CORNFLOWER LIBRARY

October 2008

User Guide and Reference Manual

Table of Contents

Welcome to Cornflower

PREFACE

Cornflower Project aims to solve the algorithmic problems that arise in the context of optimization driven decision support systems. Our main goal is to develop techniques that allow inexperienced users to exploit optimization power efficiently. We integrate and hybridize ideas developed in different communities in order to provide easily accessible high performance optimization technology. Particularly, we focus on the development of high-level constraints that allow users to model the problems as conjunctions of intuitive substructures and provide hybrid methods for their efficient combination. Moreover, we develop automization techniques for the handling of symmetries that can the cause of severe inefficiencies when handled poorly. Cornflower Library makes our algorithms and methods publicly available. Read more about Cornflower from our webpage.

About this manual

This guide is composed of parts that use an example based learning strategy to illustrate the algorithms implemented in Cornflower Library. Each part is build around a sample problem and a user works on partially completed code examples. Then, you can compile the final code and run and analyze the results. At the end of each part, there is a reference manual for the available methods.

The manual is designed to be used by C++ programmers who may or may not have any knowledge of high-level constraints, hybrid methods and symmetry breaking. The ideal usage context for this manual is to work through the examples step by step, with Cornflower Library downloaded in your computer.

How this manual is organized

This manual is divided into three parts:

- Part I: Context-Free Grammar Constraints
- Part II: Knapsack Constraint
- Part III: Symmetry Breaking

Prerequisites

The Cornflower library requires knowledge of C++ without exposing any syntactic extension. You should have required header files and libraries downloaded before using the manual.

Website

http://cs.brown.edu/research/cornflower

Terms of Use

You may use the available software without limitation for academic purposes. We kindly request you to cite our related publication(s) if you use Cornflower in your research. If you wish to use the software for commercial applications, please obtain the prior permission of Prof. Meinolf Sellmann.

License Information

Contact

You can share your development experience with us through serdark@cs.brown.edu. Please let us know for any suggestions and/or bug reports.

Part I

CONTEXT-FREE GRAMMAR CONSTRAINTS

1. Introduction

A major strength of constraint programming is its ability to provide the user with high-level constraints that capture and exploit problem structure. However, this expressiveness comes at the price that the user must be aware of the constraints that are supported by a solver. One way to overcome this problem is by providing highly expressive global constraints that can be used to model a wide variety of problems and that are associated with efficient filtering algorithms. A promising avenue in this direction is the introduction of constraints that are based on formal languages, which enjoy a wide range of applicability while allowing the user to focus on the desired properties of solutions rather than having to deal for herself with the problem of constraint filtering.

[Sellmann, 2006] devises algorithms which perform filtering for context-free grammar constraints in polynomial time and [Kadioglu and Sellmann, 2008] focuses on an incremental filtering algorithm which combines low memory requirements with very efficient incremental behavior.

2. Context-Free Grammar Constraints (CFGC)

Definition: *For a given grammar G ($\sum$ (alphabet), N (non-terminals), P(productions), S_0(start symbol)) and variables $X_1,\ldots,X_n$ with domains $D_1 := D(X_1),\ldots, D_n := D(X_n) \subseteq \sum$, we say that $Grammar_G(X_1 \ldots X_n)$ is true for an instantiation $X_1 \leftarrow w_1,\ldots, X_n \leftarrow w_n$ if and only if it holds that $w = w_1 \ldots w_n \in L_G \cap D_1 \times \ldots \times D_n$. We denote a given constraint $Grammar_G(X_1 \ldots X_n)$ over a context-free grammar G in CNF by $CFGC_G(X_1,\ldots,X_n)$.*

3. CFGC Filtering

[Sellmann, 2006] presents the filtering algorithm for $CFGC_G$ which is based on the Cooke, Younger, Kasami (CYK) parsing algorithm. Given a *word* of size *n*, let w_{ij} denote the sub-sequence of letters starting at position *i* with length *j*. Based on a grammar *G* ($\sum$, *N*, *P*, S_0) in Chomsky Normal Form, CYK determines iteratively the set of all non-terminals from where we can derive w_{ij} for all $1 \leq i \leq n$ and $1 \leq j \leq n-i$. It is easy to initialize the sets S_{i1} just based on w_i and all productions $(A \rightarrow w_i) \in P$. Then for *j* from 2 to *n* and *I* from 1 to $n - j + 1$, we have that $S_{ij} = \bigcup_{k=1}^{j-1} \{ A \mid (A \rightarrow BC) \in P \text{ with } B \in Sik \text{ and } C \in Si = k, j-k \}$. Then a word is in a language if and only if $S_0 \in S1n$. The algorithm works bottom-up by computing sets S_{ij} for increasing *j* after initializing S_{i1} with all non-terminals that can produce in one step a terminal in the domains of X_i. Then, the algorithm works top-down by removing all non-terminals from each set S_{ij} which cannot be reached from $S_0 \in S_{1n}$. We refer the interested reader to [1] and [2] for many details that are omitted here.

4. Example: Language of all correctly bracketed expressions

Assume that we are given a context-free, normal form grammar G and decision variables $X_{i:1,\ 2,\ 3,\ 4}$ with initial domains D_i as shown below. In Figure 1, we illustrate how our filtering algorithm works. First, we work bottom-up, adding non-terminals to the sets S_{ij} if they allow generating a word in $D_i \times \ldots \times D_{i+j-1}$. Then, in the second step, we work top-down and remove all non-terminals that cannot be reached from $S_o \in S_{1n}$. We see that the constraint filtering algorithm determines the only word "[], []" and the domains of the decision variables are filtered accordingly.

Language of all correctly bracketed expressions.

