Implementing Tuple Variables in Gecode
Bachelor Thesis
Uppsala University

Patrik Broman

March 10, 2016

Abstract

In constraint programming, efficient pruning is vital to avoid unnecessary search. This thesis shows that tuple variables can be used to reduce the size of the search tree, which in turn may reduces the execution time. The tuple variables have been implemented in C++ for the Gecode library. The problem used to measure performance is finding paths through a deterministic finite automaton with transition costs.

Contents

Acknowledgements

1 Introduction

Constraint programming (CP) is a declarative style programming, and is very similar to solve equations in mathematics. A problem is modeled with variables, which must satisfy certain conditions. These conditions are called *constraints*. The variables are initialized with a set of possible values. Such a set is called a *domain*. The main process is to remove values not satisfying the constraints from the domain until only one is left. A state where all variables have a domain with exactly one value each is called a *solution*.

When using CP, it is often a problem that it is impossible to remove invalid combinations of values from the domains. Consider a problem where we have the variables x and y. We initialize the domains $dom(x) = \{1, 2, 3, 4\}$ and $dom(y) = \{3, 4, 5, 6\}$. We also use the constraint $x = y$. In this case we can instantly remove 1 and 2 from x, and 5 and 6 from y. This leaves us with $dom(x) = \{3, 4\}$ and $dom(y) = \{3, 4\}$. Now, consider the same initial domains, but with the constraint $x \neq y$. We know that no solution has $x = y$, but we cannot remove any value from any of the domains.

Many CP problems could be solved more efficiently if it was possible to remove a combination of values from the variables, rather than removing the values individually. Recently, it has been shown in a paper [5] that a *tuple variable* can be used to link variables together. A *tuple* is a mathematical object: It is an ordered list of elements. In this thesis, a *tuple variable* is a variable whose domain contains tuples of integers. Currently, Gecode supports Boolean, integer, integer set and float variables. The tuple variable type described here is an extension to the Gecode library. It is shown that it is possible to reach better performance by linking variables with tuples.

To demonstrate that tuple variables can be used to increase performance, I have implemented two different variants of tuple variables in two dimensions. One of them keeps an exact representation of the domain, while the other approximates the second dimension by just storing the boundaries. Individual boundaries are stored for each value in the first dimension. I wrote propagators for these tuple variable implementations and a reference propagator to compare performance. Because the main purpose of this thesis is to show that tuple variables can improve performance, I have intentionally avoided to optimize the code that handles the tuple variables. Even obvious and simple optimizations have been avoided. For the same reason, I have optimized the reference propagator as far as I can, and I have also asked more experienced CP-programmers for help with that.

To compare performance, I used the tuple variables to find all valid strings for a deterministic finite automaton (DFA) with costs for each transition (cDFA). A cDFA does not only tell whether a string is accepted or not, but also the total cost for that particular string. The performance is measured by studying how the execution time and size of the search tree depend on four parameters: the number of states, the size of the alphabet, the maximum cost per transition, and the size of the string. Both DFAs and cDFAs are described in detail in Section 2.3.

2 Background

2.1 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. Then *propagators* are used to remove values not satisfying the constraints. This is called *pruning*. Propagators are explained in detail in Section 2.5.6. CP is a *declarative* type of programming, which basically means that the programmer states what must hold, without stating how this is achieved. This is very much like mathematics. The *constraints* work like equations.

A problem suitable for CP is a Sudoku. The most intuitive model uses a 9×9 matrix of integer variables. The constraints are that all variables have a domain of $\{1, \ldots, 9\}$ and that all variables in a row, column, and box must be different. For a specific Sudoku, constraints are added to require certain variables to have predetermined values.

When using CP, different variable types may be used. Gecode comes with Boolean, integer, integer set and float variables. The available variable types limit the options to model a specific problem. With more different variables available, a specific problem can be modeled in different ways. Consider the problem Eight queens[1]: The problem is to place 8 queens on a chess board in such a way that no queen threatens another queen. If only Boolean variables are available, the problem has to be modeled with 64 Boolean variables. With integer variables, 8 variables is sufficient. In the Boolean case, each variable $b[i][j]$ would tell whether a tile has a queen or not, while in the integer variable case each variable $x[i]$ would tell on which row the queen in column i should be placed on. By analysing the Eight queen problem, it is easy to realise that a solution must have exactly one queen per column.

