Constraint Hierarchy Networks

Roman Barták*

Department of Theoretical Computer Science
Charles University
Malostranské náměstí 2/25
Praha 1, Czech Republic

e-mail: `bartak@kti.mff.cuni.cz`
URL: `http://kti.ms.mff.cuni.cz/~bartak/`
phone: +420-2 2191 4242
fax: +420-2 2191 4323

Abstract

Constraint hierarchies have been proposed to solve over-constrained systems of constraints by specifying constraints with hierarchical preferences, i.e., hard and soft constraints. While the hard (required) constraints must hold, the soft (preferential) constraints should be satisfied as much as possible depending on the criterion used. Currently, there exists a bank of constraint hierarchy solvers/satisfaction algorithms that could be classified into two categories as refining and local propagation algorithms. The local propagation algorithms are fast but limited to equality (functional) constraints while the more general refining algorithms are not incremental.

In this paper we combine advantages of both refining and local propagation methods in a generalized framework for solving constraint hierarchies. Our approach extends ideas of local propagation, i.e. the construction of a constraint network and subsequent propagation of values, however, it suppresses limits of local propagation like restriction to LPB comparator and functional constraints. We introduce a new notion of a constraint cell and we generalize the concept of constraint hierarchy network here. Finally, we present two algorithms for the construction of constraint networks which are instances of our framework. The proposed framework fits in our other concept of plug-in architecture of constraint hierarchy solvers.

Keywords: constraint hierarchy, constraint cell, hierarchy network, constraint planning

* Partially supported by the Grant Agency of Czech Republic under the contract No 201/96/0197.

1 Introduction

Constraint hierarchies were introduced for describing over-constrained systems of constraints by specifying constraints with hierarchical strengths or preferences[1]. It allows one to specify declaratively not only the constraints that are required to hold, but also weaker, so called soft constraints at an arbitrary but finite number of strengths. Weakening the strength of constraints helps to find a solution of previously over-constrained system of constraints. Intuitively, the hierarchy does not permit to the weakest constraints to influence the result. Moreover, constraint hierarchies allow "relaxing" of constraints with the same strength by applying, e.g., weighted-sum, least-squares or similar comparators[2]. The theory of constraint hierarchies was developed in [6] and it is also described in [8,17].

The existence of efficient satisfaction algorithms is an important aspect of constraint hierarchies. Most current satisfaction algorithms, in other words constraint hierarchy solvers, can be classified into two groups: algorithms based on refining method and local propagation algorithms. However, there are also other algorithms, e.g., IHCS [12], which don't fit into any of above mentioned groups.

The refining algorithms solve constraint hierarchy in a straightforward manner by completely satisfying the strongest level first and then weaker levels successively. Thus, the refining method can be used for solving all constraint hierarchies using arbitrary comparator. The generality of the refining method is paid off by losing effectiveness as each constraint level has to be solved at a clap. In particular, the refining method does not support incremental update of the solution after adding or retracting a constraint respectively.

The refining method was first used in a simple interpreter for HCLP programs [8] and it is also employed in the DeltaStar algorithm [17] and in a hierarchical constraint logic programming language CHAL. We show later (Section 3) that our generalized framework for solving constraint hierarchies covers the refining method as well.

The local propagation is a fast incremental method for resatisfying constraints in hierarchies. Basically, it is efficient because it solves uniquely one selected constraint in every step (executing phase). In addition, when a variable is repeatedly updated, e.g., by user operation, it can easily evaluate only the necessary constraints to get a new solution. This straightforward execution phase is paid off by a foregoing planning phase that choose the order of constraints to satisfy via construction of a constraint network. Many local propagation algorithms were developed for special purposes (DeltaBlue [14], SkyBlue [13], QuickPlan [16], DETAIL [10], Houria [9], Indigo [5]).

I our previous papers [2,4] we identified the following limits of local propagation:

- solving conflicts among constraints is sometimes inappropriate
- local propagation cannot handle cycles of constraints
- local propagation works only with equality (functional) constraints
- local propagation supports only locally predicate better comparators
- local propagation cannot find multiple solutions,

and we proposed a generalized framework for solving constraint hierarchies that addresses the above problems by introducing constraint cells, i.e., chunks of constraints that has to be solved together. In [3,4] we gave a formal theory of solving constraint hierarchies by means of constraint cells. This theory shares ideas with the generalized local propagation by Hosobe, Matsuoka and Yonezawa [11] and it provides the completeness and soundness theorems.

---

1 The other methods for describing over-constrained systems include the "partial CSP" model by Freuder & Wallace, the "fuzzy CSP" model by Fargier, and the "probabilistic CSP" model by Fargier & Lang.

2 Comparator is a relation comparing valuations of variables respecting the hierarchy. Currently, there are three widely used groups of comparators, namely locally-better, regionally-better and globally-better comparators [17].

In this paper we concentrate on practical aspect of using constraint cells to solve constraint hierarchies. In particular, we describe the concept of constraint hierarchy network that defines the partial order of constraint cells for propagation of values. We also give a formal description of algorithms for construction of constraint hierarchy networks.

The paper is organized as follows. In Section 2, we introduce formally the constraint hierarchy networks and we define constraint cells and hierarchy networks in an "implementation" manner. Section 3 is dedicated to algorithms for construction of constraint hierarchy networks. We present here simple algorithm that constructs hierarchy networks corresponding to the refining method as well as more sophisticated algorithm that constructs more structured networks with smaller constraint cells. In Section 4, we sketch briefly how the values are propagated through the network to find the solution of the hierarchy. The paper is concluded with some final remarks on the proposed framework and with description of future research.

