Figaro: Yet Another Constraint Programming Library

Martin Henz, Tobias Müller, Ng Ka Boon

Abstract
Existing libraries and languages for finite domain constraint programming usually have depth-first search (with branch and bound) built-in as the only search algorithm. Exceptions are the languages claire and Oz, which support the programming of different search algorithms through special purpose programming language constructs. The goal of this work is to make abstractions for programming search algorithms available in a language-independent setting.

Figaro is an experimentation platform being designed to study non-standard search algorithms, different memory policies for search (trailing vs copying), consistency algorithms, failure handling and support for modeling. This paper focuses on the use and implementation of such abstractions for investigating programmable search algorithms and memory policies in a C++ constraint programming library.

1 Introduction

Languages and libraries for finite-domain constraint programming (CP(FD)) allow to solve finite-domain problems through exhaustive constraint propagation, interleaved with non-deterministic strengthening of a constraint store, leading to the exploration of a search tree.

Languages for CP(FD) allow a semantic embedding of CP(FD)-specific features. Prolog-based languages such as CHIP [DVS+88] semantically embed depth-first search by inheriting Prolog’s resolution, and the languages claire [CL96] and Oz [Smc95] semantically embed more generic constructs that allow to program search algorithms other than depth-first search. Libraries such as PECOS [Pug92] and llog Solver [IL097] are confined to general-purpose programming languages that do not provide such support.

We show in this work how to support programmable search algorithms in a C++ library by representing constraint stores as data objects. We call the C++ library Figaro, since its implementation reuses parts of the Mozart system [Moz99]. The distinguishing feature of Figaro from other libraries and systems is the relative addressing of propagators and variables in stores, which allows a clean separation of tree search algorithms from search heuristics and supports both copying-based and trailing based search.

We present the design of Figaro by introducing stores, variables and propagators in Section 2, the notion of search trees in Section 3, and search algorithms in Section 4.

Section 5 shows how relative addressing allows to use copying-based search in addition to trailing-based search. Finally, Section 6 describes related work and further directions.

2 Variables and Propagators

The constraint store in CP(FD) contains the current domain of each variable of the constraint problem, i.e. the set of possible values it can take. For example, for the usual model of the n-queens problem, we introduce variables $x_i$, $0 \leq i < n$ whose initial domains \{0, ... , $n-1$\} represent all possible rows in which the queens of column $i$ can be positioned. In [HS99], search algorithms use a data structure (called “rooms” in that paper) representing a store. Such a store data structure host variables and propagators and support search. In an object-oriented setting, it is natural to introduce a class store. The class store is related to the built-in data type of spaces in Oz [Sch97] and the class 1lManager of Ilog Solver [IL97] (for a comparison, see Section 6). Variables are introduced by requesting a new variable with initial domain from 1 to $h$ from a store.

```cpp
class store {
    private: ...
    public:
        var newvar(int lo, int hi);
        var getlo(var v);
        var gethi(var v);
    ...};

    For the purpose of this discussion, let us assume that var is an abstract data type whose values represent variables. In Section 5, we further discuss the var type.

    Using the store abstraction, we can introduce variables for the n-queens problem as follows. Here we employ vectors as provided by the Standard Template Library [SL95] for C++.

```cpp
int main(int argc, char * argv[]) {
    int n = atoi(argv[1]); // number of queens
    store * s = new store(); // create new store
    vector<var> vars(n);    // declare variable vector
    for (int i = 0; i < n; i++) // create n variables;
        vars[i] = s->newvar(0, n-1); // one for each row
    ...}
```

The no-attack constraints can be expressed using three constraints that constrain all variables in a given vector to be pairwise distinct modulo a given offset. Thus given a vector vars of $n$ variables and a vector offset of $n$ integers, the constraint distinctOffset(vars, offset) expresses that for every $i$ and $j$, where $0 \leq i, j \leq n; i \neq j$, the constraint $vars_i + offset_i \neq vars_j + offset_j$ holds. The implementation of the propagator distinct_offset is taken from the C++ constraint programming interface of the
Program 1 Constraints for the N-Queens Problem