Context-Free Grammar G

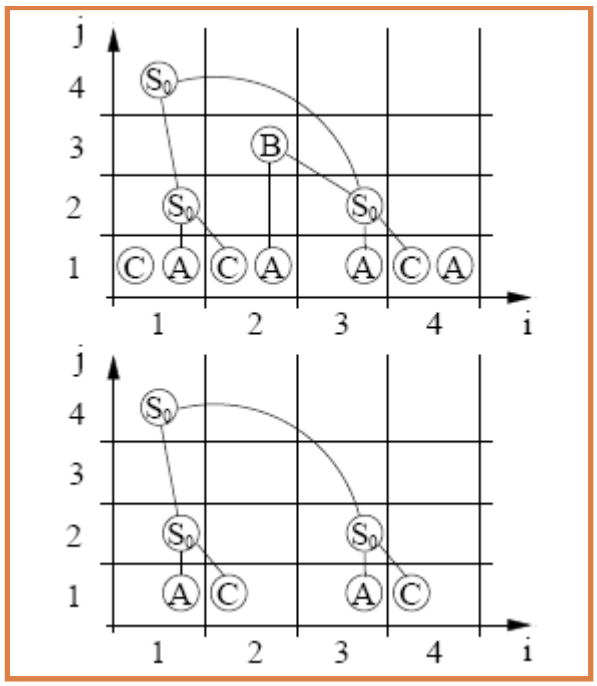
$S_o \rightarrow S_oS_o$ $\quad S_o \rightarrow AC$
$S_o \rightarrow BC$ $\quad B \rightarrow AS_o$
$A \rightarrow [$ $\quad C \rightarrow]$

Decision Variables $X_{1,\,2,\,3,\,4}$

Initial domains

$CFGC_G(X_1, X_2, X_3, X_4)$

1) The filtering algorithm first works bottom-up by computing sets S_{ij} for increasing j after initializing S_{i1} with all non-terminals that can produce in one step a terminal in the domain of X_i.



2) Then the algorithm works top-down removing all non-terminals from each set S_{ij} which cannot be reached from $S_0 \in S_{1n}$.

(Kadioglu, Sellmann AAAI'08)

Figure.1

Updated domains

5. Incremental Computation

One way to speed up filtering algorithms is to incorporate mechanism for incremental update of important data structures. In fact, the filtering algorithm is often applied with slightly changed domain information. Therefore, it is desirable that the data structures built during the first filtering operation are maintained dynamically without building them again for each subsequent run. Specifically, we consider the incremental grammar constraint filtering in this section.

5.1 Efficient CFGC Filtering

In Figure.1 we observe that the algorithm first works bottom-up, determining from which non-terminals of the grammar we derive a legal word. Then it works top-down determining which non-terminal nodes can be used in a derivation that begins with start symbol $S_0 \in S_{1n}$. In order to save both space and time, we will modify these two steps such that every non-terminal in each set Sij will perform just enough work to determine whether its respective node will remain in the shrunken graph. To this end, in the first step that works bottom-up we will need a routine that determines whether there exists a *support from below*: That is, this routine determines whether a node has any *outgoing* arcs. Analogously, in the second step that works top-down we will rely on a procedure that checks whether there exists a *support from above*: Formally, this procedure determines whether a node has any *incoming* arcs.

The challenge is to avoid having to pay with time what we save in space. To this end, we need a methodology which prevents us from searching for supports (from above or below) that have been checked unsuccessfully before. Very much like the well-known arc-consistency algorithm AC-6 for binary constraint problems we achieve this goal by ordering potential supports so that, when a support is lost, the search for a new support can start right after the last support, in the respective ordering. We refer the interested reader to [Sellmann, 2006] and [Kadioglu and Sellmann, 2008] for many details that are omitted here.

5.1.1. Example

Consider our previous example of correctly bracketed expressions with four variables each with a domain ({,}). Below, we show how the filtering algorithms construct and maintain the necessary data structures. The algorithms differ when an update occurs in one of the variable's domain. While non-incremental filtering disregards the previous structure and creates a new structure from scratch, incremental filtering updates only the necessary parts of the initial structure.

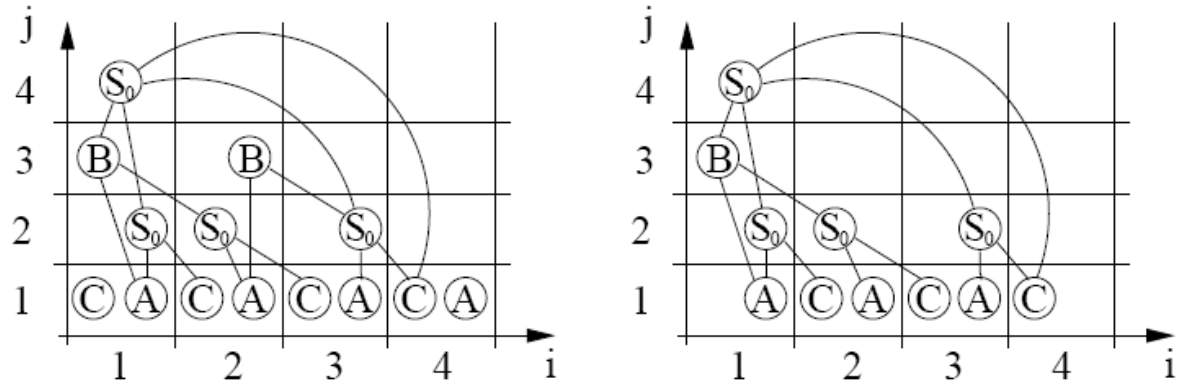
5.1.2. How to backtrack?

As the data structure is created again and again for every decision point in the non-incremental algorithm, we do not need an explicit backtrack operation. However, in the incremental case only a subpart of the original structures is changed, hence we need a way to undo the changes that we have committed and to restore back to a previous state. This is possible by storing extra information whenever a node is removed from any S_{ij} (notice that that domain filtering monotonically decreases the number of nodes in *S*). A node can be created given its position *i, j* and the production rules that were supporting it from above and below along with their splitting indices. Therefore, to save space, we do not store a node as a whole but only the enough information to create it again. This information is stored in a array called *Tracer* (see below). A pointer, called *marker* is *used* to refer the most recent backups in the *Tracer*. Each time a branching decision is made, the marker is increased. The backtrack method operates on the