2 Constraint Hierarchy Networks

In [2] we proposed to combine advantages of both refining and local propagation methods to get a generalized framework for solving constraint hierarchies. This framework is based on idea of dividing the constraint hierarchy into small chunks of constraints, called constraint cells. These cells are partially ordered in a constraint hierarchy network that reflects the relationships among constraints. To get the solution of the hierarchy, i.e., the valuation of variables, an execution algorithm propagates values through the cells using the partial order of cells defined by the network. Each time, the constraints in a cell are solved together to get the best solution according to the comparator chosen.

While the propagation of values and the partial order of cells defined by the hierarchy network correspond to the local propagation concept, solving all constraints in a cell together matches the idea behind the refining method. Hence, the proposed concept of solving constraint hierarchies is enough efficient and satisfactory general at the same time.

2.1 Constraint Cells

The basic part of our generalized framework for solving constraint hierarchies is a notion of a constraint cell. The constraint cell is a chunk of equally preferred constraints which are solved together during the executing phase. We associate a set of output variables with each constraint cell to reflect the variables which are determined/computed by the constraints in the cell (in an obvious manner of local propagation).

DEFINITION 1: (constraint cell)

1) Let C be a finite non-empty set of labeled constraints with the same preference and V be a set of all variables in constraints from C. Let In,Out$\subseteq$V be arbitrary sets of variables such that In$\cup$Out=V and In$\cap$Out=$\emptyset$. We define a *constraint cell* as a triple (C,In,Out). We also enable a constraint cell in the form $(\{\},\{\},\{v\})$ containing only the output variable *v*.
2) We call the sets In and Out from the constraint cell (C,In,Out) *input* and *output variables* respectively.
3) We also say that the constraint cell (C,In,Out) *determinates* each variable from the set Out.

By enabling more constraints in a cell one can easily handle conflicts between constraints with the same strength as well as cycles of constraints. In addition, encapsulating constraints into a cell enables usage of more types of comparators including globally-better ones. Finally, while the traditional local propagation requires all constraints to be functions, i.e., to be able to uniquely compute values of output variable(s) when the values of input variables are given, the proposed constraint cells can contain other types of constraints, e.g., inequalities. The Figure 1 demonstrates the above mentioned advantages of constraint cells over the traditional local propagation.

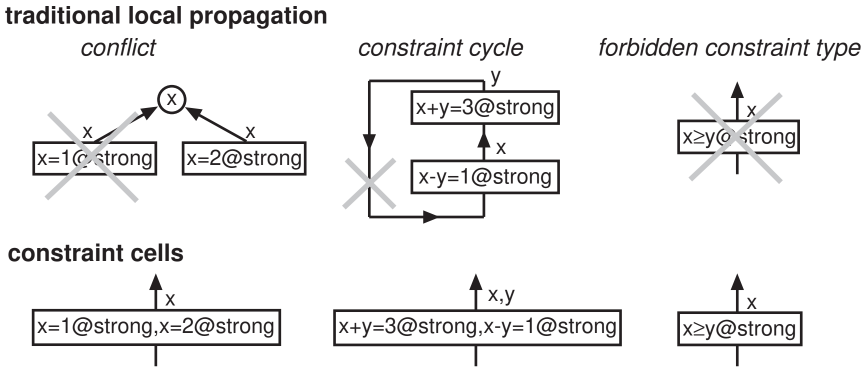

Figure 1 (traditional local propagation vs. constraint cells)

The role of a constraint cell in a constraint hierarchy network, that we formally introduce in the following section, depends on the constraints in the cell and on the distribution of variables into input and output sets of variables. To grasp formally the differences among constraint cells we classify the cells in the following manner.

DEFINITION 2: (classification of constraint cells)

We classify constraint cells into the following groups:

- *free variable cell* ({},{},{v})
- *functional cell* ({c@l},In,Out)[3] such that Out≠∅ and for arbitrary evaluation θ of variables from In there exists a unique valuation σ of variable(s) from Out such that cθσ holds
- *generative cell* (C,In,Out) such that C≠∅ and Out≠∅ and (C,In,Out) is not a functional cell
- *test cell* (C,In,∅)
- *potentially unsatisfied cell* is a generative cell or a test cell

The traditional local propagation enables only free variable cells and functional cells[4]. The planning phase of the traditional local propagation determines which constraints are satisfied, they are in functional cells and thus they are used for (unique) value propagation, and which ones are disabled. The disabled constraints are assumed unsatisfied there and they are (temporarily) disconnected from the network.

In our generalized approach we use a different mechanism for marking disabled, i.e., potentially unsatisfied constraints, namely test cells. Test cell consist of constraint(s) with all variables determined by other constraints (cells). We assume optimistically that these constraints can still be satisfied (see Figure 2) even if there is no degree of freedom to bind output variables in the test cell (as there are no output variables). The satisfaction of constraints in test cells can be assured via influence of the input variables if these variables are not determined uniquely (see Figure 2). We also introduce a generative cell as a generalization of a functional cell. Opposite to a functional cell, the generative cell can consist of more than one constraint and the values are not required to be propagated uniquely through the generative cell. As with the test cell, the constraints in the generative cell can influence values of input variables to ensure satisfaction but, again, it is not

---

[3] c@l denotes constraint c labeled with the preference/strength l.