```c
int main(int argc, char * argv[]) {
    int n = atoi(argv[1]);  // number of queens
    store * s = new store();  // create new store
    vector<var> vars(n);  // declare variable vector
    for (int i=0;i<n;i++)  // create n variables;
        vars[i]=s->newvar(0,n-1);  // one for each row
    vector<int> offset(n);  // vector for offset
    for (int i=0;i<n;i++) offset[i]=0;  // horizontal no-attack
    distinct_offset(s,vars,offset);
    for (int i=0;i<n;i++) offset[i]=i;  // diagonal-up no-attack
    distinct_offset(s,vars,offset);
    for (int i=0;i<n;i++) offset[i]=n-i; // diagonal-down no-attack
    distinct_offset(s,vars,offset);
    ...
}
```

Mozart system [MW97]. Reusing Mozart’s propagators significantly reduces the implementation effort for Figaro.

In Figaro, constraints are represented by classes which extend an abstract class *propagator*. Propagators are created with a given store, variables and auxiliary values.

```c
class distinct_offset : public propagator {
    public:
        distinct_offset(store * s, vector<var>, vector<int>...)

Using the class distinct_offset, the 5-queens problem can be expressed as in Program 1.

The creation of propagators will immediately compute the fixpoint with respect to all propagators in the store, according to the propagators’ consistency algorithms. In this process, propagators may tell new domains for variables.

The member function tell of stores allows to narrow the domain $d_1$ of a given variable such that it contains only values from the domain $d_2$ passed to tell. If the intersection of $d_1$ and $d_2$ is empty, a failure occurs.

```c
store::tell(var v, int lo, int hi);
```

If the intersection of $d_1$ and $d_2$ is empty, a failure occurs. Such failures are crucial for constraint programming, since they allow to prune the search tree. As a generic way to indicate failure to search algorithms, failing tell operations raise the C++ exception `Failure()` (see discussion on C++ exceptions in Section 6).
3 Search Trees

Usually propagation alone does not suffice to solve constraint problems. Non-deterministic search is necessary, which explores a search tree in a top-down manner. From a node to a child node, constraints are added. At each node, the fixpoint with respect to all propagators is reached before the resulting constraint store is used to devise a suitable constraint for a child node. In that manner, search trees are created dynamically, at each point exploiting the current information in the constraint store. Search trees are represented in Figaro using instances of an abstract class node.

class node {
public:
  virtual node * child(store *, int)=0;
};

The member function child of node is given an integer i and returns its ith child. Often, search trees are constructed by fixing one variable v of a given set of variables to a value x in the left child (i = 0) and excluding x from the domain of v in the right child (i = 1). Such a tree is called enumeration tree. The class in Program 2 represents naive enumeration, where the variables of a given vector are enumerated from left to right, starting with the smallest values in their domains.

By recursively applying the child function to it results, we are able to explore an enumeration tree. The tree subtree is returned when all variables are enumerated. It can be used to place another search tree at the leaves of the enumeration tree, or to collect solutions. For example, the class in Program 3 allows to display a solution to the n-queens problem.

Program 2 Naive Variable Enumeration

class naive : public node {
private:
  int idx; vector<var> vars; node * subtree;
public:
  naive(vector<var> vs, int i, node * t) : vars(vs), idx(i), subtree(t) {}
  node * child(store * s, int i) {
    if (i==0) {
      s->tell(vars[idx]), s->getlo(vars[idx]), s->gethi(vars[idx]));
      return (idx+1==vars.size() ? subtree : new naive(vars, idx+1, subtree));
    }
    else {
      s->tell(vars[idx]), s->getlo(vars[idx])+1, s->gethi(vars[idx]));
      return new naive(vars, idx, subtree);
    }
  }
};
Program 3 A Node Class for Printing Solutions

class queens_printer : public node {
public:
    queens_printer(const vector<var> vs) {
        for (int j = 0; j < vs.size(); j++)
            cout << "cl: " << vs[j] << "\nrow: " << s->getlo(vs[j]) << "\n";
    }  
    node * child(store * s, int i) {return NULL;}
}

Note that such queens_printer nodes are leaves, because their child member function returns the NULL pointer. Thus, queens_printer nodes can be used as subtree of enumeration trees.