When no propagator can prune any value, a *brancher* takes over. The brancher typically splits the domain of a variable 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 for all branches. The branchers determine the structure of the search tree. Branchers are described in Section 2.5.7.

Solving a CP-problem means alternating propagating and branching until some goal has been reached. The goal can, among others, be finding a solution, finding all solutions, or discovering that there are no solutions at all. The propagators have two main tasks. The most important task is to determine whether a *solution* has been found or not. A *solution* is a state where all variables are *assigned*, which means the size of their domains are exactly one, and all propagators report that the values assigned to the variables satisfies the constraints. A state where at least one variable has been pruned to size zero is called a *failure*. The second task for a propatator is pruning. A propagator does not necessarily be able to prune, but otherwise the branchers need to do all the work. In order to reach good performance, the propagators needs to prune as many values as possible. For example, if a variable x has a domain of $\{1, \ldots, 9\}$ and the constraint $x \leq 4$ is used, a propagator could instantaneously remove the values $\{5, \ldots, 9\}$ from the domain.

Note that the domain of a variable never grows, and no value changes. The only thing that happens to a domain after its initialisation is that values are removed by either a propagator or a brancher. In the search tree, the variable domains in a certain node is always a subset of the variable domains of the parent node.

---

[1]https://en.wikipedia.org/wiki/Eight_queens_puzzle

2.2 Tuple

In a mathematical context, a tuple is a finite length sequence (ordered list) of elements. The elements can be of any type, such as numbers, sets, or functions. The elements do not need to be of the same type. An n-dimensional tuple is called an n-tuple. In this thesis, a tuple is denoted using angle brackets. For example, $\langle 3, 2, 1, 4, 2 \rangle$ is a 5-tuple.

2.3 Deterministic Finite Automaton

A deterministic finite automaton (DFA) is a 5-tuple: $\langle Q, \Sigma, \delta, q_0, F \rangle$, where Q is a finite set of states, Σ is called the *alphabet* and is a finite set of symbols, $\delta : Q \times \Sigma \mapsto Q$ is a transition function mapping a state and a symbol to a state, $q_0 \in Q$ is called the *starting state*, and $F \subseteq Q$ is a set of states called *accepting states*. [4] A state from which there is no path (via one or more transitions) to an accepting state is commonly called a *garbage state*. Furthermore, the set of garbage states is commonly viewed as one state, called *the* garbage state. Usually the garbage state is left out in both graphical and algebraic representations.

A DFA *consumes* symbols from a string from the beginning to the end. If the DFA is in an accepting state when all symbols in the string have been consumed the DFA *accepts* the string. Figure 1 shows a 2-state DFA with omitted garbage state.
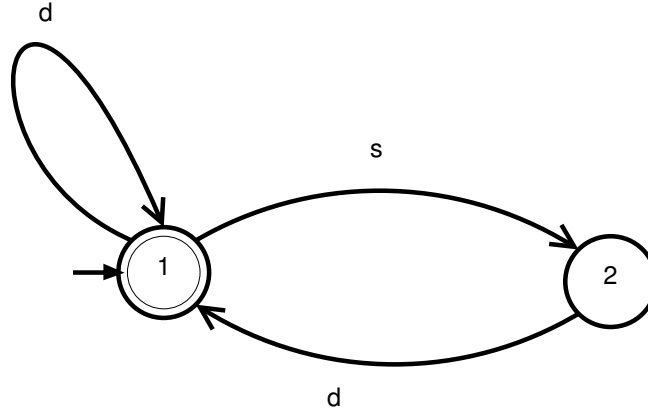

Figure 1: A DFA for the Swedish drinking protocol, as it is shown in the Gecode documentation [6]. State 1 is the starting state and the only accepting state. State 2 is a non-accepting state. The symbols d and s denotes drinking and singing. The DFA means that you may drink whenever you want, but if a song has been sung you have to drink before you leave. The regular expression for this DFA is $(d|sd)^*$