[4] In fact, instead of the free variable *x*, the traditional local propagation usually uses a special type of constraint in the form `x=user_input` that uniquely determines the value of the variable *x*.

possible to guarantee that all constraints in the generative cell are satisfied. Therefore we call both types, i.e., test and generative cells potentially unsatisfied cells. Note that the constraints from the functional cells can always be satisfied by binding output variable(s) (see Definition 2).

The Figure 2 shows differences among above mentioned types of constraint cells. It also demonstrates the value propagation through the cell.

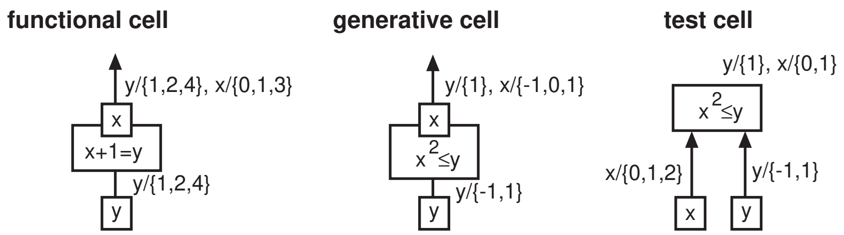

Figure 2 (differences among various constraint cells)

Remind again, that the functional cell propagates values uniquely and the constraint in the functional cell are always satisfied independently of values of input variables while the constraints in the potentially unsatisfied cells, i.e., the generative and test cells, are not guaranteed to be satisfied, the value propagation is not unique there and the potentially unsatisfied cells can even influence the values of input variables (see Figure 2). The impact of potentially unsatisfied cells on input variables develops only when the values of input variables are not unique.

In the Figure 2 we omit the strengths of constraints for clarity purpose. Because all constraints in the constraint cell have the same strength (see Definition 1) we can bind this strength with the constraint cell. This so called internal strength of the constraint cell together with the cell type will be the key for partial ordering of cells that is discussed in the following section.

DEFINITION 3: (internal strength)

The *internal strength* of the constraint cell (C,In,Out) is the strength of any constraint in C. The *internal strength* of the constraint cell ({},{},{v}) is *free* which is the strength that is weaker than any other strength of constraints.

We denote the internal strength of Cell as i_strength(Cell) and we say that Cell is stronger than Cell' if and only if i_strength(Cell)<i_strength(Cell').

2.2 Hierarchy Networks

In this section we describe reasonable decompositions of constraint hierarchies into constraint cells. We also define a partial ordering of cells represented by the hierarchy network here. This partial ordering of cells guides the sound propagation of values through the constraint cells.

Before formal definition of a hierarchy decomposition we can identify some decompositions in current approaches for solving hierarchies. While the refining method decomposes a given hierarchy into levels of equally preferred constraints, the local propagation atomizes the hierarchy into individual constraints. Both these decompositions and many others are covered by our approach.

DEFINITION 4: (conflict-free hierarchy decomposition)

Let H be a constraint hierarchy, i.e., a finite set of labeled constraints, and V be a set of all variables in constraints from H. We call the finite set CC of constraint cells a *conflict-free decomposition* of the hierarchy H if and only if the following conditions hold:

1) the constraint cells from CC consist only of constraints from H, i.e., $\forall Cell \in CC\ Cell=(C,In,Out)\ \&\ C \subseteq H\ \&\ In \subseteq V\ \&\ Out \subseteq V$

2) every constraint from H is located in just one constraint cell, i.e., $\forall c \in H\ \exists!\ Cell \in CC\ Cell=(C,In,Out)\ \&\ c \in C$

3) every variable from V is determined by just one constraint cell, i.e., $\forall v \in V\ \exists!\ Cell \in CC\ Cell=(C,In,Out)\ \&\ v \in Out$.

The satisfaction of the conditions 1) and 2) of the Definition 4 guarantees that the set CC of cells is really a decomposition of the given hierarchy H. In particular, the satisfaction of the conditions $In \subseteq V$ and $Out \subseteq V$ implies that in the decomposition there do not appear new variables, i.e., variables outside the hierarchy. The satisfaction of the condition 3) of the Definition 4 ensures that each variable is determined by any cell and there is no conflict among cells in the decomposition, i.e., there is no variable that is determined by more than one cell.

In the following paragraphs we concentrate on conflict-free decompositions which are practical for efficient solving of constraint hierarchies via value propagation through constraint cells. We also give conditions which must be satisfied by the partial ordering of cells to be feasible for sound value propagation. The feasible decomposition as well as the partial ordering is encapsulated in the notion of constraint hierarchy network (we also use abbreviated term h*ierarchy network* in the same sense). Note that some conflict-free decompositions cannot be partially ordered according to our requirements, however for every constraint hierarchy there exists at least one conflict-free decomposition that is possible to be partially ordered according to the following definition.

In what follows, we identify the partial ordering "<" of constraint cells with the directed acyclic graph that we call a hierarchy network. Thus, we write Cell<Cell' if and only if there exist a directed path in the hierarchy network from Cell to Cell'. Also, we use the denotation G/T for potentially unsatisfied cells.

