Designing Global Scheduling Constraints for Local Search: A Generic Approach

Markus Bohlin  Waldemar Kocjan  Per Kreuger  
E-mail: \{bola,waldemar,piak\}@sics.se  
Swedish Institute of Computer Science  
Box 1263, S-164 29 Kista, Sweden

ISSN 1100-3154  
ISRN:SICS-T–2002/20-SE  
November 10, 2002

Abstract

In this work we present a novel method to automate the computation of global constraints cost for local search. The method is based on the representation of a global constraints as graph properties on a binary constraint network. This formulation simplifies the implementation of global constraints in local search, and provides a cost that can be readily compared to one obtained for subproblems using binary constraints exclusively. The cost obtained can be efficiently updated during the search using incremental methods. The representation of a global constraint as outlined above can also be used for generation of suitable neighborhoods for the constraint. This is done using simple repair functions applied on the elementary constraints in the global constraint graph. We show the usability of our approach by presenting formulations of global constraints in non-overlapping and cumulative scheduling.
1 Introduction

Local search has been used numerous times in the past, mainly to solve hard problems and for problems with limited solving time. Most of the local search procedures have been custom made to solve a specific type of problem. This close tie between solver and problem means that it is nearly impossible to reuse a solver on different problems. This also means that solving a problem with local search requires both knowledge in the problem domain and in heuristics.

This close relation between problem and solver stands in sharp contrast to the declarative modeling of a problem that is possible when using other solving techniques. For example, in constraint programming and linear programming a problem is modeled as a set of constraints that should hold in order for the problem to be solved.

The lack of declarative modeling in local search procedures would be partially overcome by the inclusion of global constraints. This would allow the user of the constraint solver to specify the structure of the problem on a higher level than possible with the low-level constraints available in most general-purpose local search constraint solvers.

Recently, the usability of local search constraint satisfaction has improved with the introduction of global constraints [6, 3]. In this work a separate cost representing the degree of violation for each global constraint is calculated. The cost of all the constraints in the model is then added to form the total cost of the problem.

The problem with this way of handling global constraints is twofold. First, the construction of a cost function for a global constraint is far from trivial, and requires knowledge of the implementation of the solver in question. Second, the cost function of all constraints, global and primitive, must be comparable with each other. Otherwise, the solver efficiency will suffer from the resulting imbalance.

Our work is based on a representation of global constraints as graph properties on structured networks of elementary constraints [1]. It was shown in the same work that it is possible to represent many global constraints as graph properties on structured networks of elementary constraints. We show how to use such a representation of a global constraint to calculate a cost usable for local search procedures and ordering heuristics in systematic constraint programming. The cost obtained is based on elementary constraints, and can thus be compared to the cost of both single primitive constraints and other global constraints constructed using the same technique.

A important characteristic of local search is the efficiency of the cost calculation. Typically, a neighbor is constructed by changing the value of a
single variable in the problem. One way of speeding up the cost calculation is to take advantage of this small difference between a neighbor and the current solution. This makes it possible to use an incremental cost calculation, recalculating only the parts of the cost that has changed. We show how it is possible to use incremental cost calculation for constraints described as properties on graphs, significantly reducing computation time.

The paper is organized as follows. First, we discuss some work related to this paper. Then, we give some definitions and briefly describe the basic local search model our work is directed to. We continue by describing the model of global constraints as properties on structured networks of elementary constraints. We use the global constraint \texttt{nbdifferences} as a running example in how to model a global constraint using our approach. Next, we discuss some topics of interest when using the model proposed to calculate costs for global constraints. We discuss the cost model, appropriate elementary constraint costs, incremental cost calculations, and constraint-specific neighborhoods. We continue by demonstrating the flexibility and usability of our approach, representing ten important global constraints using the methods in this paper. We show how to efficiently calculate the cost from scratch for these constraints, and how to do it incrementally for the \texttt{alldifferent}, \texttt{cumulative-1} and \texttt{cumulative} constraints.

1.1 Related Work

Local search for general constraint satisfaction with global constraints is fairly uncommon, although a few publications have been made lately. Global constraints in local search are introduced by Nareyek, who uses an approach where a selected constraint improves the current assignment in a local manner [6]. He also gives ad hoc costs for some global constraints including \texttt{ordered-tasks} and a version of the \texttt{serialized} constraint.

Galinier and Hao describes a general constraint solver using local search [3]. The solver handles a small set of global constraints, including \texttt{alldifferent}, \texttt{capa} and \texttt{nbdifferences}, and is based on hill-climbing and tabu search. Galinier and Hao give costs for the constraints used in their work and demonstrate it on some combinatorial problems. They also propose to use the minimum number of variables that need to be modified for a constraint to be satisfied as the cost of the constraint.

Petit, Régin and Bessière use cost-based filtering for systematic constraint programming [7]. The authors also propose two general cost policies for global constraints. The first is to use the minimum number of variables needed to be modified as the cost. The second is to use the number of violated binary constraints that can be used to represent the global constraint as cost.
In our work we extend the second approach to constraints that are hard to represent using binary constraints only.

Some more general work in local search that is worth mentioning is Localizer [5]. This language makes it possible to model problems using a high-level description of local search components. This approach is however focused on the procedure used to solve the problem, and thus hard to use for one with no or little experience of local search area.

A classification of global constraints as graph properties on structured networks of elementary constraints [1] has previously been done. This work introduces many useful global constraints, including disjoint-tasks.

1.2 Preliminaries

In this section, we define some concepts central to local search. We refer to textbooks in constraint programming and satisfaction [10, 4] for definitions regarding general constraint satisfaction and other relevant topics. More material regarding local search can be found in books and surveys on this topic [2, 9, 8].

Definition 1 An assignment \( v \) for a constraint satisfaction problem with \( n \) variables is a \( n \)-compound label assigning each variable in the problem a value in the domain of the variable.

We use the notation \( v(x) \) to denote the value assigned to the variable \( x \) by \( v \).

Definition 2 Let \( V \) be the set of possible assignments for the variables in a constraint satisfaction problem. A cost function \( f : V \rightarrow \mathbb{R} \) maps each possible assignment to a real number.

In this work we use cost functions returning integer values. To formalize the transition between states in local search, we define a neighborhood function. Typically, the neighborhood function returns the set of assignments that differ in one of the values assigned to the variables.

Definition 3 A neighborhood function \( N : V \rightarrow 2^V \) is a function mapping each possible assignment in a constraint satisfaction problem to a subset of the sets of possible assignments for the problem.

Local search can be described as a process generating a sequence of assignments \( v_0, v_1, v_2, \ldots \), where \( v_0 \) is the initial assignment and where it holds
that $v_{i+1} \in N(v_i)$ for each $i \geq 0$. The goal is to minimize the cost function $f(v_i)$. The search ends when $f(v_i) = 0$ or the search is aborted.

Note that this underlying model of local search does not affect the generality of our work. The representation of global constraints and methods for cost calculation in this work can be used for local search including optimization of an objective function, as well as local search on partial assignments.