DFAs are closely related to regular expressions (regex). Every regex corresponds to a DFA and vice versa. Note that this is not true for the extended regexes one can find in many languages, such as Perl and Python. A pure regex can be written with the operators (,), |, and *. One example that uses all of them is $ab(a|b)c^*$ which is a regex for the language of all strings starting with ab, continuing with either an a or a b, and ends with zero or more c. A corresponding DFA for this regex is $Q = \{1, 2, 3, 4\}, \Sigma = \{a, b, c\}, q_0 = 1, F = \{4\}, \delta(1, a) = 2, \delta(2, b) = 3, \delta(3, a) = 4, \delta(3, b) = 4, \delta(4, c) = 4$. Any other parameter for δ returns the garbage state. One practical implication of this definition is that a pure regex can not check for matching parentheses in a string.

In this thesis an extended version of a DFA, a cost DFA (cDFA), is used. The difference is that the transition function not only maps a state and a symbol to a state, but to a tuple of both a state and a cost. The cost of an accepted string is equal to the som of the costs of all transitions. Another possible approach is to define the DFA as a 6-tuple with an separate cost function. Mathematically the two approaches are equivalent. For implementation related reasons the latter one is used in the actual code. Figure 2 shows an example of an cDFA.

In the paper Propagating Regular Counting Constraints [1] cDFA means *counting DFA*. It is defined in a similar manner, and counting can be interpreted as cost.
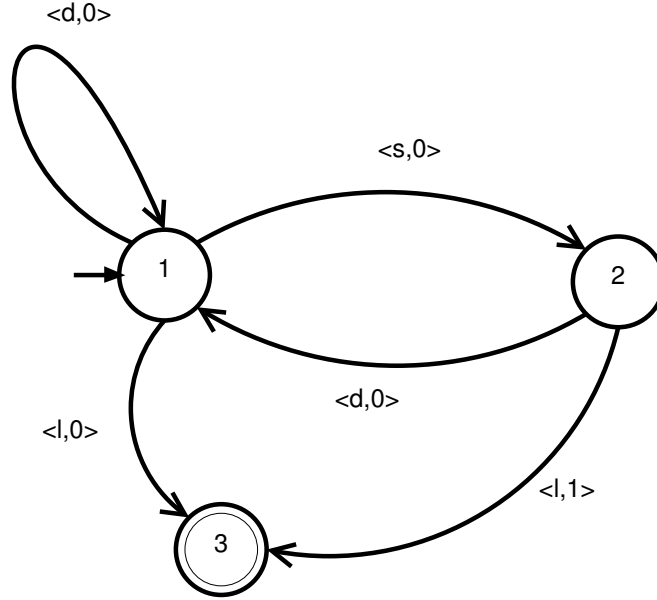

Figure 2: A cDFA for the Swedish drinking protocol. It is an altered variant of the DFA shown in Figure 1. The symbols d and s still mean drinking and singing, but the symbol l is introduced for leaving. In addition, costs are introduced for all transition, so you can leave without drinking after singing, but with some added cost.

2.4 The cDFA Constraint

A *constraint* is a rule that must hold for a solution. We have already looked at simple constraints, such as $x \leq y$, and such. Those constraints are easy to implement, and the propagators are easy to optimise. The constraint $cDFA$ is much more complicated. It is defined such that $cDFA(state_1, cost_1, state_0, cost_0, symbol, statefun, costfun)$ holds if $state_1 = statefun(state_0, symbol)$ and $cost_1 = cost_0 + costfun(state_0, symbol)$. Propagators for this constraint are discussed in Section 3.3.

2.5 Creating a Custom Variable in 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. For more information, visit the homepage: http://www.gecode.org

The following is a summary of Chapters P, B, and V in Modeling and Programming with Gecode [6].

2.5.1 Specification File

Creating the specification file is the first thing that needs to be done when designing a new variable. A simple specification file is described in Figure 3.

```
[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]
```

Figure 3: A minimal specification file.

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 a base class, from which the variable implementation class will inherit.

The [General] section is simple. 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 describe how the variables change. When a propagator wants to prune the domain of a variable, it uses the class methods that the variable implementation provides. It does not modify the domain directly. If no values are pruned, then the variable returns the modification event NONE, and if all values are pruned then it returns FAILED. When the domain of a variable gets pruned so that exactly one element remains, then 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 have changed. Note that other names than the required ones are arbitrary: DOM is usually denoting when any change has been made.