DEFINITION 5: (constraint hierarchy network)

Let CC be a conflict-free decomposition of the hierarchy H. We call the directed acyclic graph (CC,E) with nodes CC and edges E a *constraint hierarchy network* of hierarchy H if and only if the following conditions hold:

1) for every constraint cell Cell there exist edges in E directed from constraint cells determining the input variables of Cell, i.e.,

$$\forall Cell,Cell' \in CC$$
$$Cell=(C,In,Out)\ \&\ Cell'=(C',In',Out')\ \&\ In \cap Out' \neq \emptyset$$
$$\Rightarrow \quad (Cell',Cell) \in E$$

2) for every potentially unsatisfied cell there does not exist an upstream constraint cell which has the same or weaker internal strength, i.e.,

$$\forall Cell,Cell' \in CC$$
$$(Cell \text{ is G/T}\ \&\ Cell'<Cell) \Rightarrow i\_strength(Cell')<i\_strength(Cell)$$

3) there does not exist "downstream forking" in a potentially unsatisfied cell directed to other potentially unsatisfied cells, i.e.,

$$\forall Cell,Cell',Cell'' \in CC$$
$$Cell,Cell',Cell'' \text{ are G/T}\ \&\ Cell<Cell'\ \&\ Cell<Cell''$$
$$\Rightarrow \quad (Cell'=Cell'' \vee Cell'<Cell'' \vee Cell'>Cell'')$$

Definition 5 describes a constraint hierarchy network that expresses naturally the partial ordering of constraint cells. This ordering can then be used for value propagation during the executing phase that computes exact values of variables in the constraint hierarchy. The meaning of the condition 1) of Definition 5 is straightforward as one requires the input variables of the cell to be computed before the value propagation can go through the cell and compute values of output variables. The conditions 2) and 3) of the Definition 5 ensures the correctness of the propagation as one requires the satisfaction of the stronger constraints to be preferred to the weaker constraints. Because there are no stronger or equally preferred constraint cells in hierarchy network upstream any potentially unsatisfied cell, the satisfaction of constraints in potentially unsatisfied cells is not influenced by weaker constraints. These conditions are discussed in more detail in [4] where also a theoretical justification is given.

Our approach to constraint hierarchy networks allows various hierarchy networks to be constructed for a given constraint hierarchy, from a basic one, that decomposes the hierarchy into levels, to a more structured network, that keeps the cells as small as possible. It depends on the type of used constraints and comparators and on the quality of the planning and executing algorithm which hierarchy network is chosen. The Figure 3 shows two different hierarchy networks corresponding to one constraint hierarchy.

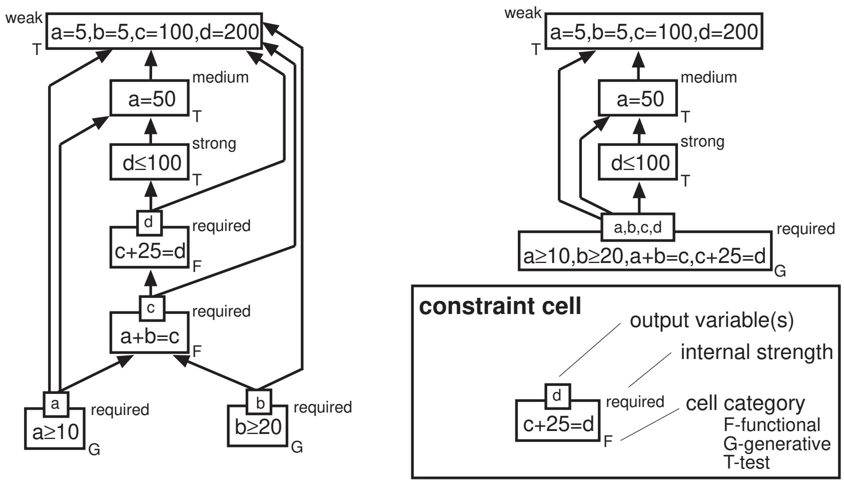

Figure 3 (constraint hierarchy networks)

3 Construction of Hierarchy Networks

When the constraint hierarchy network is defined, one may ask whether it is possible to represent arbitrary constraint hierarchy as a corresponding hierarchy network. The answer is yes which is not really surprising as we present our approach as a generalization.

Definition 5 provides formal description of hierarchy networks but it does instruct how to construct the hierarchy network, i.e., how to distribute constraints into cells and which edges should be added. Therefore, it is not surprising that there can exist more sound algorithms for construction of different hierarchy networks. In this section we present three of them which differ in the size of constraint cells and the number of edges used. The two Straton algorithms[5] builds constraint cells corresponding to preference levels in the hierarchy. They differ in the policy of adding edges. The BlueOne algorithm[6] maintains smaller constraint cells in case of one-output functional constraints are used.

3.1 Straton Algorithm - pure stratification

In [4] we presented an algorithm for construction of hierarchy networks that is based on stratification of the hierarchy. This algorithm keeps all equally preferred constraints in one constraint cell and maintains necessary edges so the conditions of Definition 5 hold (see the right network in Figure 3). In fact, this algorithm connects cells which share common

---

5 The name of Straton algorithm is derived from stratification that is performed by the algorithm.

6 The name of BlueOne algorithm reflects the "Blue" family of algorithms from University of Washington. In fact, BlueOne is a generalization of DeltaBlue algorithm.

variables, we call such edges green, and maintains edges, called red, between cells representing successive levels of the hierarchy. The presence of green edges ensures satisfaction of condition 1 of Definition 5 while the red edges[7] guarantee satisfaction of conditions 2 and 3 of Definition 5. As there is a mistake in [4] (some necessary edges are removed accidentally) we present now a corrected version under the name Straton algorithm.