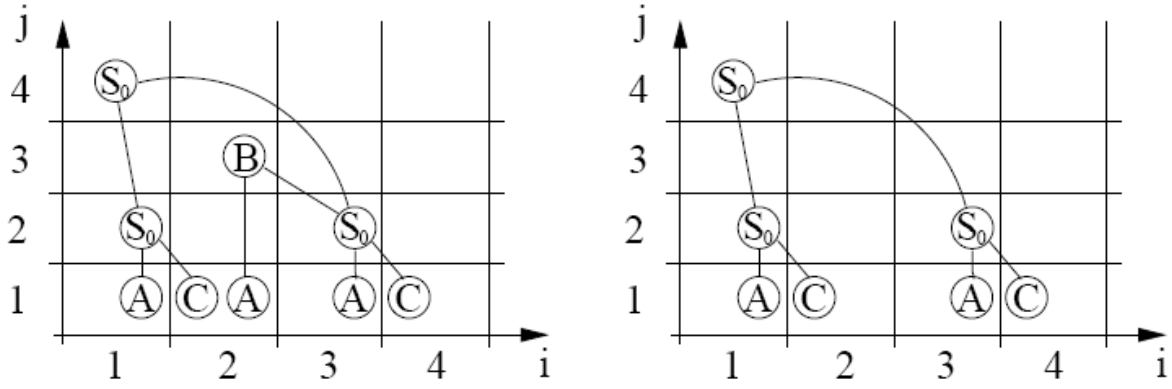
part of the *Tracer* that is pointed by the *marker*, *Tracer[marker]*, considering the changes in reverse order it re-creates the removed nodes using their stored information.

Non-Incremental Filtering

1) Create the initial data structure S_0. The figure on the left works bottom-up and the picture on the right works top-down. Left parenthesis is removed from the domain of first variable and right parenthesis is removed from the domain of last variable.

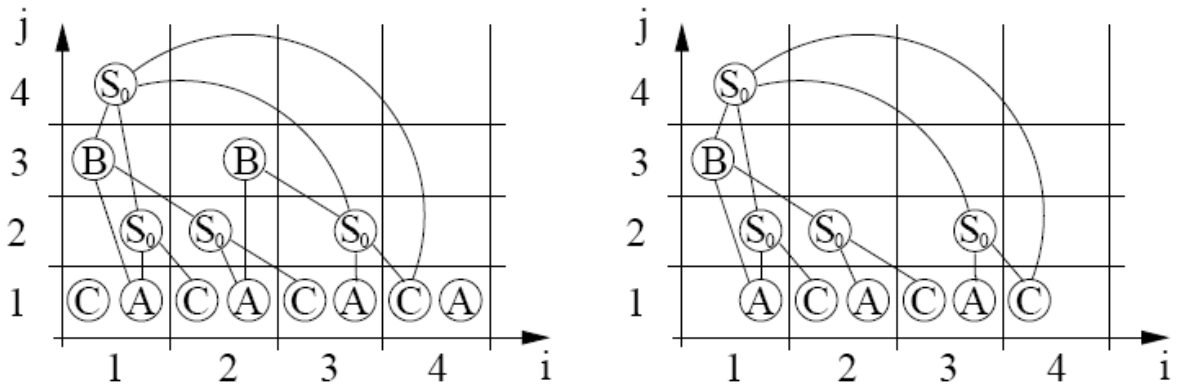


2) Now, consider a change in the domain of third variable. The algorithm clears the old data structure S_0 and creates a new data structure, S_1. Again the left figure works bottom-up and the right figure works top-down. Notice the filtered values from the domains.

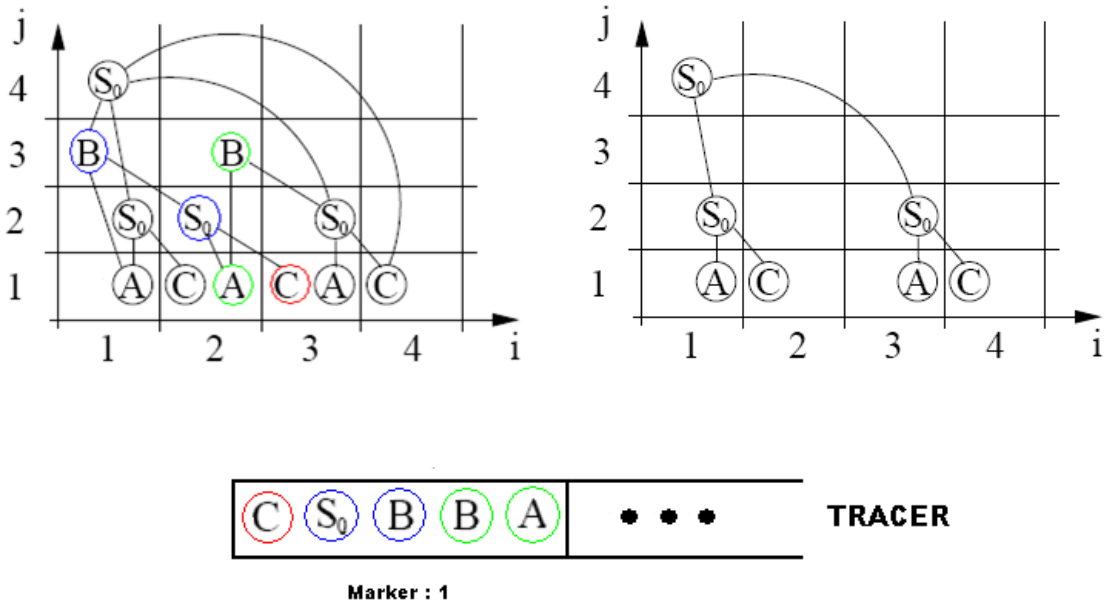


Incremental Filtering

1) The initialization part is identical to the non-incremental case; create the initial data structure S_0. The figure on the left works bottom-up and the picture on the right works top-down. Left parenthesis is removed from the domain of first variable and right parenthesis is removed from the domain of last variable.



2) Now, when the left parenthesis is removed from the domain of third variable the incremental algorithm removes the corresponding node (the node in red circle) from S_0 and propagates the effect of this removal up (the nodes in blue circle) and down (the nodes in green circle). As the nodes are removed' backup information is stored in *Tracer*. At each decision point the *marker* is shifted to right. When backtrack is invoked the nodes in *Tracer[marker]* is re-created and *marker* is shifted to left.



6. Building the example

6.1. Ilog Solver Extension

- Download cfgc.tar.gz file from the webpage and unzip it. The package includes a precompiled library (.a) file, necessary header files (cfgc.h, grammar.h, IlcGCI.h) and a sample main.cpp together with a make file to compile them all. All source codes are compiled using g++.4-1 and Ilog Solver 6.4 on a Linux machine.