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

2.5.2 Variable Implementation

The variable implementation class would with the specification file in the previous section be called IntTupleVarImp and inherit from the class IntTupleVarBase generated from the specification file. It has to implement a function called `assigned` that returns true iff the variable is assigned. The implementation class is the class that does all the work. This is where the modification methods are defined, and the modification methods are the only methods that directly change the domain. The implementation class implements both the getters and setters that are used by the variable class and view classes.

2.5.3 Variable Class

The variable class is the class that is used when modeling a problem and inherits from the variable implementation. It is a read-only 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 by convention.

2.5.4 Deltas

Deltas are used for passing information about changes to improve efficiency. In this thesis an empty class is used. The only reason it is defined at all is that some mandatory virtual functions takes deltas as arguments.

2.5.5 Views

The variable implementation class implements 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.

2.5.6 Propagators

A propagator implements constraints and prunes variables. It interacts with the views and *subscribes* to variables. A propagator should subscribe to all variables it depends on. When a propagator has subscribed to a variable, it receives information whenever the domain of that variable changes in a way that is relevant. For instance, a propagator for the constraint $x \leq y$ should subscribe to the maximum value of x and minimum value of y, because if none of these changes, then there is nothing the propagator can do. Subscription is done using the view classes. A propagator class must have the following methods:

- post - The method for posting a constraint, which basically is declaring that a certain constraint should be used.

- dispose - This works as a destructor. The only reason Gecode does not use regular destructors is that destructors in C++ cannot take arguments.

- copy - A method to copy the propagator.

- cost - Estimates the cost to run the propagator. Making a good estimate 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 prunes domains and determines the status to return.

The `propagate` and `post` methods have the following possible return values:

- ES_FAILED - There is no combination of values in the domains that satisfies the constraint.

- ES_FIX - The propagator is at a fixpoint. It is impossible for the propagator to prune more values before a domain has changed for any of the variables the propagator has subscribed to. Further changes can be done 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.

- ES_SUBSUMED - The propagator is done. Regardless of any changes made to the variables, this particular propagator will not be able to do anything more.

The propagator must be able to return ES_FAILED and at least one of the others. ES_NOFIX is safe to return in the sense that it does not promise anything. If it cannot be guaranteed that the propagator has reached a fixpoint, then ES_NOFIX should be returned.

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) satisfy the constraint. Domain consistency means that all values in the whole domains are parts of at least one solution. Formally, domain consistency can be defined the following way: Given a constraint C, a value $a \in dom(x)$ for a variable $x \in vars(C)$ is said to have a *support* in C if there exists a tuple $t \in C$ such that $a = t[x]$ and $t[y] \in dom(y)$, for every $y \in vars(C)$. A constraint C is *domain consistent* if for each $x \in vars(C)$, each value $a \in dom(x)$ has a support in C [3]. The notation $t \in C$ means that t is an assignment to each of the variables in C, satisfying the constraint. The notation $t[x]$ denotes the value assigned to variable x by tuple t.

Bounds consistency can be defined in a similar way. First, we define $bounds(x)$. Second, we rephrase the last sentence in the definition of domain consistency to: A constraint C is *bounds consistent* if for each $x \in vars(C)$, each value $a \in bounds(x)$ has a support in C. For an integer variable x, $bounds(x)$ is defined to $\{min(dom(x)), max(dom(x))\}$. For tuple variables, we define $bounds(t)$ to be the set of all values $v \in dom(t)$ such that $\forall v : \nexists a, b \in dom(t) : v.x = a.x = b.x \land a.y \leq v.y \leq b.y$.

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 $x = 1$, $x = 7$, The values $y = 1$ and $y = 7$. $x = 5$ and $y = 3$ are not considered. If the same constraint were applied with domain consistency, then we would end up with $dom(x) = \{1, 7\}$ and $dom(y) = \{1, 7\}$. Pruning with domain consistency does in general cost more execution time, but results in a smaller search tree.