4 Programming Inference Algorithms

During the exploration of a search tree, failure may occur as a result of applying the child function of a node. That means one of the decisions leading to the corresponding node was the wrong one. Unfortunately, after that decision was taken the store has changed through creating variables and propagators and telling domains. In order to undo these changes and trying an alternative, we introduce the following operations on stores.

mark store::mark();
void store::backtrack(mark m);

The function store::mark returns a value that represents the current state of the store, and the function backtrack undoes all changes done to the store since the given mark was obtained. A search algorithm using store::mark and store::backtrack is given in Program 4.

Program 4 First-solution Depth-first Search

node * solve_one(store * s, node * t) {
    if (t == NULL) return t;
    int m = s->mark();
    try {return solve_one(s,t->child(s,0));}
    catch (Failure) {
        s->backtrack(m);
        return solve_one(s,t->child(s,1));
    }
}

Using this search algorithm, we are finally able to solve the n-queens problem as shown in Program 5.
Program 5 Solving N-Queens with Figaro

```c
int main(int argc, char * argv[]) {
    int n = atoi(argv[1]);
    store * s = new store();
    vector<var> vars(n);
    ...
    try {solve_one(s,new naive(vars,0,new queens_printer(vars)));
    } catch (Failure) {printf("no solution\n");}
}
```

Note that in the exposition above, we used several simplifications to clarify the design underlying Figaro. Both enumeration and search can be improved significantly by introducing additional tests and member functions. For instance, the creation of node objects can be avoided, when choosing the left child node of an enumeration node by incrementing the idx member of the parent.

5 Copying-Based Search and Relative Addressing

Note that in the previous section the same store is passed between solve_one and child. Search is done entirely by trailing and backtracking, as in most constraint programming systems. Schulte [Sch99] shows that copying-based search as employed by the Mozart system, combined with recomputation of spaces, can compete with the performance of trailing-based systems. To study the performance of memory policies, it appears to be attractive to provide both copying and trailing in the same system. In order to support copying-based search, we use a suitable C++ copy constructor.

```c
class store {
    ...
public:
    store(const &store);
};
```

For combining copying and trailing-based search, we propose that in the copy, all marks are removed and that no information is trailed in a store before the first mark is obtained.

Since we pass the store, on which a node operates, explicitly to the store, it is straightforward to use copying-based search in our setup. We illustrate this using example of limited discrepancy search (LDS), a search algorithm proposed by Harvey and Ginsberg [HG95]. LDS addresses the question how to avoid getting stuck in a small leftmost subtree in the presence of a strong heuristic for building the search tree. Let us assume that a script uses a heuristic which generates binary nodes whose left child are considered much more likely to lead to a solutions than the right child. Then the number of discrepancies of a solution is the number of right children in the path from the root to the solution. LDS prescribes to search for solutions with a small number of discrepancies first.
Program 6 A Copying-based Search Algorithm For Limited Discrepancy Search

```c
node * probe(store * s, node * t, int d) {
    if (t=NULL) return t;
    if (d > 0) {
        store * s1 = new store(s);
        try {return probe(s, t->child(s,1), d-1);}  
        catch (Failure) {
            return probe(s1, t->child(s,0), d);}
    } else return probe(s, t->child(s,0), 0);
}
void lds_one(store * s, node * t, int d) {
    try {return probe(s, t, d);}  
    catch (Failure) {lds_one(s, t, d+1);}
}
```

The algorithm lds_one given in Program 6 searches for a solution according to LDS
assuming that a solution exists with increasing number of discrepancies, starting with
a given d, typically 0. The auxiliary function probe returns a solution within a given
number of discrepancies d, if such a solution exists. Note that once the number of allowed
discrepancies has reached 0, there is no need to make copies any longer. Instead, probe
descends straight down towards a solution.

In copying-based search, tree descriptions will use var values that stem from calls of
newvar on a store and apply operations such as tell to copies of the store. Thus
variables must be invariant with respect to copying. We achieve this invariance by using
as var values the relative address of the variable in the store data structure. In stores, a
dynamic array keeps track of variables, variables are represented by their indices in this
array.

