

# Implementing Tuple Variables in Gecode

## Bachelor Thesis

### Uppsala University – Autumn 2014

Patrik Broman

September 29, 2015

#### Abstract

This is the abstract

## Contents

<b>1</b>	<b>Background</b>	<b>2</b>
1.1	Constraint Programming . . . . .	2
1.2	Gecode . . . . .	2
1.3	The need for Tuple Variables . . . . .	3
<b>2</b>	<b>Introduction</b>	<b>3</b>
<b>3</b>	<b>Tuple Variables</b>	<b>3</b>
3.1	Creating a New Variable in Gecode . . . . .	3
3.2	Restrictions . . . . .	6
3.3	Implementation of Exact IntPair . . . . .	6
3.4	Implementation of Approximate IntPair . . . . .	6
3.5	Correctness verification . . . . .	6
3.6	Performance Test . . . . .	6
<b>4</b>	<b>Related Work</b>	<b>7</b>
4.1	Implementation of bit-vector variables in a CP solver . . . . .	7
4.2	A propagator design framework for constraints over sequences . . . . .	7
<b>5</b>	<b>Conclusions and future work</b>	<b>7</b>
5.1	Rewrite the implementation in such a way that it may be accepted in a new Gecode release . . . . .	7
5.2	Test different implementations with different domain representations . . . . .	7
5.3	Extend the tuples to arbitrary dimensions and not just pairs . . . . .	7
5.4	Extend the tuples to floats . . . . .	7
5.5	Reach better performance for a specific problem with an array of pair variables than with two arrays of integer variables . . . . .	7
<b>6</b>	<b>Git repository</b>	<b>7</b>

# Acknowledgements

Thanks to Jean-Noël Monette and Pierre Flener for being helpful supervisors.

Thanks to Joseph Scott for helping me configuring Gecode.

Thanks to Johan Gustafsson for general help with C++.

## 1 Background

### 1.1 Constraint Programming

Constraint programming (CP) is a method to solve problems by modeling the problem with variables and constraints that the variables must fulfill. The variables have an initial domain of possible values. The CP solver then uses the constraints to remove impossible values.

A very typical problem suitable for constraint programming is a sudoku. The most intuitive model is a  $9 \times 9$  matrix of integer variables. The constraints that all variables have a domain of  $\{1..9\}$ , and that all variables in a row, column and box must be different. For a specific sudoku constraints are added to require specific variables to have predetermined values.

In constraint programming, different variable types may be used. Gecode comes with boolean, integer, set and float variables. The available variable types limits the options to model a specific problem. With more variable types a specific problem can be modeled in different ways. There is a problem called Eight queens<sup>1</sup>. The problem is to place eight queens on a chess board in such a way that no queen threatens another. If only boolean variables are available, the problem would have to be modeled with 64 boolean variables. With integer variables it would be sufficient with only 8 variables. In the boolean case each variable  $b[i][j]$  would tell if a tile has a queen or not, while in the integer case each variable  $x[i]$  would tell on which row the queen in column  $i$  should be placed on. By analyzing the Eight queen problem it is easy to realize that a solution must have one queen per column.

In principal the CP-solver loops propagating and branching until the problem is solved. The propagators has two main tasks. The most important task is to determine if a solution has been found or not. A solution is a state where all variables are assigned, which means their domain has been pruned (shrunk) to exactly one, and all propagators reports that the values assigned to the variables satisfies the constraints. The second task is not mandatory, but crucial for performance. A propagator can prune values from a variable's domain. For example, if a variable  $x$  has a domain of  $\{1..9\}$  and the constraint  $x \leq 4$  is used, a propagator could instantaneously remove the values  $\{5..9\}$  from the domain.

When no propagator can prune any values a brancher takes over. The brancher typically splits the variable it is connected to into two parts, one with an assigned value and one with the rest of the domain, thus creating a search tree. After this branching the propagators are called again. The branchers determines the structure of the tree, and the search engine determines how the tree will be traversed.

### 1.2 Gecode

Gecode is an open source CP solver programmed in C++ and is licensed under the MIT license. It is flexible and supports customization of most parts, including variable types, branchers and propagators. It is suitable for both education and professional use. In this thesis, Gecode 4.3.2 is used.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Eight\\_queens\\_puzzle](https://en.wikipedia.org/wiki/Eight_queens_puzzle)

For more information, have a look at the homepage. <http://www.gecode.org>

### 1.3 The need for Tuple Variables

When using a CP solver, it is desirable to minimize the search tree to avoid brute force searching. The size of the search tree depends on the model and the propagators. With good models and propagators the variable domains can be pruned to smaller sizes, keeping the branching and searching to a minimum. For example, if two variables  $x$  and  $y$  have domains  $dom(x) = \{1, 2, 3\}$  and  $dom(y) = \{2, 3, 4\}$ , and the constraint  $y < x$  is applied the domains can be pruned to  $dom(x) = \{3\}$  and  $dom(y) = \{2\}$ . If we instead have the same variables but add the constraint  $x \neq y$  nothing would be pruned, since for all values in both domains there exists solutions.