- Compilation and Linking

  Inside the unzipped package type "make main" to compile. You have to change $SOLVERDIR parameter in the make file to your own Ilog Solver directory. In order to link your source codes to our library you can use -L option of g++ compiler. If you are using another compiler please refer to its manual.

- Implementation

  - Include the necessary header files.

```
#include "grammar.h"
#include "cfgc.h"
#include "IlcGCI.h"
#include <ilsolver/ilosolverany.h>
```

  - Construct a grammar. The input grammar file must have one production per line and must be in Chomsky Normal Form.

```
grammar g("input.txt");
```

Example input file for the language of all correctly bracketed expressions:

```
S0 S0 S0
S0 A C
S0 B C
B A S0
A [
C ]
```

input.txt

- Initialize domains. The domain size is determined by the number of terminals in the grammar. The *getDomain()* function returns a pointer for each terminal value to be represented in the domain.

```
IloAnyArray domain(env, g.getNumTerminals());

for(i=0; i < g.getNumTerminals(); i++)
{
      domain[i] = (IloAny) g.getDomain(i);
}
```

- Create decision variables. Decision variables are of the type IloAnyVar and they have a set of pointers in their domain to terminal values of the given grammar.

```
IloAnyVarArray variables(env, numVariables);

for(i=0; i < numVariables; i++)
{
       IloAnyVar anyVar(env, domain);
}
```

- Construct CFGC. The parameters are grammar g, instance of an IloSolver, IloAnyVarArray decision variables of size numVariables. A boolean flag is used to select between **incremental** and **non-incremental** propagation. The last parameter is the constraint id.

```
cfgc* cfgcPtr = new cfgc(g, solver, variables,
numVariables, incremental, “correctly bracketed”);
```

- Add CFGC to problem model. We wrap our cfgc instance into an Ilog constraint using IloGC class.

```
model.add( IloGC(env, variables, cfgcPtr, incremental)
```

- You can introduce additional constraints using operators = and ≠.

```
model.add (variables[2] != g.getTerminal("]");
```

- Run solver. IlcGCI class implements a simple branching routine, IloMyBranching, which selects the variable with the minimum domain size and assigns it to the first value in its domain. You can modify IlcGCI file to implement your own branching heuristic.

```
solver.solve(IloMyBranching(env, variables, cfgcPtr,
incremental))
```

- Display solution.

```
for(i=0; i < numVariables; i++)
{
      solver.out ≪
      *((string*)solver.getAnyValue(variable[i]));
}
```

6.2. Standalone Application

❖ Download cfgc.tar.gz file from the webpage and unzip it. The package includes a precompiled library (.a) file, necessary header files (cfgc.h, grammar.h) and a sample main.cpp together with a make file to compile them all. All source codes are compiled using g++.4-1 on a Linux machine.

❖ Compilation and Linking

Inside the unzipped package type "make main" to compile. In order to link your source codes to our library you can use -L option of g++ compiler. If you are using another compiler please refer to its manual.

❖ Implementation

- Include the necessary header files.

```
#include "grammar.h"
#include "cfgc.h"
```

- Construct a grammar. The input grammar file must have one production per line and must be in Chomsky Normal Form.

```
grammar g("input.txt");
```

Example input file for the language of all correctly bracketed expressions:

```
S0 S0 S0
S0 A C
S0 B C
B A S0
A [
C ]
```

input.txt

- Initialize domains. A STL set container is used a domain. The domain size is determined by the number of terminals in the grammar. The *getDomain()* function returns a pointer for each terminal value to be represented in the domain.

```
set<string*> domain;

for(i=0; i < g.getNumTerminals(); i++)
{
	domain.insert = g.getDomain(i);
}
```

- Create decision variables. Decision variables are of type *STL set* which have *string* values in their domains. Precondition: The *string* values in the domains must be terminal values in input the grammar as initialized above.

```
vector<set<string*> > variables(numVariables, domain);
```

- Construct CFGC. The parameters are grammar g, number of variables, a pointer to the variables and an id for the constraint.

```
cfgc c(g, numVariables, &variables, "correctly bracketed");
```

- **Non-incremental filtering.** This method is used to generate the graph in the Figure1. Whenever a variable is modified, you can re-call the method again. Variable array is updated when *filterFromScratch* is called.

```
c.filterFromScratch();
variables[2].erase(g.getTerminal("]"));
c.filterFromScratch();
```

- **Incremental filtering.** You should invoke *filterFromScratch method* as the first filtering in order to generate the initial graph in the Figure1. Then, whenever a variable is updated you can store its domain delta and call *filterFromUpdate* method. Here, we do not generate a graph again; instead we update only the necessary parts of the initial graph. Variable array is updated when *filterFromScratch* is called.

- **Backtrack.** The *filterFromUpdate* method records the necessary information that will allow backtracking to a previous state. A restore point can be created by *setMarker* method before calling *filterFromUpdate* method. Then, you can roll back the changes using *backtrack()* method.

```
c.setMarker();
variables[2].erase(g.getTerminal("]"));
vector< pair<int, string*>> affected;
affected.push_back( pair<int, string*>
                    (2, g.getTerminal("]"));
c.filterFromUpdate(affected);
c.backtrack();
```

- Display solution.

```
for(i=0; i < numVariables; i++)
{
      cout≪"Domain(var: "≪i≪") = ";
      set<string>::const_iterator iter;
      for(iter = variables[i].begin();
          iter != variables[i].end(); ++iter)
      {
            cout.out ≪ *((string*)*iter ≪" ";
      }
}
```

7. Writing a Search Goal

IlcGCI class provides a sample search goal. This goal works as follows; it selects the variable with the minimum domain size and assigns it the first value in its domain. You may want to implement different search goals for your specific problems by modifying *IlcMyBranhing* method of IlcGCI class. When you are using incremental filtering, you should invoke *setMarker* method of cfgc class as you set or remove a value from a variable and you should invoke *backtrack* and *propagate* methods of cfgc class in your *IlcPropate* and *IlcBacktrack* functions. Please refer to Ilog user's manual for details about Ilog search goal.

8. References

[1] Meinolf Sellmann
The Theory of Grammar Constraints
Proceedings of the 12th intern. Conference on the Principles and Practice of Constraint Programming (CP), Springer LNCS 4204, pp. 530-544, 2006.