Straton algorithm is a transcription of traditional refining method in terms of constraint cells and hierarchy networks. The algorithm adds labeled constraints incrementally in two steps:

1) If there exists a cell with the internal strength equal to the strength of the added constraint, then the algorithm adds the constraint into this cell. Otherwise, it creates a new cell containing this constraint.
2) In the second phase the algorithm decides which variables of the added constraint are input and output respectively (depending on variables used in other cells).

When a new cell is added in step 1, the algorithm removes redundant red edge (if any) and add new red edges between new cell and neighboring (in terms of internal strength) cells. During the step 2, the algorithm adds and removes necessary and redundant green edges respectively. Note, that Straton algorithm does not use the free variable cells. In fact, almost all constraint cells are generative and test, only few of them can be functional.

```
straton_add_constraint(c@lab,(CC,E))

   if exists Cell=(Cs,In,Out) in CC s.t. i_strength(Cell)=lab then
      Cs<- Cs ∪ {c@lab}              % add constraint to existing cell
   else
      Cell<- NewCell({c@l},{},{})    % create new cell (Cs,In,Out)
      CC<- CC ∪ {Cell}
      E<- E ∪ {(Cell,CA)} s.t. CA<-ClosestWeaker(Cell)
      E<- E ∪ {(CB,Cell)} s.t. CB<-ClosestStronger(Cell)
      if InputVars(CA) ∩ OutputVars(CB)=∅ then
         E<- E - {(CB,CA)}           % remove redundant red edge
      end if
   end if

   for each v in vars(c) do
      if any Cell'=(Cs',In',Out') determines v then
         case of
           :(i_strength(Cell')<lab)   % v is determined by stronger cell
              In<- In ∪ {v}           % add v among input variables
              E<- E ∪ {(Cell',Cell)} % add corresponding green edge
           :(i_strength(Cell')>lab)   % v is determined by weaker cell
              % move v from output to input variables in Cell'
              Out'<- Out' - {v}
              In'<- In' ∪ {v}
              % remove green edges invalidated by moving variable v
              E<- E - {(Cell',C2) | (Cell',C2)∈E & C2≠ClosestWeaker(Cell')
                                    & Out'∩InputVars(C2)=∅}
              Out<- Out ∪ {v}         % add v among output variables
              % add new green edges
              E<- E ∪ {(Cell,C2) | C2∈CC & v∈InputVars(C2)}
         end case
      else                   % variable v is not present in the network
         Out<- Out ∪ {v}  % add v among output variables
      end if
   end for

end straton_add_constraint
```

Figure 4 (Straton algorithm - constraint adding)

While Straton algorithm is very simple, it requires a subsequent execution phase to mimic the refining method. However, such algorithm of the execution phase is not very effective as all equally preferred constraints are processed together (depending on comparator used).

7 The terms red and green edges are used for explanation purpose only, they are not differentiated in the hierarchy network.

3.2 Straton 2 Algorithm

The pure Straton algorithm does not exploit advantages of constraint hierarchies because the constraint cells are as large as possible (but smaller cells are better for satisfaction) and it keeps (red) edges between levels which does not necessary reflect the dependencies among constraints. To remove the second of the above disadvantages, i.e., spare edges, we introduce Straton 2 algorithm.

Straton 2 algorithm keeps all equally preferred constraints in one constraint cell, as Straton algorithm does, but it maintains only edges necessary to satisfy the conditions of Definition 5. To simplify maintenance of edges we break up Straton 2 algorithm into incremental and non-incremental part. The incremental part allows adding and removing constraints incrementally as it maintains only the green edges in the above used sense, i.e., edges connecting cells with common variables. Thus, only the satisfaction of condition 1 of Definition 5 is kept in the incremental part of Straton 2 algorithm. Because the edges direct from weaker to stronger cells, the condition 2 of the same definition is also satisfied. The condition 3 of Definition 5 is necessary only for sound value propagation through cells, so we introduce non-incremental part of Straton 2 algorithm that is executed right before the value propagation and that adds remaining edges, we call them orange, to satisfy respective condition. The following Figure 5 shows the differences between constraint networks constructed by Straton and Straton 2 algorithms. Note, that both algorithms build same constraint cells corresponding to levels in constraint hierarchy.

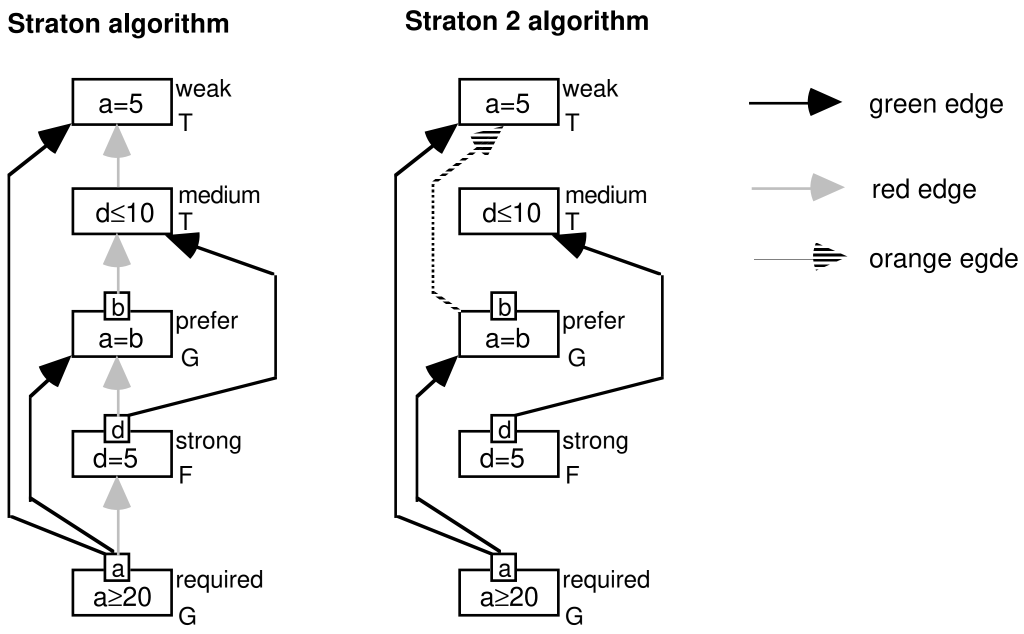

Figure 5 (Straton family of algorithms - differences)