2.5.7 Branchers

When there are no propagators able to prune any variable, search is performed. The branchers determine the structure of the search tree. The simplest brancher just puts the first value in the domain - according to some order - 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. A brancher can create an arbitrary number of nodes, but the union of all nodes must be the original domain, and at least one node must be different from the others. Ideally, the intersection between two nodes created by a brancher is empty. A brancher is not strictly needed, but if they are not used, then the propagators need to be able to prune the domains to a solution.

2.6 Pair Variables

When solving a problem with CP, it is desirable to minimize the search tree to avoid brute force searching. The size of the search tree depends in part on the model and the strengths of 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, then the domains can be pruned to $dom(x) = \{3\}$ and $dom(y) = \{2\}$ in one step, and the solution is found without searching. If we instead have the same variables and domains but instead use the constraint $x \neq y$, nothing can be pruned, since there exist solutions for all values in both domains.

In this example, the problem was modeled with two integer variables. If the problem is modeled with one tuple variable instead of two integer variables, then pruning would be possible. In that case, we 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 domain from 9 to 7.

If we look at the case where $dom(x), dom(y) = \{1, \ldots, 1000\}$ and the constraint $x = y + 1$, then the benefits are more obvious. Using integer variables, only the values $x = 1$ and $y = 1000$ can be pruned. If tuple variables are used, then 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. A more realistic example is the constraint $cDFA$ described in Section 2.4. Furthermore, the constraint is linked in several steps: $cDFA(state_n, cost_n, state_{n-1}, cost_{n-1}, symbol_{n-1}, statefun, costfun)$. A solution to this problem would be three arrays: one for states, one for costs, and one for symbols. With tuple variables, this could be done more efficiently by combining states and costs into one variable: $cDFA(P, Q, symbol, statefun, costfun)$ where $P = \langle state_1, cost_1\rangle$ and $Q = \langle state_0, cost_0\rangle$.

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

For a pair variable p, the first dimension is denoted $p.x$ and the second is denoted $p.y$.

3 Implementing Pair Variables

To keep things simple, only two-dimensional integer tuples (Pairs) are considered. For a pair p, we use $p.x$ to represent the value in the first dimension, and $p.y$ for the second.

Two variants are implemented. One stores an exact representation of the domain, while the other stores an overapproximation. This means that the domain is bigger than it should be. The domain still fulfills the condition that it contains all values that it should, but also some values that it should not. The approximate version only stores the boundaries for the second dimension. It stores separate boundaries for each value in the first dimension. An approximation saves memory, but the biggest benefit is that some operations can be done in constant time. A more detailed description is found in Section 3.2. Consider the case with a tuple t and the constraint $t.x \neq 5$. The exact version has to remove an arbitrary amount of elements. The approximate version only needs to remove one element.
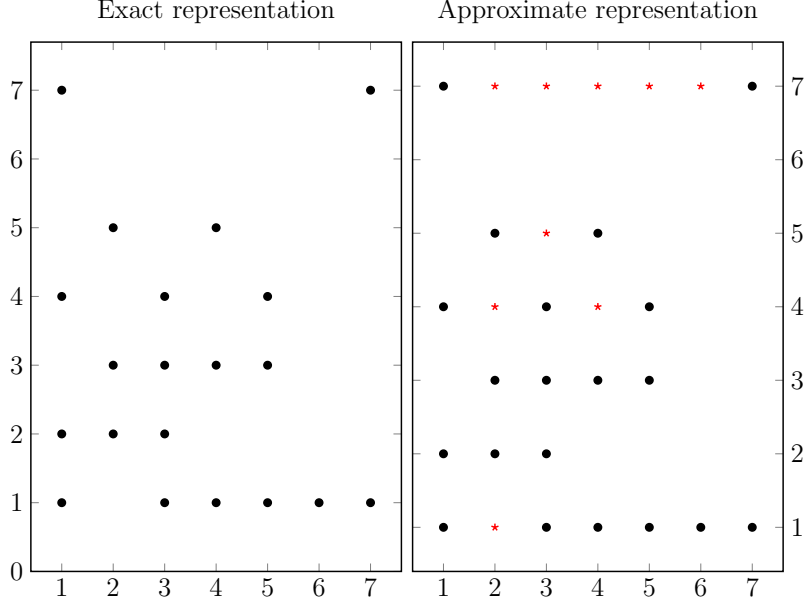
Figure 4: Comparison between an exact and an approximate representation of the same domain. The red stars are values that should not be there, but the approximate representation only stores the boundaries for the second dimension.