[2] Serdar Kadioglu and Meinolf Sellmann
Efficient Context-Free Grammar Constraints
Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI), pp. 310-316, 2008.

9. REFERENCE MANUAL

CFGC

Category Class

Inheritance Path

Definition File `cfgc.h`

Include File

```
grammar.h
<ilsolver/ilosolverany.h>
<ilsolver/ilosolverint.h>
```

Constructor Summary	
public	cfgc(grammar g, IloSolver s, IloAnyVarArray V, int n, bool incremental, string id)
public	cfgc(grammar g, string id)

Method Summary	
public bool	propagate()
public void	setMarker()
public void	backtrack()
public void	filterFromScratch()
public void	filterFromUpdate()
public void	filterFromUpdate(vector< pair<int, string*> > affected)

Description The class cfgc is at the core of grammar constraints. It implements both incremental and non-incremental filtering algorithms. An instance from grammar class is used as a private member to enforce the production rules and to prune infeasible values from the domain of IloAnyVar variables.

Constructor

```
cfgc (grammar g, IloSolver s, IloAnyVarArray V, int n,
bool incremental, string id)
```

This constructor creates an instance of grammar constraint to be used as an Ilog extension. Before using it in Ilog, you must wrap it into an Ilog grammar constraint. See also IlcGCI class.

```
cfgc (grammar g, vector< set<sring*>>* variablesPTr,
string id)
```

This constructor creates an instance of grammar constraint to be used for standalone applications.

Methods

```
public bool propagate()
```

This member function is invoked when the incremental algorithm is called. When a domain change happens in any of the variables, propagation is called. You should use that function if you want to modify the Ilog search goal. The method returns true unless the propagation causes an empty domain for some variable.

```
public void setMarker()
```

Whenever a variable is assigned to a value setMarker is used to create a restore point. You should use that function if you want to modify the provided Ilog search goal. The setMarker method indicates the nearest restore point on *Tracer* data structure which is used to store backtrack information.

```
public void backtrack()
```

This member is invoked when propagation returns false. It restores back the previous state before the propagation using a data structure called *Tracer*. Whenever a node in the graph is removed or its supports change, the *Tracer* saves its previous support indices from above and below and the splitting indices to be used later as backtrack information. See also setMarker().

```
public void filterFromScratch()
```

Non-incremental filtering algorithm.

`public void filterFromUpdate()`

`public void filterFromUpdate(`vector< pair<int, string*> > affected`)`

Incremental filtering algorithm. The second function is used in standalone applications. It requires an array of domain delta information which stores a variableIndex and pointer to removed value.

GRAMMAR

Category Class

Inheritance Path

Definition File `grammar.h`

Include File `structs.h`

Constructor Summary	
public	grammar(string inputfile)

Method Summary	
public bool	getStatus()
public int	getNumProductions()
public int	getNumTerminals()
public int	getNumNonTerminals()
public string *	getTerminal(string s)
public string *	getDomain(int i)
public string	getStartSymbol()
public void	setStartSymbol(string s)

Description This class is used to create an instance from user's grammar input. It provides pointers to initialize domains of lloAny variables.

Constructor

```
grammar (string inputfile)
```

This constructor creates an instance of grammar given the input file. The file must contain one production per line. The grammar must be context-free and in Chomsky Normal Form. See sample grammar in the package.

Methods

```
public bool getStatus()
```

Returns true if the grammar object is successfully created.

```
public int getNumProdcutions()
```

Returns the number of productions in the given grammar.

```
public int getNumTerminals()
```

Returns the number of terminals (alphabet, $\sum$) in the input grammar. The initial domain size for IloAny variables is equal to that number.

```
public int getNumNonTerminals()
```

Returns the number of non-terminal in the input grammar.

```
public string* getTerminal(string s)
```

Returns a pointer to the terminal string *s* in the grammar. This member is used when adding extra constraints to problem. See the example in section 4. Precondition: *s* is a terminal in the input grammar.

```
public string* getDomain(int i)
```

Returns a pointer for the i^{th} terminal. This member is used to initialize variable domains. See the example in section 4.

```
public string getStartSymbol()
```

Returns the start symbol S_0 of the grammar. The default start symbol is the left hand side production in first line of the input file.

```
public void setStartSymbol(string s)
```

This method can be used to change start symbol if needed.
Precondition: *s* is a left hand side terminal in the given grammar.

ILCGCI

Category Class

Inheritance Path

Definition File `IlcGCI.h`

Include File
```
<ilsolver/ilosolverint.h>
<ilsolver/ilosolverany.h>
```

Constructor Summary	
`public`	`IloGC(IloSolver s, IloAnyVarArray V, cfgc* c, bool incremental)`

Method Summary	
`public void`	`propagate()`
`ILCGOAL2`	`backtrack()`

Description This class is used to convert a grammar constraint into an Ilog grammar constraint.

Constructor `IloGC (IloSolver s, IloAnyVarArray V, cfgc* c, bool incremental)`

This constructor creates an instance of Ilog grammar constraint given an Ilog solver, decision variables, actual grammar constraint and a flag to filter incremental or not. IloGC is a wrapper used around IlcGCI class.

Methods

`public void propagate()`

This method invokes an Ilog *fail()*, if grammar constraint filtering returns failure.

ILCGOAL2 `backtrack()`

This method invokes an Ilog *fail()*. If you choose to filer incrementally it makes a call to *backtrack*() method of cfgc class.

Part II

KNAPSACK CONSTRAINT

1. Knapsack Constraint (KP)

Definition: *For a given set of items I each with a weight w_i and a profit p_i, find the subgroup $S \subseteq I$ such that the total profit of the included items is greater than some threshold B and the accumulated weight of the included items is less than or equal to another threshold C.* [1]

2. KP Filtering

Given the knapsack problem and the thresholds, the filtering algorithm computes which items must be included or excluded in order to satisfy the given constraints. A simple example of this would be the algorithm finding that an item is too heavy to be included in any solution or that another item must be included because even combined all other items would not exceed the profit bound. Performing this type of filtering can often simplify the problem that needs to be solved by shrinking the number of variables and constraints that need to be considered.