2 Global Constraints as Properties on Graphs

In this section we describe how to model global constraints using properties on networks of elementary constraints. The work we present is especially suited for local search and incremental cost calculation for global constraints, and is a modified subset of a similar model found in [1]. Here it has also been shown that many global constraint can be expressed as such properties on structured networks of elementary constraints.

We represent a global constraint as a directed graph of elementary constraints, and a graph property that must hold for the constraint to be satisfied. Formally, we can describe a global constraint by the following properties:

1. A set of vertex generators, used for generation of vertices in the constraint graph,
2. A set of arc generators, used for generation of arcs in the constraint graph,
3. A set of elementary constraints, which are attached to the arcs in the graph to form the arc constraints on the constraint graph,
4. A graph property that express a global property of the graph that must hold for the constraint to be satisfied.

This section is organized as follows. First, we describe the vertex and arc generators. We then define the elementary constraints used in this work. Finally, we define properties on structured networks, and show how we can use these to calculate costs for global constraints.

We use meta-variables to give reference to subsets of vertices and arcs produced using the generators. As an example of our model, we use the global constraint `nbddifferences`, which for a list of tuples of two variables ensures that at most $k$ tuples take the same value. We specify the arguments of a global constraint as follows.

Constraint: `nbddifferences`
Arguments: `VarList : list ⟨var1.dvar, var2.dvar⟩, k : int`
The name and type of the argument is specified as `Name:type` or possibly `Name:TypeName` if `TypeName` is specified later. The types we use are integer domain variables (`dvar`), integer constants (`int`), tuples of items (`(attribute:type, . . . )`), lists with elements of a certain type (`list type`). We use the syntax `attr(x)` to extract the attribute `attr` from a tuple `x`. Typenames are specified as `TypeName :: type`. As an example of a typename, we may rewrite the specification of the arguments to the constraint above as follows.

Constraint: `nbdifferences`
Arguments: `VarList : list Item,k:int`
          `Item :: ⟨var1:dvar, var2:dvar⟩`

Sometimes we put restrictions on the arguments given to the global constraint. We state these after the arguments has been specified in the global constraint model.

2.1 Vertex Generators

The vertex generators produces vertices for the constraint network used to represent a global constraint. A vertex generator basically takes a list of items given as arguments to the global constraint as input, and generates a list of vertices. Each global constraint has at least one vertex generator. If a constraint has more than one vertex generator, the concatenated results of all vertex generators are used to form the list of vertices in the graph.

<table>
<thead>
<tr>
<th>Name</th>
<th>Vertices generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>Each item in the list produce one vertex.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Takes a list of tuples <code>A = [a_1, a_2, . . . , a_n]</code> and an attribute <code>p</code>. The vertex generator produces a list of vertices <code>V = [v_1, v_2, . . . , v_n]</code> where each vertex <code>v_i</code> is the attribute <code>p</code> of item <code>a_i</code>.</td>
</tr>
<tr>
<td>Expand</td>
<td>Takes a list of items <code>A = [a_1, a_2, . . . , a_n]</code> and an expansion operator <code>M</code>. The vertex generator produces a list of vertices which is the concatenation <code>V = V_1 ∪ V_2 ∪ . . . ∪ V_n</code>, where each list of vertices <code>V_i</code> is the result of applying the operator <code>M</code> on item <code>a_i</code>.</td>
</tr>
</tbody>
</table>

Table 1: The vertex generators used to produce the vertices in the constraint network for a global constraint.

In this work we use the three vertex generators that are shown in Table 1. The simplest vertex generator is the Identity generator, generating one
vertex for each item in the list, the vertex being identical to the item. A more complex vertex generator is the \textit{Attribute} generator, which generates one vertex from each item in a list by the extraction of an attribute from the item. In the specification of \texttt{nbdifferences}, we use the vertex generators as follows.

\begin{align*}
\text{Vertex Generators:} & \quad V_1 = \text{Attribute}(\text{VarList, var1}) \\
& \quad V_2 = \text{Attribute}(\text{VarList, var2})
\end{align*}

If \texttt{VarList} is a list with \( n \) tuples, the example would produce \( 2n \) vertices by extracting the attributes \texttt{var1} and \texttt{var2}, naming the first \( n \) vertices \( V_1 \) and the rest of the vertices \( V_2 \). As an example, if

\begin{equation}
\text{VarList} = [(a, b), (c, d), (e, f), (g, h)],
\end{equation}

we would get the two lists of vertices

\begin{align*}
V_1 &= [a, c, e, g], \\
V_2 &= [b, d, f, h].
\end{align*}

### 2.2 Arc Generators

Arc generators produce the directed arcs in the constraint network corresponding to a global constraint. An arc generator takes either one or two lists of vertices, and produces a set of arcs between the vertices. The arc generators used in this work are described in Table 2.

As an example of using arc generators, we now specify the arc generator used in the \texttt{nbdifferences} constraint.

\begin{align*}
\text{Arc Generators:} & \quad A = \text{ProductEq}(V_1, V_2)
\end{align*}

The specification above takes two lists of vertices \( V_1 \) and \( V_2 \), and use the \texttt{ProductEq} arc generator to produce a set \( P \) of arcs between all tuples of variables in the list of tuples of two variables used as argument to the constraint. Suppose that

\begin{align*}
V_1 &= [a, c, e, g], \\
V_2 &= [b, d, f, h].
\end{align*}

We would then get the set of arcs

\begin{equation}
A = \{(a, b), (c, d), (e, f), (g, h)\}.
\end{equation}

This graph is shown in Figure 1.
<table>
<thead>
<tr>
<th>Name</th>
<th>Arcs generated</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Arcs ((v_i, v_i)) on all vertices (v_i).</td>
<td></td>
</tr>
<tr>
<td>Path</td>
<td>Arcs between all adjacent pairs ((v_i, v_{i+1})) of vertices in the list.</td>
<td></td>
</tr>
<tr>
<td>Clique</td>
<td>Each vertex (v_i) in the vertex list is connected to a vertex (v_j) in the vertex list if and only if (i &lt; j).</td>
<td></td>
</tr>
<tr>
<td>Clique</td>
<td>The set of arcs ((v_i, v_j)) where (v_i, v_j) is in the vertex list.</td>
<td></td>
</tr>
<tr>
<td>ProductEq</td>
<td>Arcs between vertex (v_i) in the first vertex list, and vertex (v_i) second list. The lists must have the same length.</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>Arcs from each vertex in the first list to each vertex in the second list.</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The arc generators used to produce the arcs in the constraint network for a global constraint. Each arc generator takes either one or two lists of vertices, and connects these using directed arcs.

2.3 Elementary Constraints

An elementary constraint is a constraint that is applied on the arcs in the constraint network for the global constraint. We use general unary and binary constraints as elementary constraints. Some specialized elementary constraints useful to express global scheduling constraints are shown in Table 3. To continue our discussion of elementary constraints we first need to define tasks.