The incremental part of Straton 2 algorithm consists of two procedures for adding and removing constraints. While constraint adding is almost the same as in the original Straton algorithm (with the exception of maintaining red edges), constraint retraction is new in Straton 2 solver. When a constraint is being removed, the system finds the cell containing given constraint (there is only one cell with the internal strength equal to the preference of the constraint), removes constraint from the cell, changes sets of input and output variables accordingly and, finally, re-connects green edges in such a way that the condition 1 of Definition 5 still holds. If the cell consists of only the removed constraint then the cell is removed from the network as well. The algorithm for constraint retraction considers the constraint to be in any cell of the network.

```
straton2_add_constraint(c@lab,(CC,E))

   if exists Cell=(Cs,In,Out) in CC s.t. i_strength(Cell)=lab then
```

```
      Cs<- Cs ∪ {c@lab}              % add constraint to existing cell
   else
      Cell<- NewCell({c@l},{},{})    % create new cell (Cs,In,Out)
      CC<- CC ∪ {Cell}
   end if

   for each v in vars(c) do
      if any Cell'=(Cs',In',Out') determines v then
         case of
           :(i_strength(Cell')<lab)   % v is determined by stronger cell
              In<- In ∪ {v}           % add v among input variables
              E<- E ∪ {(Cell',Cell)} % add corresponding green edge
           :(i_strength(Cell')>lab)   % v is determined by weaker cell
              % move v from output to input variables in Cell'
              Out'<- Out' - {v}
              In'<- In' ∪ {v}
              % remove green edges invalidated by moving variable v
              E<- E - {(Cell',C2) | (Cell',C2)∈E & Out'∩InputVars(C2)=∅}
              Out<- Out ∪ {v}         % add v among output variables
              % add new green edges
              E<- E ∪ {(Cell,C2) | C2∈CC & v∈InputVars(C2)}
         end case
      else                  % variable v is not present in the network
         Out<- Out ∪ {v}  % add v among output variables
      end if
   end for

end straton2_add_constraint
```

Figure 6 (Straton 2 algorithm - constraint adding)

```
straton2_remove_constraint(c@lab,(CC,E))

   Cell=(Cs,In,Out) in CC s.t. i_strength(Cell)=lab
   Cs<- Cs − {c@lab}              % remove constraint from existing cell

   % disconnect edges to Cell invalidated by removing input variables
   In<- In ∩ vars(Cs)
   E<- E - {(C,Cell) | (C,Cell)∈E & In∩OutputVars(C)=∅}

   % disconnect edges from Cell invalidated by removing output variables
   RemovedOut<- Out − vars(Cs)
   Out<- Out - RemovedOut
   E<- E - {(Cell,C) | (Cell,C)∈E & Out∩InputVars(C)=∅}

   % find new cells determing removed variables and reconnect edges
   for each v in RemovedOut do
      CellsUsingV<- {C | C∈CC & v∈InputVars(C)}
      if CellsUsingV non-empty then
         NewDeterminer<- StrongestCell(CellsUsingV)
         CellsUsingV<- CellsUsingV - {NewDeterminer}
         NewDeterminer=(Cs',In',Out')
         In'<- In' - {v}
         Out'<- Out' ∪ {v}
         E<- E ∪ {(NewDeterminer,C) | C∈CellsUsingV}
      end if
   end for

   if Cs is empty then        % remove empty cell from the network
      CC<- CC - {Cell}
   end if

end straton2_remove_constraint
```

Figure 7 (Straton 2 algorithm - constraint retraction)

Both constraint adding and retraction maintains partial hierarchy networks[8] satisfying the conditions 1 and 2 of Definition 5 of hierarchy network. To construct full hierarchy network one needs to add some edges so the condition 3 of the same definition also holds. This is accomplished by non-incremental completion part of Straton 2 algorithm that adds some edges, we call them orange, to remove downstream forking in potentially unsatisfied cells.

[8] We speak about partial (constraint) hierarchy networks because the condition 3 of Definition 5 of hierarchy network is not necessary satisfied.