3. Using KP Library

An example program that uses the kpf library is provided in the kp.tar.gz. This program creates a small random knapsack problem and solves it using a standard branch and bound approach. The following section describe how to set up this example, and goes over the main functions, explaining how the kpf library can be used.

3.1. KP Solver

- Download kp.tar.gz file from the webpage and unzip it. The package includes a precompiled library (.a) file, necessary header files (kpf.h, kp.h, general_includes.h) and a sample solver.cpp, solver.h, and main.cpp together with a make file to compile them all. All source codes are compiled using g++.3-3 on a Linux machine.

- Compilation and Linking
  Inside the unzipped package type “make main” to compile. In order to link your source codes to our library you can use -L option of g++ compiler. If you are using another compiler please refer to its manual.

3.2. Key Functions

This section describes the functionality of the methods in the KPF class and their usage.

3.2.1 Initializing KPF

To initialize an instance of the KPF filtering class, the constructor needs to be passed information specifying the knapsack problem being solved. To describe the knapsack problem, the user must specify the number of items, an array of weights and profits of each item, the maximum allowed weight, and the minimum allowed profit.

The example bellow generates a random knapsack problem and then initializes the KPF instance with this information:

```
// set the number of items to be a random value
//    between 20 and 40
int numItems = rand()\%20+20;

// set the maximum weight threshold to be
//    between 100 and 200
int C = rand()\%100+100;

// set the minimum profit threshold
int B = 0;

// allocate memory for the weight and profit
//    arrays (respectively)
long long* w = (long long*)malloc(numItems*sizeof(long long));
long long* p = (long long*)malloc(numItems*sizeof(long long));

// generate items randomly where the weights and profits
//    are between -100 and 100
for( int ii = 0; ii < numItems; ii++ ){
     p[ii] = rand()\%200 - 100;
     w[ii] = rand()\%200 - 100;
}

// initialize KPF
KPF* kp = new KPF(numItems, w, p, C, B);
```

3.2.2 Filtering Items and Updating

Once the KPF instance is initialized, it has all the information about the knapsack problem, and calling the *filter* method will set the passed in arrays to the items that must be either included or excluded (respectively) from the knapsack in order to achieve the specified weight and profit thresholds. *filter* is called with references to the two arrays that will hold the indices of the filtered items, as well as references to the sizes of these arrays. These provided arrays must be allocated to hold twice the number of items in the problem. Upon completion, the method returns the total number of filtered items.

Thus, to find which items can be filtered from the current knapsack problem, the following code can be used:

```
// number of items that need to be included
long long numInclude = 0;

// number of items that can be excluded
long long numExclude = 0;

// total number of items that can be filtered
long long numFilter = 0;

// array to hold the items that need to be included to
maintain feasibility
long long*
   fInclude = (long long*)malloc(2*numItems*sizeof(long long));

// array to hold items that can be excluded since they will
//    not be part of any feasible solution
long long*
   fExclude = (long long*)malloc(2*numItems*sizeof(long long));

// filter all possible itmes
numFilter =
   kp->filter(fInclude, fExclude, numInclude, numExclude);
```

Once *filter* returns the arrays of items that can be filtered, the changes can be applied to the internal representation of the KPF instance through the *update* function. *update* takes in an array of the items to be committed, the size of this array, and the new statuses of the items. The status of the item is set to 1 if the item is being included, 0 if it is being excluded, and -1 if the status of the item is undetermined. Therefore, once we know which items can be filtered by using the above code, we can commit these changes by calling *update* as in the example bellow:

```
// array of items to be filtered
long long*
filter = (long long*)malloc(numFilter*sizeof(long long));

// array of the statuses of the filtered items
long long* status = (char*)malloc(numFilter*sizeof(char));

// record the items that need to be included
for( int ii = 0; ii < numInclude; ii++ ){
     filter[ii] = fInclude[ii];
     status[ii] = 1;
}

// record the items that need to be excluded
for( int jj = 0; jj < numExclude; jj++ ){
     filter[jj+numInclude] = fExclude[jj];
     status[jj+numInclude] = 0;
}

// update the kp solver to register the changes
kp->update(filter, numFilter, status);
```

Once the status of an item is set using the above code, it can be reverted back to its original undefined status by calling *update* again, this time setting the status to -1. It is therefore important for the user to manually keep a history of all the changes if they want to backtrack, as is the case in a branch and bound search.

3.2.3 Bounds and Thresholds

In addition to maintaining an internal representation of the knapsack problem and filtering items, the KPF instance can also be used to inquire about statistics of the current knapsack problem. These statistics include the upper bound on the achievable profit and the lower bound on the accumulated weight. These values can be extracted by calling *getUpperBoundProfit* and *getLowerBoundWeight* respectively, as is demonstrated bellow:

```
// get the upper bound on the achievable profit of the
//    knapsack problem, given the items that have
//    been filtered
double upperBoundProfit = kp->getUpperBoundProfit();

// get the lower bound on the achievable weight of the
//    knapsack problem, given the items that have
//    been filtered
double lowerBoundWeight = kp->getLowerBoundWeight();
```

The user can change the thresholds of the knapsack problem being filtered by KPF using *setNewThreshold* and *setNewCapacity*.

```
// Require that a feasible solution have an accumulated profit
//    greater or equal to the current accumulated profit
kp->setNewThreshold( kp->getTotalProfit() );

// Require that a feasible solution have an accumulated weight
//    less than the current accumulated weight
kp->setNewCapacity( kp->getTotalWeight() );
```

4. References

[1] Yuri Malitsky, Meinolf Sellmann, Willem-Jan van Hoeve
Length-Lex Bounds Consistency for Knapsack Constraint
Proceedings of the 12th intern. Conference on the Principles and Practice of Constraint Programming (CP), Springer LNCS 5202, pp. 266-281, 2008.

5. REFERENCE MANUAL

KPF

Category Class

Inheritance Path

Definition File kpf.h

Include File kp.h

Constructor Summary	
public	KPF(long long n, long long* weight, long long* profit, long long C, long long B)

Method Summary	
public long long	filter(long long* filter1, long long* filter0, long long& filteredAlready1, long long& filteredAlready0, bool noCommit=true)
public bool	update(long long* filter, long long n, char* newStatus, bool dive = false)
public bool	upperBoundProfit(void)
public double	getLowerBoundWeight(void)
public double	getUpperBoundProfit(void)
public void	setNewThreshold(long long t)
public void	setNewCapacity(long long c)
public long long	getTotalWeight(void)
public long long	getTotalProfit(void)

Description

The KPF class provides the interface to the KP class, enforcing all the production, filtering and bounding rules.