**Definition 4** A task \(T\) is a tuple \((\text{start : dvar}, \text{end : dvar})\), with the attribute start representing start time of the task, and the attribute end representing
Figure 1: The graph obtained by using the ProductEq arc generator on the two lists of vertices $V_1 = [a, c, e, g], V_2 = [b, d, f, h]$

the end time of the task.

The specialized elementary constraint overlap defines the temporal overlapping relation between tasks $A$ and $B$. The constraint $\text{start-overlap}_{(\leq)}(A, B)$ is the overlapping relation between the start of $A$ and a task $B$. The corresponding relation between the end of a task and another task is called $\text{end-overlap}_{(\leq)}(A, B)$. The constraint $\text{cyclic-start-overlap}_{(\leq, \ell)}(A, B)$ is the corresponding cyclic overlapping relation with cycle length $\ell$. For these constraints, we use the order $\preceq$ to eliminate tasks with the same start as the one we are considering. We must do this in order to get a correct cost function, which is shown later in the discussion of the cumulative-1 constraint.

<table>
<thead>
<tr>
<th>$\text{overlap}(A, B)$</th>
<th>$\equiv \text{start}(A) &lt; \text{end}(B) \land \text{start}(B) &lt; \text{end}(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{start-overlap}_{(\leq)}(A, B)$</td>
<td>$\equiv \text{start}(A) &gt; \text{start}(B) \land \text{start}(A) &lt; \text{end}(B)$ $\lor$ $\text{start}(A) = \text{start}(B) \land B \preceq A \land \text{start}(A) &lt; \text{end}(B)$</td>
</tr>
<tr>
<td>$\text{end-overlap}_{(\leq)}(A, B)$</td>
<td>$\equiv \text{end}(A) \geq \text{start}(B) \land \text{end}(A) &lt; \text{end}(B)$ $\lor$ $\text{end}(A) = \text{start}(B) \land B \preceq A \land \text{end}(A) &lt; \text{end}(B)$</td>
</tr>
<tr>
<td>$\text{cyclic-start-overlap}_{(\leq, \ell)}(A, B)$</td>
<td>$\equiv \text{start-overlap}_{(\leq, \ell)}(A, B) \lor \text{start}(A) &lt; \text{end}(B) - \ell$</td>
</tr>
</tbody>
</table>

Table 3: Specialized elementary constraints.

An elementary constraint is specified in terms of two vertex placeholders [1] and [2], which specify the vertex the arc is leaving and entering respectively. When using the elementary constraint, we also specify a set of arcs
to which the elementary constraint should be applied.

We now continue with an example of the elementary constraints in the specification of the nbdifferences constraint. We express the elementary constraints on the arcs as follows.

\[ \text{Arc Constraints: } A : [1] = [2] \]

This specifies that a binary equality constraint should be applied on all arcs in \( A \).

### 2.4 Graph Properties

The last attributes we need in order to specify global constraints formally are the graph properties we enforce on the final graph obtained when applying an assignment on the elementary constraints.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less((r, k))</td>
<td>( \equiv r &lt; k )</td>
</tr>
<tr>
<td>( C_{\text{Less}} = \max(0, r - k - 1) )</td>
<td></td>
</tr>
<tr>
<td>LessEq((r, k))</td>
<td>( \equiv r \leq k )</td>
</tr>
<tr>
<td>( C_{\text{LessEq}} = \max(0, r - k) )</td>
<td></td>
</tr>
<tr>
<td>Greater((r, k))</td>
<td>( \equiv r &gt; k )</td>
</tr>
<tr>
<td>( C_{\text{Greater}} = \max(0, k - r - 1) )</td>
<td></td>
</tr>
<tr>
<td>GreaterEq((r, k))</td>
<td>( \equiv r \geq k )</td>
</tr>
<tr>
<td>( C_{\text{GreaterEq}} = \max(0, k - r) )</td>
<td></td>
</tr>
<tr>
<td>Equal((r, k))</td>
<td>( \equiv r = k )</td>
</tr>
<tr>
<td>( C_{\text{Equal}} =</td>
<td>k - r</td>
</tr>
<tr>
<td>NotEqual((r, k))</td>
<td>( \equiv r \neq k )</td>
</tr>
<tr>
<td>( C_{\text{NotEqual}} = \begin{cases} 1, &amp; \text{if } r = k \ 0, &amp; \text{otherwise.} \end{cases} )</td>
<td></td>
</tr>
<tr>
<td>ExpLessEq(_{a})((S, k))</td>
<td>( \equiv \delta \leq 0 )</td>
</tr>
<tr>
<td>( C_{\text{ExpLessEq}} = \begin{cases} (\delta(\delta + 1) - \epsilon(\epsilon + 1))/2, &amp; \text{if } \delta &gt; 0 \ 0, &amp; \text{otherwise,} \end{cases} )</td>
<td></td>
</tr>
<tr>
<td>where ( \delta = \left( \sum_{(a, v) \in S} a(v) \right) - k, \epsilon = \max(0, \delta - a(z)), ) and ( z ) is predecessor in ( S ).</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Constraints on numerical functions on sets of arcs, with semantics and cost \( C \).
A property consists of a \textit{property constraint} on a \textit{set expression}. The property constraints we use are simple linear inequalities, shown in Table 4. A set expression is a function from a set to a numerical value. The set expressions we use are described in Table 5.

The property constraints take as argument a numerical value (given by a set expression, Table 5) and a constant. The property constraints checks if the corresponding linear constraint holds. If the linear constraint does not hold, the property constraint gives an appropriate cost for the result of the set expression. For the family of cumulative constraints, the $\text{ExpLessEq}$ property constraint calculates what the cost should have been if the vertices were expanded to the number of identical vertices of the attribute. The $\text{ExpLessEq}$ constraint incorporates an implicit set expression and takes a set of arcs as argument. The set of arcs must be successor arcs leaving the same vertex, and thus $\text{ExpLessEq}$ works only if the $\text{Succ}$ set generator is used. The $\text{ExpLessEq}$ constraint is described in more detail later in the discussion of the cumulative constraint.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Arc Set Expression & Description \\
\hline
$\text{Card}(S)$ & Cardinality of the set $S$ \\
$\text{AttribSum1}(a,S)$ & Sum of the attribute $a$ applied on all vertices $s$ where $(s,t) \in S$. \\
$\text{AttribSum2}(a,S)$ & Sum of the attribute $a$ applied on all vertices $t$ where $(s,t) \in S$. \\
\hline
\end{tabular}
\caption{Arc Set Expressions.}
\end{table}

The set expressions calculate a numerical value for a set of arcs. The input to a set expression is a set of arcs generated by an \textit{set generator}. The set generators take a set of arcs $A$ from the graph and an assignment $v$, and returns a collection of subsets of the arcs in $A$ with elementary constraints attached that are satisfied by $v$. The two set generators we consider in this article are shown in Table 6. We enforce that the properties of the global constraint are satisfied for all arc sets returned from the set generator.