Completion algorithm traverses the partial hierarchy network from weaker to stronger constraint cells and computes both direct and indirect successors of each cell. If the algorithm finds any generative cell[9] then it go through its successors in weak to strong manner[10] and adds necessary orange edges to remove downstream forking as specified by the condition 3 of Definition 5.

```
straton2_completion((CC,E))

   Queue<- SortCellsW2S(CC)  % sort cells in weak to strong manner
   CurrentCell<- RemoveHead(Queue)
   Succ(CurrentCell)<- EmptyQueue

   while Queue not empty do
      CurrentCell<- RemoveHead(Queue)
      DirSucc<- {C | (CurrentCell,C)∈E}
      Succ(CurrentCell)<- SortCellsW2S(DirSucc ∪ {∪Succ(C) | C∈DirSucc})

      if CurrentCell is generative then
         TestQueue<- RemoveFunctional(Succ(CurrentCell))

         while TestQueue non empty do
            TestCell<- RemoveHead(TestQueue)
            RestQueue<- TestQueue

            while RestQueue not empty do
               TestCell2<-RemoveHead(RestQueue)
               if TestCell∉Succ(TestCell2) then
                  insert TestCell to Succ(TestCell2)
                  E<- E ∪ {(TestCell2,TestCell)    % add orange edge
               end if
            end while
         end while
      end if
   end while

end straton2_completion
```

Figure 8 (Straton 2 algorithm - completion)

For simplicity reasons we use queue data structure to collect lists of cells in the above description of the completion algorithm. It is also possible to use bitmaps because the number of cells is limited by the number of hierarchy levels that is constant. Then, it is easier and even more efficient to implement operations like membership, union or difference, and there is no need to sort cells. Altogether, the worst complexity of the completion algorithm is $O(n^3)$, where n is a number of potentially unsatisfied cells. Because n is less or equal to the number of hierarchy levels, the worst complexity of the completion algorithm is a function depending on the number of levels, that is a constant.

We believe that it is possible to construct fully incremental version of Straton 2 algorithm that maintains orange edges incrementally. The question is whether it worthes the whistle.

A smaller number of edges in constraint hierarchy network is the main advantage of Straton 2 algorithm on Straton algorithm. Also, the structure of hierarchy networks constructed by Straton 2 reflects better the dependencies among constraints (see Figure 5). Because the constraint cells correspond to hierarchy levels, it is still necessary to use some form of refining algorithm for value propagation. However, a better structure of hierarchy network enables the value propagation algorithm to be more efficient.

3.3 BlueOne Algorithm

The Straton family of algorithms mimics the behavior of refining method that is not very effective because all equally preferred constraints have to be solved together (depending on comparator chosen). Therefore, in [2] we proposed a sophisticated algorithm, called

[9] Test cells have not successors in partial hierarchy network because they have not output variables.

[10] If weak to strong order is used then the number of added orange edges is minimal.

now BlueOne, that constructs more structured constraint hierarchy networks by keeping constraint cells as small as possible.

BlueOne algorithm is based on ideas of DeltaBlue algorithm [14], it processes one-output functional constraints, e.g., linear equalities, and it considers all other constraints, e.g., inequalities, to be generative. BlueOne algorithm adds labeled constraints incrementally to the hierarchy network in three steps:

1) Try to add the constraint as a new function cell.
2) If step 1 fails then add constraint as a new generative or test cell.
3) Add edges or join some cells to remove downstream forking and, thus, to satisfy the condition 3 of Definition 5. The Figure 9 sketches the idea of removing downstream forking.

During the process of adding a constraint we use similar techniques as DeltaBlue algorithm, namely walkabout strength. Also, some constraints, i.e., functional cells can be removed during the process of adding a constraint. These temporarily retracted constraints are added to the network using the same algorithm until all constraints are in the hierarchy network.

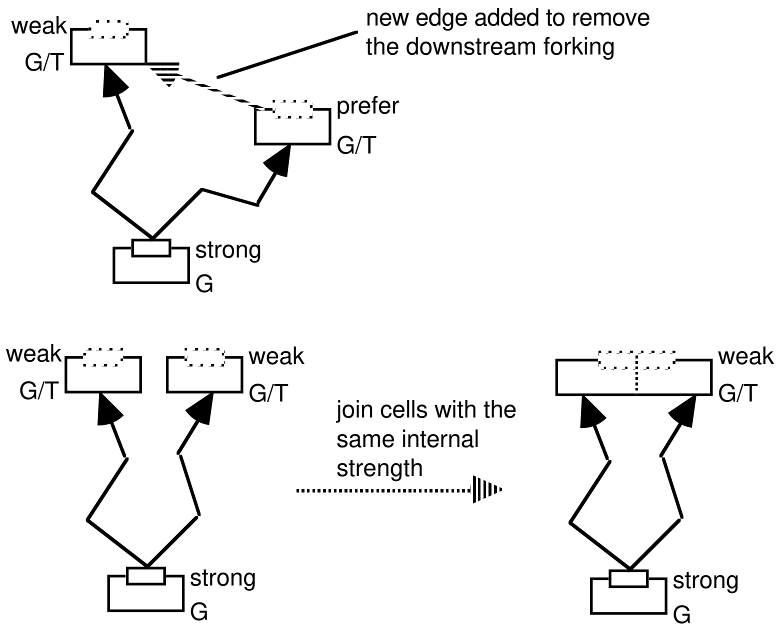

Figure 9 (removing downstream forking)

This is only a brief non-formal description of BlueOne algorithm. We have implemented an experimental prototype of the algorithm that is available on-line at URL: `http://kti.mff.cuni.cz/~bartak/html/constraints.html`. This implementation uses some ad-hoc techniques so we are now working on formal description of DeltaBlue algorithm.

4 Value Propagation

In the above sections, we presented a general framework for solving constraint hierarchies with particular emphasis on construction of hierarchy networks, i.e., constraint planning. When a hierarchy network is constructed the propagation phase of constraint solving starts. During this phase, sets of values are propagated through constraint cells in the order defined by the hierarchy networks. The value propagation starts with a complete domain for each variable and it reduces the domains successively as the constraints are being satisfied. Because potentially unsatisfied cells can even influence values of input variables, this value propagation process is not linear and sometimes it has to resatisfy constraints in already visited cells. Because the zig-zag cycling decreases efficiency of the value propagation, the sophisticated planning algorithms tries to keep a many constraints as possible in the unctional cells.