Constructor

```
KPF(long long n, long long* weight, long long* profit,
long long C, long long B)
```

This constructor creates an instance of the knapsack problem.

Methods

```
public long long filter(long long* filter1, long long*
filter0, long long& filteredAlready1, long long&
filteredAlready0, bool noCommit=true);
```

The method computes which items must be included into or excluded from the knapsack in order to satisfy the defined thresholds. This function takes in four input parameters all of which are references to variables that are to be set by the function. The first parameter is an allocated array of size 2n which will hold the indices of the items that must be included in the knapsack. The second parameter is another pre-allocated array of size 2n which will hold the indices of the items that cannot be part of a satisfying solution. The third and fourth parameters should be set to 0 and are then set to the number of items in the respective arrays by the function. The fifth, optional, parameter is a Boolean which states whether the found item changes should be automatically committed. By default this is set to false. The function returns the total number of filtered items or -1 if a feasible solution can't be attained.

```
public bool update (long long* filter, long long n,
char* newStatus, bool dive = false)
```

This function changes the status of items of the problem. The first parameter is an array of indices of the items whose status needs to be changed. The second parameter defines the number of items that need to be changed, while the third is a character array that states the new status of the items (1 for inclusion, 0 for exclusion, and -1 for undecided). The fourth is an optional Boolean parameter meant to be a safety feature that prevents changing the status of items that have already been set to be either included or excluded. By default this parameter is set to false. The update functions returns true if all updates were successful and false otherwise. Setting an item to a state it's already in does not change the state of the problem.

```
public bool upperBoundProfit(void)
```

Returns true if the relaxed solution is integer.

```
public double getLowerBoundWeight(void)
```

Returns the lower bound on the weight for the current partial assignment.

```
public double getUpperBoundProfit(void)
```

Returns the upper bound on the profit on the current partial assignment.

```
public void setNewThreshold(long long t)
```

Sets the new threshold on the knapsack such that the gained profit is greater or equal to the passed-in variable.

```
public void setNewCapacity(long long t)
```

Sets the new capacity on the knapsack such that the accumulated weight is smaller or equal to the passed-in variable.

```
public long long getTotalWeight(void)
```

Returns the total accumulated weight of the items included in the current partial assignment.

```
public long long getTotalProfit(void)
```

Returns the sum profit of the included items in the current partial assignment.

Part III

SYMMETRY BREAKING

1. Introduction

Symmetries occur naturally in many problems (i.e two identical flights to operate or rotation and reflection symmetries in magic square problem) and can cause severe difficulties for exact solvers since we might waste time visiting symmetric solutions and the parts of the search space that we visited already. Hence, symmetry breaking is crucial for search space exploration.

We can define two types symmetry. A variable symmetry is a bijection on variables that preserves solutions. For example, consider an all-different constraint on a set of variables. We can always change any two variable. Analogously, value symmetry is a bijection on values that preserves solutions. Again, in our all-different example, we can freely interchange any two values.

2. A Short Overview of Symmetry Breaking

There are various methods to break symmetries. Static symmetry breaking adds static constraints to the problem before the search and dynamic symmetry breaking tries to detect symmetries during search. [Fahle et al, 2001] presents one such dynamic method; symmetry breaking by dominance detection (SBDD) where they filter values based on symmetric dominance analysis when comparing the current node with other previously expanded nodes. [Sellmann and Hentenryck, 2005] devised structural symmetry breaking (SSB) for problems exhibiting both piecewise symmetric values and variables. Static and dynamic SSB was developed in [Flener et al, 2006] and recently, [Heller et al, 2008] compared static SSB and dynamic SSB in practice. It was shown that static SSB works much faster than dynamic counter part. However, one disadvantage of static SSB is reported to be the variance in runtime when static symmetry breaking constraints clashes with dynamic search orderings or when the static search orderings is not performing well for different problem instances. [Heller et al, 2008] combines static SSB with "model restarts" to address this problem. The details and formal definitions which are omitted here can be found in related papers. In Cornflower library we offer a method to apply static SSB given the symmetric partition for values and variables.

3. Signatures

Before we outline static SSB, we need to introduce the definition of signature. Given a partial assignment A to variables, we define a signature for each value as follows:

$$Signature_A(Value\ v) := (\ |Variable\ x \in (x, v) \in A|\ )_{i \leq numberOfVariablePartitions}$$

That is, given an assignment A, the signature of a value v is the tuple that counts, for each variable partition, the number of variables x that are assigned to the value v.

3.1 Example

Consider the following problem with variables $\{X_1 \ldots X_8\}$ over domains $D(X_i) = \{v_1 \ldots v_6\}$. Assume that the first four and the last four variables are symmetric with each other, i.e. $P_1 = \{ X_1 \ldots X_4\}$ and $P_2 = \{X_5 \ldots X_8\}$. Further assume that there are two value partitions $V_1 = \{v_1 \ldots v_3\}$ and $V_2 = \{ v_4 \ldots v_6\}$. Suppose we are given the assignment in Figure 1.

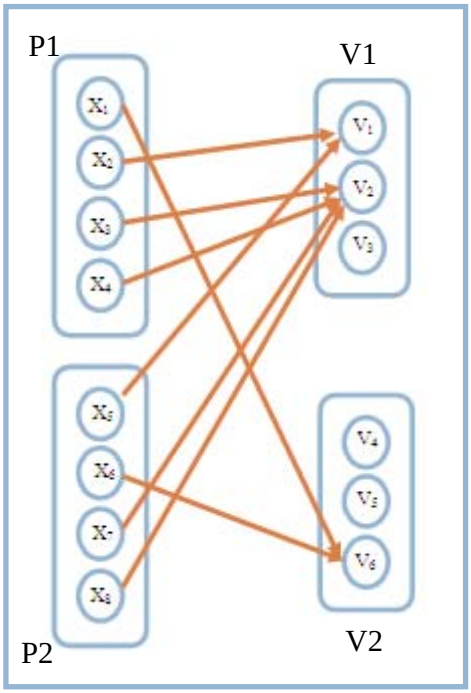


FIGURE 1: Assignment *A*

The signatures for each value are:

I. $v_2 \in V_1$ has signature (2,2)
II. $v_1 \in V_1$ has signature (1,1)
III. $v_6 \in V_2$ has signature (1,1)

4. Static SSB

As shown in [Flener et al, 2006], we can use the signature abstraction to devise a linear set of constraints which provably leaves only one solution in each equivalence class of solutions. We break the symmetries as follows:

I. To break variable symmetry, we force the variables with smaller indices to take smaller or equal values within each variable partition.
II. To break value symmetry, we use the signature of values in complete assignments. Within each value partition we require that smaller values have lexicographically larger or equal signatures than larger values. This is accomplished by using the global cardinality constraint (GCC) which counts how many times a value is taken by a given set of variables.