In this example the problem was modeled with two integer variables. A tuple is an ordered list of elements with fixed length. If the problem would be modeled with one tuple variable instead of two integers pruning would be possible. Then we would have a tuple variable  $t$  with domain  $dom(t) = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle, \langle 3, 4 \rangle\}$  and pruning with the same constraint would yield  $dom(t) = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 1, 4 \rangle, \langle 2, 3 \rangle, \langle 2, 4 \rangle, \langle 3, 2 \rangle, \langle 3, 4 \rangle\}$ . This shrinks the size of the search tree from 9 to 7.

If we instead look at the case where  $dom(x), dom(y) = \{1..1000\}$  and the constraint  $x = y$  the benefits are more obvious. Using integer variables no values can be pruned. If tuple variables are used, the size of the domain would shrink from  $10^6$  to  $10^3$ . This indicates that tuple variables can be quite useful. Note though that these two examples are selected because they are easy to understand, and in reality the performance would probably drop if tuple variables were used for them.

The main benefit of a tuple is the possibility to link variables together. The most obvious case is when dealing with coordinates since a coordinate by definition is a tuple, but it can be generalized to any case where it is interesting to see if a combination of two values are part of a solution.

## 2 Introduction

In this thesis only two dimensional tuples, so called 2-tuples are considered. The main reason is to keep it simple, but it is also likely that 2-tuples would be the most common to use to use.

## 3 Tuple Variables

### 3.1 Creating a New Variable in Gecode

#### 3.1.1 Specification File

Creating the specification file is the first thing that needs to be done when designing a new variable. A simple specification file could look like this:

```
[ General ]
Name:      IntTuple
Namespace: MPG::IntTuple
[ ModEvent ]
Name:      FAILED=FAILED
[ ModEvent ]
Name:      VAL=ASSIGNED
[ ModEvent ]
```

Name:	NONE=NONE
[ PropCond ]	
Name:	NONE=NONE
[ PropCond ]	
Name:	VAL=ASSIGNED
ScheduledBy:	VAL
[ End ]	

There are three different sections in the file. The specification file must start with the [General] section and end with [End]. There are also sections for modification events and propagation conditions. The specification file is used by a configuration script to generate base classes.

The [General] section is trivial. This is where the name and namespace of the variable are specified. Both name and namespace are arbitrary. In Gecode the standard variables have namespace Gecode::VarName and in the examples in the documentation they instead have the namespace MPG::VarName.

The [ModEvent] section specifies the modification events. The modification events describes how the variables changes. When a propagator wants to prune the domain for a variable it tells the variable to do that, using the class methods that the variable implementation provides. If no values are pruned, the variable returns the modification event NONE, and if all values are pruned it returns FAILED. When the domain of a variable gets pruned to one ASSIGNED is returned. For all variables, modification events for NONE, FAILED and ASSIGNED are required. More modification events may be added to avoid unnecessary executions of propagators. For example, the modification event BND can be used when only the boundaries of a domain has changed.

The [PropCond] section describes how the propagators are scheduled, depending on how the variables have changed. For all variables, propagator conditions for NONE and ASSIGNED are required.

### 3.1.2 Variable Implementation

The variable implementation class would with the specification file in the previous section be called IntTupleImp and inherit from the class IntTupleVarBase generated from the specification file. It has to implement a function called assigned that returns true if the variable is assigned.

### 3.1.3 Variable Class

The variable class is the class that is used when modeling the actual problem and inherits from the variable implementation. It is a read only user interface, and a programmer that does not program any custom branchers or propagators will not use anything else than this. This class would be called IntTupleVar.

### 3.1.4 Deltas

### 3.1.5 Views

The variable implementation class does implement methods for modification of the variable, but the variable class does not give access to these. Instead these are called from a view, which is a read and write interface to the implementation class.

### 3.1.6 Propagators

A propagator implements constraints and prunes variables. They interact with the views and not variable class and have four possible return values.

- `ES_FAILED` - There are no values in the domain that satisfies the constraint. The variable is failed.
- `ES_FIX` - The propagator is at a fixpoint. It is impossible for the propagator to prune more values before the domain has changed for any of the variables the propagator depend on by either a brancher or another propagator.
- `ES_NOFIX` - The propagator may be at a fixpoint, but it is not guaranteed. Running it once more may or may not prune some values. Unless it is at a fixpoint for sure, this should be used.
- `ES_SUBSUMED` - The propagator is done. Regardless of any changes made to the variable this particular propagator will not be able to do anything more.