Now again consider the \textit{mbdifferences} global constraint. We want to express that there are at most $k$ satisfied elementary constraints. We can do this using the following specification.

\begin{align*}
\text{Set Generators:} & \quad s = \text{Arcs}(A, v) \\
\text{Graph Constraint:} & \quad \text{LessEq}(\text{Card}(s), k)
\end{align*}
<table>
<thead>
<tr>
<th>Name</th>
<th>Collection of sets of arcs generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arcs</td>
<td>The collection of the set of arcs with satisfied elementary constraints attached in the graph.</td>
</tr>
<tr>
<td>Succ</td>
<td>The collection of sets of outgoing arcs with satisfied elementary constraints attached, for each node in the graph.</td>
</tr>
</tbody>
</table>

Table 6: Arc Set Generators

3 Local Search and Global Constraints

The cost for a global constraint should satisfy the following two properties. First, the cost should be based on primitive constraints. Second, the cost should reflect how many such primitive constraints are violated by the global constraint.

Both these properties must be fulfilled in order for the cost to be comparable with the one for a single primitive constraint, and with other global constraints designed after the same principle. We have decided to use generic unary and binary constraints as the primitive constraint, which we call elementary constraints.

To measure how many elementary constraints are violated, we use the graph properties of a global constraint to obtain a cost. This cost is based on the degree of violation of the property constraint in the global constraint. In Table 4 we have listed the cost for the different property constraints used in our work. Because we apply a property constraint to all arc sets given by the set generator, we take the sum of the costs obtained to get the final cost for the property constraint.

We assume without loss of generality that we have a local search engine that in each iteration changes one single variable to any value in its domain. With a neighbor operator like this, it is meaningless to reason about the distance from satisfaction of a unary or binary constraint. Based on this, we define the cost of each elementary constraint to be 1.

3.1 Incremental Cost Calculation

An absolute requirement of a local search solver is that the evaluation of the neighborhood is efficient. In this paper we evaluate neighbors by applying a cost function. Consider a typical problem where the neighbor function generates a neighborhood of 500 assignments. We assume that we evaluate all neighbors in order to select the best one. If we also assume that the local
search finds a solution in 200 iterations, we have to evaluate \(500 \times 200 = 100,000\) assignments in total. This shows that efficient cost calculation is important for local search.

Incremental cost calculation is a way to reduce computation time by recomputing only the parts of an assignment that differs from a previous assignment, for which a cost is already calculated. In the framework we present, incremental cost calculation for a changed variable \(x\) can be realized for a global constraint by keeping track of the partial cost \(c_i\) calculated for each set \(S_i\) generated by the set generator. In an incremental cost calculation, we reconsider each set \(S_i\) for which \(x\) is part of an arc \(a\). If \(x\) is part of a previously satisfied elementary constraint on \(a\) that is now violated, we reduce \(c_i\) by the amount contributed by the constraint attached to \(a\). If \(x\) is part of a previously violated elementary constraint that has now been violated, we instead increase \(c_i\) by the amount contributed by the constraint attached to \(a\). We can then use the repaired partial costs \(c_i\) to calculate the total cost.

### 3.2 Constraint-Specific Neighborhoods

To further increase the performance of local search, specialized neighborhoods can be used to decrease the size of the neighborhood. For example, in non-overlapping scheduling problems the local transition can be done by placing a task either before or after another task [6], ignoring time points where no improvement could possibly be found.

Constraint-specific neighborhoods can be realized in frameworks based on graph properties and arc constraints. We do this by providing functions that repair elementary constraints. We can then select individual elementary constraints for repair, and use the repairing functions to generate suitable neighbors, which can in turn be evaluated using incremental cost calculation.

As an example of the above, consider the non-overlapping serialized constraint. Assume that the local search procedure annotates each variable with the number of satisfied elementary constraints involving it. Also assume that \(X\) is one of the tasks in the serialized constraint involving most active elementary constraints. The serialized constraint can now pick one of its elementary overlapping constraints that involve \(X\) and is active, and use the repair function to select a suitable neighbor. We also specify in the call to the repair function which variable we want to repair. A possible repair function for a binary overlapping constraint on two tasks \(A\) and \(B\) could move either \(A\) or \(B\) so that \(A\) ends exactly as \(B\) starts, or vice versa.
4 Global Constraint Representation

In this section, we show the generality and usability of our approach by giving representations of ten important global constraints using the framework we have proposed. We also show how to efficiently calculate the cost by giving examples, and how to use incremental cost calculations to further reduce the cost computation time.

4.1 The alldifferent constraint

The global constraint alldifferent restricts a set of variables from taking the same values. We can express an alldifferent constraint on \( n \) variables semantically with \( n(n - 1)/2 \) binary inequalities, restricting each possible pair of unique variables from taking the same value. Using this observation, we base our model (shown in Table 4.1) of the alldifferent constraint on binary equalities between the pairs of variables in the constraint.

<table>
<thead>
<tr>
<th>Constraint:</th>
<th>alldifferent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>VarList : list dvar</td>
</tr>
<tr>
<td>Vertex Generators:</td>
<td>( V = \text{Identity}(\text{VarList}) )</td>
</tr>
<tr>
<td>Arc Generators:</td>
<td>( A = \text{CliqueLt}(V) )</td>
</tr>
<tr>
<td>Arc Constraints:</td>
<td>( A : [1] = [2] )</td>
</tr>
<tr>
<td>Set Generators:</td>
<td>( L = \text{Arcs}(A, v) )</td>
</tr>
<tr>
<td>Graph Properties:</td>
<td>( \forall S \in L : \text{Equal(Card}(S), 0) )</td>
</tr>
</tbody>
</table>

Table 7: The alldifferent constraint.

We use the \textit{Identity} vertex generator to construct vertices directly from the arguments (variables) given to the constraint. Using the \textit{CliqueLt} arc generator we get arcs between each unique vertex. We use an equality constraint as the elementary constraint, because we want to count the number of equal variable pairs to see if the constraint is violated or satisfied – if the number of satisfied equalities is zero, then the global alldifferent constraint is satisfied. We can express this using the \textit{Equal} property constraint on the cardinality of the set of satisfied equality arcs.

If the constraint is violated, the cost for the constraint will be the number of equal variable pairs, according to the definition of the cost for the \textit{Equal} set constraint in Table 4.