For the example above, we introduce constraints I-II to break variable symmetry and constraints III-VI to break value symmetry.

I. $X_1 \leq X_2 \leq X_3 \leq X_4$
II. $X_5 \leq X_6 \leq X_7 \leq X_8$
III. GCC ($[X_1, X_2, X_3, X_4]$, $[v_1, v_2, v_3, v_4\ v_5, v_6]$, $[card_{v1}^1, card_{v2}^1, card_{v3}^1, card_{v4}^1, card_{v5}^1, card_{v6}^1]$
IV. GCC ($[X_5, X_6, X_7, X_8]$, $[v_1, v_2, v_3, v_4\ v_5, v_6]$, $[card_{v1}^2, card_{v2}^2, card_{v3}^2, card_{v4}^2, card_{v5}^2, card_{v6}^2]$
V. $(card_{v1}^1, card_{v1}^2) \geq_{LEX} (card_{v2}^1, card_{v2}^2) \geq_{LEX} (card_{v3}^1, card_{v3}^2)$
VI. $(card_{v4}^1, card_{v4}^2) \geq_{LEX} (card_{v5}^1, card_{v5}^2) \geq_{LEX} (card_{v6}^1, card_{v6}^2)$

5. Building the example using Ilog Solver

- Download ssbb.tar.gz file from the webpage and unzip it. The package includes a precompiled library (.a) file, necessary header files (staticSSB.h) and a sample main.cpp together with a make file to compile them all. All source codes are compiled using g++.4-1 and Ilog Solver 6.4 on a Linux machine.

- Compilation and Linking

  Inside the unzipped package type "make main" to compile. You have to change $SOLVERDIR parameter in the make file to your own Ilog Solver directory. In order to link your source codes to our library you can use -L option of g++ compiler. If you are using another compiler please refer to its manual.

- Implementation

  - Include the necessary header files.

```
#include "staticSSB.h"
```

  - Declare the size of variable partitions.

```
int numVarPartitions = 2;
vector<int> varPartitionSize (numVarPartitions);
varPartitionSize[0] = 4;
varPartitionSize[1] = 4;
```

  - Declare the size of value partitions.

```
int numValPartitions = 2;
vector<int> valPartitionSize (numValPartitions);
valPartitionSize[0] = 3;
valPartitionSize[1] = 3;
```

- Declare the minimum and the maximum value that is possible among the domains of all variables.

```
int minValue = 1;
int maxValue = 6;
```

- Define an Ilog environment, model and solver.

```
IloEnv env;
IloModel model(env);
IloSolver solver(model);
```

- Create decision variables. The domains of the variables not necessarily have to be between the minimum and the maximum value. They can be a smaller subset too.

```
IloArray<IloIntVarArray > variables (env, numVarPartitions)

for(i=0; i < numVarPartitions; i++)
{
      for(i=0; i < g.getNumTerminals(); i++)
      {
            variables[i] = IloIntVarArray(env,
                                    partitionSize[i],
                                    minValue,
                                    maxValue);
      }
}
```

- You can decide the order of variable partitions in the value signatures.

```
vector<int> signatureOrder (numVarPartitions);
signatureOrder [0] = 0;
signatureOrder [1] = 1;
```

- Now, you can use **static structural symmetry breaking**.

```
staticSSB(   env, model, variables,
             minValue, maxValue,
             valPartitionSize,
             signatureOrder );
```

8. References

[1] Daniel Heller, Aurojit Panda, Meinolf Sellmann, Justin Yip
Model Restarts for Structural Symmetry Breaking
Proceedings of the 14th intern. Conference on the Principles nad Prctice of Constraint Programming (CP), Springer LNCS 5202, pp. 539-544, 2008.

[2] Meinolf Sellmann and Pascal Van Hentenryck
Structural Symmetry Breaking
Nineteenth International Joint Conference on Artificial Intelligence (IJCAI), 2005

[3] Pierre Flener, Justin Pearson, Meinolf Sellmann, Pascal Van Hentenryck
Static and Dynamic Structural Symmetry Breaking
Proceedings of he 12th intern. Conference on the Principles and Practice of Constraint Programming (CP), Springer LNCS 4204, pp. 695-699, 206

[4] Torsten Fahle, Stefan Schamberger, Meinolf Sellmann
Symmetry Breaking
Proceedings of the 7th intern. Conference on the Principles and Practice of Constraint Programming (CP), Springer LNCS 2239, pp. 93-107, 2001

staticSSB

Category Class

Inheritance Path

Definition File staticSSB.h

Include File <ilsolver/ilosolverint.h>

Method Summary	
public void	staticSSB(&IloEnv, &IloModel, IloArray<IloIntVarArray> variables, int minValues, int maxValues, vector<int> valPartitionSize, signatureOrder)

Description This class provides the staticSSB method to enforce symmetry breaking given the variab

Methods

public void staticSSB (&IloEnv, &IloModel, IloArray<IloIntVarArray> variables, int minValue, int maxValue, vector<int>valPartitionSize, signatureOrder)

This method modifies a given Ilog Environment and Model based on the provided variable and value partitions. The method starts by updating the model with static less than or equal constraints within each variable partition to break variable symmetries. Then, it uses global cardinality constraint (GCC) to obtain the cardinality information of the each value within each variable partition. Signatures are formed for each value using the cardinalities and finally, lexicographic constraints are added to the model for each value partition to break value symmetries. The user has the option to decide on the ordering of variable partition in the signatures.