The propagator must implement `ES_FAILED` and at least one of the others. `ES_NOFIX` is safe to return in the sense that it does not promise anything. Furthermore, the propagator needs the following methods:

- `post` - The method for posting a constraint, which basically is saying that a certain constraint should be used.
- `disposal` - This works as a destructor. The only reason to not have a regular destructor is that destructors in C++ can't take arguments.
- `copy` - A method to copy the propagator.
- `cost` - Estimates the cost to run the propagator. This does not affect the correctness, but may improve efficiency by making it easier for the Gecode engine to schedule cheap propagators before expensive ones.
- `propagate` - The method which does the actual work. It prunes domains and determines the status to return.

A propagator must also subscribe to variables, that is, telling Gecode when the propagator should be awoken. For instance, a propagator that says that  $x$  should be less than  $y$  should subscribe to the maximum value of  $x$  and minimum value of  $y$ .

**Consistency** A propagator can work with different consistencies. Bound consistency means that the propagator ensures that the bounds (min and max value for an integer variable) satisfies the constraint. If if  $dom(x) = 1, 5, 7, 8$  and  $dom(y) = 1, 3, 7$  and the constraint  $x = y$  is applied, the propagator would yield  $dom(x) = 1, 5, 7$  and  $dom(y) = 1, 3, 7$  with bounds consistency, since there are solutions to  $x = 1, x = 7, y = 1$  and  $y = 7$ .  $x = 5$  and  $y = 3$  is not considered. Pruning the whole domain does in general cost more execution time.

### 3.1.7 Branchers

When there are no propagators able to prune any variables the CP solver must start searching. The branchers determines how this is done. The simplest brancher just puts the first value in the first branch and the rest of the values in a second branch. Another way is to pick a random

value in the domain, but more sophisticated ways exist. For the IntPair variable only a simple brancher is implemented. The domain of IntPair is just a long sorted list, and the brancher picks the first value.

A brancher must implement the following methods:

- status
- choice
- commit
- print
- ngl

### **3.2 Restrictions**

To keep things simple only two-dimensional integer tuples are considered, so called Pairs.

### **3.3 Implementation of Exact IntPair**

The exact version does not approximate the domain, that is, the implementation is a long list of all integer pairs that's currently in the domain.

### **3.4 Implementation of Approximate IntPair**

This version of IntPair approximates the domain by just storing one dimension exactly. For the other dimension, only the boundaries are stored. The benefit is vastly less use of memory, but the drawback is that it can't be pruned as efficient as the exact version.

### **3.5 Correctness verification**

The correctness is verified in several steps. For all problems an assertion is made to ensure that the solution printed actually is a solution before the solution is printed.

### **3.6 Performance Test**

To test the performance the same problem is solved in three different ways. Two of them is IntPair, one exact and one with one dimension approximated. The last one is solved with the integer variable that comes with Gecode, and is just for reference. The problem is a DFA with costs. The executables accepts six arguments:

- seed The seed for the random generator to generate the same DFA for all executables.
- states Number of states.
- tokens Size of the alphabet.
- cost Max cost per transition in the DFA. This is decreased by one, so setting this to 1 means no cost.
- max cost Maximum cumulative cost for the variables.
- steps Number of steps in the DFA.

### 3.6.1 First initial performance test

An initial naive test with states=10, tokens=4, cost=3, max cost=10, steps=10 gave the following result:

```
int      1m15.252s
approx   2m16.260s
exact    0m14.474s
```

Another test with states=15, tokens=2, cost=20, max cost=200, steps=13 gave:

```
int      0m53.991s
approx   0m7.351s
exact    0m40.735s
```

This is very interesting. In the first test the exact version of IntPair is the clear winner, and in the second test it is the approximate version that wins. The difference is definitely significant with 14 vs 75 and 7 vs 54 seconds. What makes this even more interesting is that while the built in IntVar is highly optimised, both implementations of IntPair is the opposite. The whole variable implementation with modification events and the brancher uses poor algorithms and are just enough to get the tests to compile and run. The only code written for the built IntVar is the propagator, and even that one is way better written than the propagators for IntPair. While the IntPair propagators are very naive, the Int propagator uses optimisations like preallocating space for the vectors and uses binary search instead of linear search.

## 4 Related Work

### 4.1 Implementation of bit-vector variables in a CP solver

### 4.2 A propagator design framework for constraints over sequences

## 5 Conclusions and future work

### 5.1 Rewrite the implementation in such a way that it may be accepted in a new Gecode release

All big software libraries have coding standards. There is plenty of work to rewrite the code for IntPairs in such a way that it satisfies Gecodes standard.

### 5.2 Test different implementations with different domain representations

### 5.3 Extend the tuples to arbitrary dimensions and not just pairs

This would require a complete rewrite of the variable implementation.

### 5.4 Extend the tuples to floats

## 6 Git repository

<http://github.com/klutt/gecode-tuples>

## References

- [1] Calculus, Adams
- [2] Operating systems and its concepts, Galvin