#define var int

The same technique, we use for propagators; trail entries and propagator lists for vari-
ables use relative addresses. Note that relative addressing makes copying particularly
easy, whereas absolute addressing as employed by the Mozart system and Ilog Solver
necessitates recursive traversal of data structures in stores.

6 Directions and Perspectives

The design presented here has been inspired by a proposal for an ML library [HS99], in
which data structures for representing constraint stores were called rooms. The design
presented here has been used to develop a modular architecture for programming search
algorithms [CHN00]. In this architecture, aspects of search algorithms such as the mem-
ory policy, optimization, interactivity and search tree visualization can be programmed
independently. The goal to allow the programming of different search algorithms in a li-
brary for finite domain constraint programming has been apparently recognized recently
in the Ilog Solver library as briefly mentioned in [LP99].
Figaro is currently being implemented by reusing parts of the Mozart system. In particular Mozart’s sophisticated representation of domains and its propagation algorithms such as serialization and cumulative constraints for scheduling applications are being reused. We hope by this, we can reduce the development time of Figaro.

Representing Constraint Stores
Search in Oz [Smc95] is programmable through the abstraction of a space [Sch97]. The store abstraction was inspired by spaces and shares with them the ability to manipulate constraint stores together with their variables and propagators as data. However, spaces are tightly integrated into the language Oz such that the space with respect to which variables and propagators are introduced is kept implicit. In addition to variables and propagators, spaces host threads. The programming of search engines in Oz amounts to communicating and synchronizing with the threads of spaces. Due to the concurrent setup, search in Oz is based on cloning of spaces, whereas our approach supports both backtracking and copy-based search.

In Ilog Solver [LO97], constraint stores are represented by instances of the C++ class IManager. The incremental building of the search tree is supported through data structures, called goals, which are installed in manager objects. The separation of stores from node objects for search in Figaro allows to cleanly separate distribution from tree search algorithms and supports copying-based search well (see next section).

Memory Policy
Relative addressing of variables and propagators in stores in Figaro, which is not present in Ilog Solver, is the the key feature that allows to use copying-based search in addition to trailing.

Both copying-based and trailing-based search can be combined with recomputation [Sch99]. We hope that the flexibility to use both memory policies leads to interesting, possibly adaptive, and more efficient combinations of the two memory policies and recomputation.

Another memory management issue is the creation of nodes, which are in the presented simplified design not explicitly deallocated. We are currently experimenting with search algorithms that explicitly deallocate nodes as well as with automatic memory management systems for C++.

Consistency Algorithms
Different constraint programming languages and libraries use different consistency algorithms, usually variants of AC3 and AC5. In practice, the trade-off between the two seems to be to use more elaborate data structures and reduce pure computation time (AC5) versus simpler data structure and redundant computations (AC3). This trade-off becomes interesting in the light of a widening gap between computation speed and memory access speed in modern processors. We hope to carry out practical experiments on consistency algorithms with Figaro.
Representation of Failure

As in the ML design [HS99], we treated failure by exceptions in this paper. Another possibility would be to let stores assume a failure state, when an exception is encountered. This design issue is not settled yet and depends on practical considerations such as the efficiency of exception handling in C++ compilers. Benchmarks will shed some light on the efficiency of the two mechanisms for failure.

Modeling and Interfacing through Scripting Languages

Figaro is designed as a C++ library for constraint programming. Typically, the library will be linked to applications that make use of constraint programming for problem solving. However, in some applications the need for a more formal and flexible formulation of constraint problems arises. This need is addressed in symbolic programming languages for constraint programming (Prolog-based, Oz, CLAIRE) and in modeling languages for constraint programming such as OPL [Hen99]. To address this need, we provide a generic interface to scripting languages such as Tcl and Perl. In addition to modeling, the use of scripting languages aids the development of and experimentation with the library and improves interoperability.

Acknowledgements

Gert Smolka collaborated on the development of the room concept and corresponding abstractions in an ML setting, which provided a blueprint for stores in Figaro. Christian Schulte helped us see the importance of indirect addressing and pointed out references. The project benefited from a travel grant from the National University of Singapore (project ReAlloc) and the hosting of the third author by the Programming Systems Lab, Saarbrücken.

References