In our experimental implementation, we utilize the Indigo algorithm [5] as a kernel of value propagation algorithm. Indigo algorithm was designed to propagate intervals through equality and inequality constraints in the decreasing order of constraint strength. In our approach we use the order of constraints defined by the constraint hierarchy network. Note, that this order is not necessarily decreasing in the constraints' strength. A detail description of using Indigo for value propagation in constraint hierarchy networks can be found in [2,3].

5 Conclusions

In this paper we present a generalized framework for hierarchical constraint solving, in particular for constraint planning. We define constraint cells and constraint hierarchy networks and we put special emphasis on the construction of hierarchy networks. We formally describe two planning algorithms of Straton family and we sketch ideas of the sophisticated planning algorithm BlueOne. Finally, we briefly describe the process of value propagation through the hierarchy network.

The closest work related to our approach is probably [10,11] that describes the DETAIL algorithm. However, there are some significant differences between DETAIL and our framework. While the DETAIL works with equality (functional) constraints and concentrates especially on removing cycles and conflicts from constraint graphs, we mainly focus on support of all types of constraints and comparators. Also, the notion of a constraint cell is a bit different in DETAIL where it serves more as a meta-notion denoting a closed part of the constraint graph. DETAIL allows constraints with different strengths to be in one constraint cell, whereas our approach gathers only equally preferred constraints in the constraint cell.

The aim of the future research is to show practical applicability of the proposed framework in real-life applications. We put special emphasis on formal description of the sophisticated planning algorithm BlueOne, whose experimental implementation promises good efficiency results. Further, we expect to further generalize the BlueOne algorithm to enable working with multi-variable output functional cells.

Acknowledgments

I would like to thank professor Petr Štěpánek for his continuous support, useful discussions and comments. This research was supported in part by the Grant Agency of Czech Republic under Grant No. 201/96/0197.

References

[1] Barták, R., A Plug-In Architecture of Constraint Hierarchy Solvers, in Proceedings of PACT'97, London, April 1997, also available as Tech. Report No 96/8, Department of Computer Science, Charles University, December 1996

[2] Barták, R., A Generalized Algorithm for Solving Constraint Hierarchies, Tech. Report No 97/1, Department of Theoretical Computer Science, Charles University, January 1997

[3] Barták, R., Expert Systems Based on Constraints, Doctoral Dissertation (in Czech), Charles University, Prague, April 1997

[4] Barták, R., A Generalized Framework for Constraint Planning, Tech. Report No 97/9, Department of Theoretical Computer Science, Charles University, June 1997

[5] Borning, A., Anderson, R., Freeman-Benson, B., The Indigo Algorithm, Tech. Report 96-05-01, Department of Computer Science and Engineering, University of Washington, July 1996

[6] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., Woolf, M., Constraint Hierarchies, in: *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications*, pp.48-60, ACM, October 1987

[7] Borning, A., Freeman-Benson, B., The OTI Constraint Solver: A Constraint Library for Constructing Interactive Graphical User Interfaces, in: *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pp. 624-628, Cassis, France, September 1995

[8] Borning, A., Maher, M., Martindale, A., Wilson, M., Constraint Hierarchies and Logic Programming, in: *Proceedings of the Sixth International Conference on Logic Programming*, pp. 149-164, Lisbon, June, 1989

[9] Bouzoubaa, M., Neveu, B., Hasle, G., Houria III: Solver for Hierarchical System, Planning of Lexicographic Weight Sum Better Graph For Functional Constraints, in: *the Fifth INFORMS Computer Science Technical Section Conference on Computer Science and Operations Research*, Dallas, Texas, Jan. 8-10, 1996

[10] Hosobe, H., Miyashita, K., Takahashi, S., Matsuoka, S., Yonezawa, A., Locally Simultaneous Constraint Satisfaction, in: *Principles and Practice of Constraint Programming---PPCP'94 (A. Borning ed.), no. 874 in Lecture Notes in Computer Science*, pp. 51-62, Springer-Verlag, October 1994

[11] Hosobe, H., Matsuoka, S., Yonezawa, A., Generalized Local Propagation: A Framework for Solving Constraint Hierarchies, in: *Principles and Practice of Constraint Programming---CP'96 (E. Freuder ed.), Lecture Notes in Computer Science*, Springer-Verlag, August 1996

[12] Menezes, F., Barahona, P., An Incremental Hierarchical Constraint Solver, in: [15]

[13] Sannella, M., The SkyBlue Constraint Solver, Tech. Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993

[14] Sannella, M., Freeman-Benson, B., Maloney, J., Borning,A., Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm, Tech. Report 92-07-05, Department of Computer Science and Engineering, University of Washington, July 1992

[15] Saraswat, V. and Van Hentenryck, P. (eds.), *Principles and Practice of Constraint Programming*, The MIT Press, Cambridge, Massachusetts, 1995

[16] Vander Zanden, Brad, An Incremental Algorithm for Satisfying Hierarchies of Multi-way Dataflow Constraints, Tech. Report, Department of Computer Science, University of Tennessee, March 1995

[17] Wilson, M., Borning, A., Hierarchical Constraint Logic Programming, TR 93-01-02a, Department of Computer Science and Engineering, University of Washington, May 1993