As an example of the cost calculation, consider an alldifferent constraint on a list \([a, b, c, d, e, f]\) of variables. Using the \textit{Identity} vertex gener-
ator we get the list
\[ V = [a, b, c, d, e, f] \]
of vertices. The arc generator \textit{CliqueLt} generates the following set of arcs.
\[ A = \{(a, b), (a, c), (a, d), (a, e), (a, f), (b, c), (b, d), (b, e), (b, f), (c, d), (c, e), (c, f), (d, e), (d, f), (e, f)\}. \]

On the arcs in \( A \) we apply the binary equality constraint \([1] = [2]\). Suppose now that we have the assignment below.
\[ v(x) = \{a \mapsto 1, b \mapsto 1, c \mapsto 2, d \mapsto 3, e \mapsto 3, f \mapsto 3\}. \]
The set generator application \( L = \text{Arcs}(A, v) \) gives the following collection of sets of arcs.
\[ L = \{ \{(a, b), (d, e), (d, f), (e, f)\} \}. \]
The set expression \( \text{Card}(S) \) yields a result of 4. Inserting this in the graph property \( \forall S \in L : \text{Equal}(\text{Card}(S), 0) \) gives the cost \( C = \lvert 0 - 4 \rvert = 4 \).

Now we consider an incremental cost calculation for this constraint. Suppose \( e \) is changed to 3. We have two arcs \((a, c), (b, c)\) entering and three arcs \((c, d), (c, e), (c, f)\) leaving \( c \). The constraints on the arcs entering \( c \) was and stays violated, because \( a \neq c \) and \( b \neq c \). The constraints on the arcs leaving \( c \) was violated, but now gets satisfied. This means that we increase the cost of the single arc set with 3, which gives a total cost of 7 for the \textit{allDifferent} constraint. Because a vertex can have at most \( n - 1 \) arcs connected to it, where \( n \) is the number of edges, we can update the cost incrementally in \( O(n) \) time.

### 4.2 The capa constraint
The \textit{capa} constraint [3] takes a list of tuples \( \text{TupList} = [(\text{var}_1, w_1), \ldots, (\text{var}_p, w_p)] \), where \( \text{var}_i \) is a variable and \( w_i \) a corresponding integer weight, and two integers \( a \) and \( k \). The constraint checks that the sum of the weights of the variables taking the value \( a \) is less than or equal to \( k \). We can state the constraint as a loop graph with the unary constraint \( \text{var}([1]) = a \). We use the \textit{Loop} arc generator to do this.
Table 8: The capa constraint.

4.3 The nbdifferences constraint

The nbdifferences constraint [3] takes a list of tuples of two variables

\[ \text{TupList} = [\langle \text{var}_1, \text{var}_2 \rangle, \ldots, \langle \text{var}_n, \text{var}_{2n} \rangle] \]

and an integer k as arguments, and checks that the number of tuples taking the same value is less than or equal to k.

\[ |\langle x, y \rangle \in S : v(x) = v(y)| \leq k \]

We can represent the constraint as a graph with the variables as vertices, and arcs with an associated equality constraint \([1] = [2]\) between the tuples. We use the Attribute vertex generator, and the ProductEq arc generator to form the graph.

Table 9: The nbdifferences constraint.
4.4 The ordered-tasks constraint

The \texttt{ordered-tasks} constraint on a set of \( n \) tasks handles a set of \( n - 1 \) binary precedence relations between tasks on the form

\[
\text{end}(T_i) + d_i \leq \text{start}(T_{i+1}),
\]

where \( d_i \) is a constant. The global constraint on \( n \) tasks is equivalent to the \( n - 1 \) following constraints.

\[
\begin{align*}
\text{end}(t_1) & + d_1 \leq \text{start}(t_2) \\
\text{end}(t_2) & + d_2 \leq \text{start}(t_3) \\
& \vdots \\
\text{end}(t_{n-1}) & + d_{n-1} \leq \text{start}(t_n)
\end{align*}
\]

We handle this constraint by building a path including all vertices using the \texttt{Path} arc generator. The vertices are produced using the \texttt{Identity} vertex generator. We add anti-precedence constraints on the arcs. Using the \texttt{Equality} property constraint ensure that no anti-precedence constraints are satisfied in order for the global constraint to be satisfied.

```
Constraint:       ordered-tasks
Arguments:       T: list (t: Task, shift: int)
                    Task :: (start: dvar, end: dvar)
Vertex Generators: V = Identity(T)
Arc Generators:   A = Path(V)
Arc Constraints:  A : end(t([1])) + t([1]).d > start(t([2]))
Set Generator:   L = Arcs(A, v)
Graph Properties: \forall S \in L : Equal(Card(S), 0)
```

Table 10: The \texttt{ordered-tasks} constraint, which ensures that a list of tasks is ordered as in the order taken from the list, and with a time distance of \texttt{shift(t)} between the execution of task \( t \) and task \( t + 1 \).

4.5 The serialized constraint

The \texttt{serialized} global constraint ensures that the tasks in a list do not overlap in time. A \texttt{serialized} constraint on \( n \) tasks is semantically equivalent to \( n(n - 1)/2 \) non-overlapping constraints, restricting all possible pairs of tasks from overlapping. We use a negated variant of this as the basis for our model of the \texttt{serialized} constraint.
Constraint: \texttt{serialized} \\
Arguments: Tasks : list \langle \texttt{start:dvar}, \texttt{end:dvar} \rangle \\
Vertex Generators: \( V = \texttt{Identity}(\text{Tasks}) \) \\
Arc Generators: \( A = \texttt{CliqueLt}(V) \) \\
Arc Constraints: \( A : \texttt{overlap}([1], [2]) \) \\
Set Generator: \( L = \texttt{Arcs}(A, v) \) \\
Graph Properties: \( \forall S \in L : \texttt{Equal}(\text{Card}(S), 0) \)

Table 11: The \texttt{serialized} constraint.

The representation of \texttt{serialized} as graph properties, shown in Table 11, is similar to the one used for the \texttt{alldifferent} constraint. We use the elementary constraint \texttt{overlap} (shown in Table 3) as arc constraint between the vertices. As vertices of the constraint graph, we use the tasks themselves, using the \texttt{Identity} vertex generator. Again, we use the \texttt{CliqueLt} arc generator, and apply the \texttt{overlap} arc constraint on the arcs. The rest of the formulation of \texttt{serialized} is in analogy with the model for \texttt{alldifferent}.

4.6 The disjoint-tasks constraint

The \texttt{disjoint-tasks} constraint takes two lists of tasks, and ensures that no task in the first list intersects with a task in the second list. The formulation of the constraint as graph properties on a structured network is straightforward and shown in Table 12. We use the \texttt{Identity} vertex generator to produce one vertex for each task. The arc generator used is \texttt{Product}, which produces an arc between each vertex in the first list and each vertex in the second list. We again use \texttt{overlap} as the elementary constraint, and counts the number of overlaps as a measure on the cost of the constraint. The \texttt{disjoint-tasks} constraint is a variant of the \texttt{serialized} constraint [1].

