints affected by the changed event. So every Event Variable instance references the constraint instances which uses this instance. If the event changes, the Event Variable will check whether its value has changed. When the value changes the constraint instance will be evaluted. The old costs will get subtracted from plan costs and the new costs will be added. This strategy offers a great potential of optimization. If there are 100 events and in one move changes at most one event. On that way the computation effort can be reduced nearly to 1 percent. In most cases only the assignments of one resource type of the event changes at one move. So only the variables associated to this event and this resource type must get updated after a move.

Conclusion

Computing of constraints occupies the greatest part of computation time of the execution of Automatic Time Tabling algorithms. Especially neighborhood searching techniques needs many cost evaluations after very little changes. The potential of optimization is enormous. With an intelligent update mechanism like that one described above the computation time can be reduced hugely.