4.7 The cumulative-1 constraint

In this section, we generalize the \texttt{serialized} constraint to a restricted \( k \)-capacity cumulative constraint called \texttt{cumulative-1}, where the resource can handle at most \( k \) tasks at the same time. The semantics of the constraint prevents more than \( k \) tasks from executing at each time point.

\[
\text{cumulative-1}(T, k) \equiv \\
\forall t \in \{\text{min}(\text{start}(T)), \text{max}(\text{end}(T))\} : \\
|\{t | t \in T \land \text{active-at}(t, i)\}| \leq k
\]
<table>
<thead>
<tr>
<th>Constraint:</th>
<th>disjoint-tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>Tasks1: list (start:dvar,end:dvar)</td>
</tr>
<tr>
<td></td>
<td>Tasks2: list (start:dvar,end:dvar)</td>
</tr>
<tr>
<td>Vertex Generators:</td>
<td>$V_1 = \text{Identity}(\text{Tasks1})$</td>
</tr>
<tr>
<td></td>
<td>$V_2 = \text{Identity}(\text{Tasks2})$</td>
</tr>
<tr>
<td>Arc Generators:</td>
<td>$A = \text{Product}(V_1,V_2)$</td>
</tr>
<tr>
<td>Arc Constraints:</td>
<td>$A : \text{overlap}([1],[2])$</td>
</tr>
<tr>
<td>Set Generator:</td>
<td>$L = \text{Arcs}(A,v)$</td>
</tr>
<tr>
<td>Graph Properties:</td>
<td>$\forall S \in L : \text{Equal(Card}(S),0)$</td>
</tr>
</tbody>
</table>

Table 12: The disjoint-tasks constraint.

We note that the only time points where the resource usage will increase is at the start times of the tasks. Because every task overlapping with a single time point is overlapping with each other, we can state the semantics of cumulative-1 as follows.

$$\text{cumulative-1}(T,k) \equiv$$

$$\forall s \in T : \left| \{ t | t \in T \land \text{start-overlap}_{(\leq)}(s, \text{start}(t)) \} \right| \leq k$$

This ensures that for each start time point of a task, the number of overlapping tasks should be less than or equal to $k$. The total order $\leq$ is arbitrary and of no consequence for the semantics of the constraint.

The cumulative-1 constraint is formulated using graph properties in Table 13. We use a cost for cumulative-1 based on the number of intersections between tasks, reduced by the allowed intersections between tasks. This cost is similar to the one used for serialized. We produce the vertices of the constraint using the Identity vertex generator, and use the Clique arc generator to get a graph with arcs between any pair $(i, j)$ of vertices. We apply the start-overlap constraint on the arcs, and use the set generator Succ to get all sets of successor arcs. We then apply a LessEq constraint on the cardinality of these sets to form the graph property.

Tasks starting at the exact same time present a problem for the cost calculations. The problem is that if $p$ tasks start at the same time, we get $p$ identical successor vertices. This gives a cost that is incorrect. To see this, assume a capacity $k = 1$ and consider Figure 2. The leftmost setup yields a cost of $2 + 2 + 2 = 6$, and the rightmost gives a cost of $0 + 1 + 2 = 3$. The situations should be considered equal since all three tasks intersect with each other.

Two corrections of this problem are possible. First, we may prohibit two tasks from having the same start time. This is obviously not an elegant
solution. Second, we may impose an order \( \leq \) on the tasks, and include a task \( B \) with start time equal to the predecessor task \( A \) in the successor arcs set if and only if \( B \leq A \). We will use this approach. We include an order in some of the overlapping constraints in Table 3 to correct this problem, and use the order from the list of tasks generated from the vertex generator.

![Diagram](image)

**Figure 2:** Two equivalent schedules which yields different cost unless corrected.

<table>
<thead>
<tr>
<th>Constraint:</th>
<th>cumulative-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>Tasks: list ( \langle \text{start:dvar,end:dvar}\rangle, k: \text{int} )</td>
</tr>
<tr>
<td>Restrictions:</td>
<td>( \forall t \in \text{Tasks}: \text{start}(t) &lt; \text{end}(t) )</td>
</tr>
<tr>
<td>Vertex Generators:</td>
<td>( V = \text{Identity}(\text{Tasks}) )</td>
</tr>
<tr>
<td>Arc Generators:</td>
<td>( A = \text{Clique}(V) )</td>
</tr>
<tr>
<td>Arc Constraints:</td>
<td>( A : \text{start-overlap}_V([1],[2]) )</td>
</tr>
<tr>
<td>Set Generator:</td>
<td>( L = \text{Succ}(A,v) )</td>
</tr>
<tr>
<td>Graph Properties:</td>
<td>( \forall S \in L : \text{LessEq}(\text{Card}(S), k) )</td>
</tr>
</tbody>
</table>

**Table 13:** The cumulative-1 constraint.

We now demonstrate the cost calculation for a cumulative-1 constraint with a capacity of \( k = 2 \) on a list of tasks \( [A, B, C, D, E, F] \). We use the Identity vertex generator and get the vertex list \( V = [A, B, C, D, E, F] \). The arc generator Clique generates a set of arcs \( \mathcal{A} = \{(x,y)|x,y \in V\} \).

\[
\mathcal{A} = \{ (A, A), (A, B), (A, C), (A, D), (A, E), (A, F),
        (B, B), (B, C), (B, D), (B, E), (B, F),
        (C, C), (C, D), (C, E), (C, F),
        (D, D), (D, E), (D, F),
        (E, E), (E, F),
        (F, F) \}. 
\]
On $A$ we apply the binary constraint $\text{start-overlap}_V([1, 2])$. Now suppose we have the total assignment of tasks with the following start- and end times.

<table>
<thead>
<tr>
<th>Task</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>end</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>13</td>
</tr>
</tbody>
</table>

The execution of these tasks are shown in Figure 3. The set generator $\text{Succ}(A, v)$ yields the following collection of sets of arcs.

![Figure 3: An example of an infeasible schedule for a cumulative-1 constraint with capacity 2.](image)

$$L = \{ (A, A), (B, A), (B, B), (C, A), (C, C), (D, C), (D, D), (E, C), (E, D), (E, E), (F, D), (F, E), (F, F) \}.$$  

Applying the set expression $\text{Card}(S)$ on all sets in $L$ yields the results $\{1, 2, 2, 2, 3, 3\}$. If we insert these results into the graph property $\text{LessEq}(\text{Card}(S), k)$ where $k = 2$, we get the set of costs $\{0, 0, 0, 0, 1, 1\}$. The total cost of the cumulative-1 constraint thus becomes $1 + 1 = 2$.

We now consider incremental updates of the cost of a cumulative-1 constraint. Suppose that the start time of task $B$ is moved to 3. We have six arcs entering and six arcs leaving $B$. No arcs leaving $B$ will change their state. The arc leaving $C$ and entering $B$ will become satisfied. This arc is part of the successor arc set for task $C$, so this partial cost will be increased by one. The new set expression result for this set will be 3, and inserting this into the graph property yields a new cost for this set of 1. The new total cost of the cumulative-1 constraint thus becomes 3. Each vertex is connected by $n$ arcs leaving it and $n$ entering it, where $n$ is the number of vertices. We can therefore update the cost incrementally in $O(2n)$ time.
4.8 The cumulative constraint

The cumulative constraint is a generalization of the cumulative-1 constraint, where we associate a nonnegative height with each task. The constraint ensures for the given list of tasks that for any point in time, the sum of the heights of the tasks executing at that time is less than or equal to the capacity $k$ of the constraint.

$$cumulative(T) \equiv \forall i \in \{\min(start(T)), \max(end(T))\} : \sum_{t \in T: overlap(t,i)} height(t) \leq k$$

One way of handling the generalization of the cumulative constraint is to split each task with a height $h$ greater than 1 into $h$ different tasks, with equal start and end times, but with height 1. If we do this, we can handle the constraint in exactly the same way as we did for the cumulative-1 constraint. Formulation of the cumulative constraint using this approach is shown in Table 14, were we use a similar approach as for the cumulative-1 constraint. The difference is that the tuples in the argument list has been extended with a height attribute, and that we use the Expand vertex generator to produce $h$ vertices for a task with height $h$.

<table>
<thead>
<tr>
<th>Constraint:</th>
<th>cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>$T : list\ Task, k : int$</td>
</tr>
<tr>
<td></td>
<td>Task :: $\langle\text{start, dvar, end, dvar, height: int}\rangle$</td>
</tr>
<tr>
<td>Restrictions:</td>
<td>$\forall t \in T : height(t) \geq 0$</td>
</tr>
<tr>
<td></td>
<td>$k \geq 0$</td>
</tr>
<tr>
<td>Vertex Generators:</td>
<td>$V = \text{Expand}(\text{Tasks, } M)$ where</td>
</tr>
<tr>
<td></td>
<td>$M(\langle s, e, h \rangle) = [\langle s, e, 1 \rangle, \ldots, \langle s, e, 1 \rangle]^{h \text{ items}}$</td>
</tr>
<tr>
<td>Arc Generators:</td>
<td>$A = \text{Clique}(V)$</td>
</tr>
<tr>
<td>Arc Constraints:</td>
<td>$A : start-overlap_{\gamma}([1], [2])$</td>
</tr>
<tr>
<td>Set Generator:</td>
<td>$L = Succ(A, \nu)$</td>
</tr>
<tr>
<td>Graph Properties:</td>
<td>$\forall S \in L : \text{LessEq}(\text{Card}(S), k)$</td>
</tr>
</tbody>
</table>

Table 14: The cumulative constraint, where each task $A \in T$ with height $w_A$ is expanded into $w_A$ different tasks with same start- and end-time, and with height 1. This constraint is then handled exactly as a cumulative-1 constraint.
4.8.1 The ExpLessEq property constraint

The downside of this expansion is that the number of tasks we have to handle increase. This can affect performance significantly. Therefore, we propose to handle the cumulative constraint by the specialized property constraint ExpLessEq\(_a\). The semantics of this property constraint is the same as for the LessEq property constraint, but the cost calculated is the number of elementary constraints that would be satisfied if the tasks were expanded in the way described above. Instead of taking the numerical result of a set expression, the constraint takes a set as argument, and implicitly calculates the result of the set expression AttrSum\(_2\) and the attribute \(a\).

<table>
<thead>
<tr>
<th>Constraint:</th>
<th>cumulative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments:</td>
<td>(T : \text{list Task}, k : \text{int})</td>
</tr>
<tr>
<td></td>
<td>Task :: ((\text{start:dvar}, \text{end:dvar}, \text{height:int}))</td>
</tr>
<tr>
<td>Restrictions:</td>
<td>(\forall t \in T : \text{height}(t) \geq 0)</td>
</tr>
<tr>
<td></td>
<td>(k \geq 0)</td>
</tr>
<tr>
<td>Vertex Generators:</td>
<td>(V = \text{Identity}(T))</td>
</tr>
<tr>
<td>Arc Generators:</td>
<td>(A = \text{Clique}(V))</td>
</tr>
<tr>
<td>Arc Constraints:</td>
<td>(A : \text{start-overlap}_V([1], [2]))</td>
</tr>
<tr>
<td>Set Generator:</td>
<td>(L = \text{Succ}(A, v))</td>
</tr>
<tr>
<td>Graph Properties:</td>
<td>(\forall S \in L : \text{ExpLessEq}_{\text{height}}(S, k))</td>
</tr>
</tbody>
</table>

Table 15: The cumulative constraint, where the expansion of the tasks is handled by the specialized ExpLessEq set constraint. The ExpLessEq calculates the contribution that each task \(t\) would have if it were expanded into height\((t)\) different tasks.

We now explain how the ExpLessEq property calculates a cost for a set of arcs. We assume that we have expanded the vertices \(V\) in question into a new set \(W\), and that the order that holds for the original set of vertices holds for the new set of vertices as well, i.e., \(v_i <_L v_j \iff \forall s, t : w_{is} <_L w_{jt}\) where \(v_i\) and \(v_j\) denotes original vertices, and \(w_{is}\) and \(w_{jt}\) denote vertices expanded from \(v_i\) and \(v_j\) respectively. We also assume that the order \(<_L\) is total, so that the new vertices expanded from one single vertex has an internal total order. The order \(<_L\) is arbitrary.

Assume we have a set of successor arcs

\[
S = \{(z, z), (z, b), (z, c), (z, d), (z, e)\}
\]

If the sum of the attribute \(a\) of the successor vertices \{\(z, b, c, d, e\}\) is less than or equal to \(k\), the cost is zero by definition. However, if the sum is greater
than $k$, we set

$$\delta = \left( \sum_{(u,v) \in S} a(v) \right) - k,$$

where $a$ is the attribute of the ExpLessEq property. In the expanded graph, we would have $a(z)$ different sets of successors (due to the expansion of the predecessor vertex), each with between $\delta + k - a(z)$ and $\delta + k$ successor vertices, due to the total order of the vertices. More specifically, the cost contribution of the set of successors with origin in $z$ will be the $a(z)$ last terms of the expression

$$1 + 2 + 3 + \cdots + \delta = \frac{\delta(\delta + 1)}{2},$$

because we allow $k$ arcs to be satisfied without cost. We can calculate the $a(z)$ last terms by subtraction of the first $\epsilon = \max(0, \delta - a(z))$ terms, i.e.,

$$(\delta(\delta + 1) - \epsilon(\epsilon + 1))/2.$$

The cost for the ExpLessEq$_a$ property constraint given an attribute $a$, a set of successor arcs $S$ and a constant $k$ is therefore defined as follows.

$$\mathcal{C}_{\text{ExpLessEq}_a} = \begin{cases} 
(\delta(\delta + 1) - \epsilon(\epsilon + 1))/2, & \text{if } \delta > 0 \\
0, & \text{otherwise},
\end{cases}$$

where $\delta = \left( \sum_{(u,v) \in S} a(v) \right) - k$, $\epsilon = \max(0, \delta - a(z))$ and $z$ is predecessor of $S$. This approach avoids the negative effect on performance resulting from explicit expansion of the tasks, and instead calculates the result of cost calculation on this expanded set of tasks. This new formulation of the cumulative constraint is shown in Table 15.

We now demonstrate the cost calculation for a cumulative constraint with a capacity of $k = 3$ on a list of tasks $[A, B, C, D, E, F]$. We use the Identity vertex generator and get the vertex list

$$V = [A, B, C, D, E, F].$$

The arc generator Clique generates a set of arcs $\mathcal{A} = \{(x, y) | x, y \in V\}$.

$$\mathcal{A} = \{(A, A), (A, B), (A, C), (A, D), (A, E), (A, F),
(B, B), (B, C), (B, D), (B, E), (B, F),
(C, C), (C, D), (C, E), (C, F),
(D, D), (D, E), (D, F),
(E, E), (E, F),
(F, F)\}.$$
On $A$ we apply the binary constraint $\text{start-overlap}_V([1, [2])$. Now suppose we have the total assignment of tasks with the following attributes.

<table>
<thead>
<tr>
<th>Task</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>end</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>height</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

The execution of these tasks are shown in Figure 4. The set generator

![Diagram](image)

Figure 4: An example of an infeasible schedule for the cumulative constraint with capacity 3.

$\text{Succ}(A, v)$ yields the following collection of sets of arcs.

$$L = \{(A, A)\},$$
$$\{(B, A), (B, B)\},$$
$$\{(C, A), (C, C)\},$$
$$\{(D, C), (D, D)\},$$
$$\{(E, C), (E, D), (E, E)\},$$
$$\{(F, D), (F, E), (F, F)\} \}.$$

We now apply the set expression

$$\text{ExpLessEq}(S)_{\text{height}}(S, k)$$

with $k = 3$ on all sets in the collection $L$. We get the following results when we calculate $\delta$.

$$\delta_A = \text{height}(A) - k = -1$$
$$\delta_B = \text{height}(A) + \text{height}(B) - k = 2$$

25
\[
\begin{align*}
\delta_C &= \text{height}(A) + \text{height}(C) - k = 0 \\
\delta_D &= \text{height}(C) + \text{height}(D) - k = 1 \\
\delta_E &= \text{height}(C) + \text{height}(D) + \text{height}(E) - k = 2 \\
\delta_F &= \text{height}(D) + \text{height}(E) + \text{height}(F) - k = 6
\end{align*}
\]

This form the basis for the cost calculations following.

\[
\begin{align*}
C_A &= 0 \\
C_B &= (2 \times 3 - 0)/2 = 3 \\
C_C &= 0 \\
C_D &= (1 \times 2 - 0)/2 = 1 \\
C_E &= (2 \times 3 - 1 \times 2)/2 = 2 \\
C_F &= (6 \times 7 - 1 \times 2)/2 = 20
\end{align*}
\]

We now take the sum of the successor costs calculated above and get a cost of 26 for the constraint.

Incremental cost calculation for the cumulative constraint can be realized in \(O(2n)\) time. We show this by an example. Assume that we have the cost from the previous example, and that the start time of task \(F\) changes to 13. We have six arcs entering and six arcs leaving \(F\). The arc \((F,D)\) becomes violated, and no other arc will be affected. The partial cost of the successor arc set of \(F\) decreases by the height of \(D\) which is 3. This gives the new value \(\delta_F = 3\), and the new cost \(C_F = (3 \times 4 - 0)/2 = 6\). This give a total cost decrease of 14, and a new total cost for the cumulative constraint of 12. Each vertex have \(2n\) arcs connected to it, where \(n\) is the number of vertices. We can thus update the cost incrementally in \(O(2n)\) time.

### 4.9 The cumulative constraint with cyclic time

We can further extend the family of cumulative constraints with a cyclic cumulative constraint called cyclic-cumulative. The constraint ensures that for all time-points \(0 \leq i \leq \ell\), where \(\ell\) is the cycle length, the sum of the heights of all tasks running at time \(i\) is less than or equal to the capacity \(k\) of the constraint. In addition, any task \(A\) ending after the cycle length should wrap around, and be considered to end its execution at time \(\text{end}(A) - \ell\).

We describe the cyclic-cumulative constraint in Table 16. We can model the constraint by using the cyclic-start-overlap arc constraint, described in Table 3, which handles the eventual wrap around of tasks in time. The formulation is otherwise in analogy to the one for the cumulative constraint shown in Table 15.
4.10 The cumulative constraint with negative heights

We further extend our family of cumulative scheduling constraints with one handling both positive and negative heights associated to the tasks. We do this by extending our reasoning to end times of the tasks as well as the start times. We begin by generating two lists of vertices $V_1$ and $V_2$, both using the Identity vertex generators. We then use the Clique arc generator on $V_1$ to connect every vertex in $V_1$ with each other vertex in $V_1$. We use the Product arc generator on $V_2$ and $V_1$ to connect every vertex in $V_2$ to every vertex in $V_1$. On the first set of arcs we attach a start-overlap constraint, and on the second set we attach an end-overlap constraint. We then use the Succ set generator on the union of arcs to get all successor arcs for all start times and end times of the tasks. We can then apply the ExpLessEq property on the set of successor arcs in analogy with how we did in the formulation of cumulative in Table 15.

5 Conclusions and Future Work

We have presented a new general approach for the design of global constraints usable in local search and heuristics for complete search procedures. We have use a representation of a global constraint as a graph property on a structured network of elementary constraints, and showed how to efficiently compute a cost for a constraint represented in this way. We also gave a method for doing incremental cost calculations and showed that it is possible to create specific neighbor functions for a global constraint, further increasing the efficiency. To demonstrate the generality and usability of our approach we
Constraint: \texttt{negative-cumulative}

Arguments: $T : \text{list Task,}k : \text{int}$

Task :: \langle start,dvar,end,dvar,height:int \rangle

Vertex Generators: $V_1 = \text{Identity}(T)$

$V_2 = \text{Identity}(T)$

Arc Generators: $A_1 = \text{Clique}(V_1)$

$A_2 = \text{Product}(V_2, V_1)$

Arc Constraints: $A_1 : \text{start-overlap}_{(V)}([1], [2])$

$A_2 : \text{end-overlap}_{(V)}([1], [2])$

Set Generators: $L = \text{Succ}(A_1 \cup A_2, v)$

Graph Properties: $\forall S \in L : \text{ExpLessEq}_{\text{height}}(S, k)$

Table 17: The \texttt{negative-cumulative} constraint.

showed how to represent ten important global constraints using our method. We believe that this work further increases the usability of general local search algorithms.

We plan to proceed with this work by implementing a full local search system, capable of modeling the global constraints in this work and more, and to give empirical results showing the accuracy, generality and efficiency of our approach. To conclude, we think that global constraints in local search can significantly increase the usability of local search methods for modeling and solving of hard combinatorial problems in scheduling, planning and other related areas.

References