The same benefit also holds for the following example: First we define a constraint $YLTIXEQ$ (y less than if x equal) such that $YLTIXEQ(p, a, b)$ means that $p.x = a \implies p.y < b$. With the exact variant there are possibly several values that need to be removed. The approximate version only needs to change the value of the higher bound for the element with x-value b.

The drawback with the approximation is that it can not be pruned as efficiently as an exact representation. This is shown in Figure 4.

The specification files are kept almost as minimal as described in Section 2.5.1. Both implementations use identical specification files, except for the name of the variable.

Class names follows the convention in Gecode. The variable classes are therefore named IntPairExactVar and IntPairApproxVar.

There are three things added to the specification file described in Figure 3. The first two are the DOM modification event and the DOM propagating condition. These are used to schedule the propagators. As soon as the domain of any variable is changed, all propagators that have subscribed to that variable will be scheduled to run again. The last change to the specification file is the line `Dispose:true`. This is needed when using external memory resources outside Gecode. Our domains are stored as vectors from the C++ standard library, and therefore dispose is required. The way Gecode is designed, the destructors for variables are never called. If a variable needs a destructor it needs to be replaced by a method called `dispose`. This method explicitly calls `vector::~vector()`, which is the destructor for `vector`.

The main purpose of this thesis is to show that pair variables have benefits over regular integer variables for certain problems. For that reason the code for pair variables has purposely been kept unoptimised. For example, the methods implemented in the variable implementation do not use the fact that the domains are sorted. Just fixing this would decrease the time complexity from linear to logarithmic for operations like removing a value from a domain. This would be especially efficient when removing ranges of values. Currently, they are removed one by one.

The complete source can be found at `http://www.github.com/klutt/gecode-tuples`

```
[ General ]
Name:              IntPairApprox
Namespace:         MPG::IntPairApprox
Dispose:           true
[ ModEvent ]
Name:              FAILED=FAILED
[ ModEvent ]
Name:              NONE=NONE
[ ModEvent ]
Name:              VAL=ASSIGNED
Combine:           VAL=VAL
[ ModEvent ]
Name:              DOM=SUBSCRIBE
Combine:           VAL=VAL,  DOM=DOM
[ PropCond ]
Name:              NONE=NONE
[ PropCond ]
Name:              VAL=ASSIGNED
ScheduledBy:       VAL
[ PropCond ]
Name:              DOM
ScheduledBy:       VAL,  DOM
```

Figure 5: The specification file for IntPairApproxVar. The specification file for IntPairExactVar is equal, except for name and namespace.

3.1 Implementation of Pair with Exact Domain

The exact version does not approximate the domain, that is, the implementation is a long list of all integer pairs that are currently in the domain. The domain is stored as `std::vector<struct {int x, int y}>`.

Modification Methods

- **nq(Pair p) -** Remove the pair p from the domain. The time complexity is $\mathcal{O}(n)$ where n is number of elements in the domain.

Other methods

- **contains(Pair p) -** Returns true if the domain contains the pair p, and false otherwise. The time complexity is $\mathcal{O}(n)$ where n is number of values in the domain.

3.2 Implementation of Pair with Approximate Domain

This version of IntPairVar approximates the domain by just storing one dimension exactly. For the other dimension, only the boundaries are stored. Unique boundaries are stored for each value in the first dimension. The benefits are less use of memory and faster calculations for some operations, but the drawback is that the domain can not be pruned as efficient as the exact version, because no values whose y-value is between the bounds can be pruned. The domain is stored as `std::vector<struct{int x, int y_low, int y_high}>`

Modification Methods

- **nq(Pair p) -** Remove the pair p from the domain. The time complexity is $\mathcal{O}(n)$ where n is the number of different x-values in domain.

- **yeqforspecificx(PairInterval p) -** Remove all pairs from the domain whose y-value is not in the interval $[p.ylow; p.yhigh]$ iff those values x-values is equal to x. The time complexity is $\mathcal{O}(n)$ where n is the number of different x-values in domain. In the actual source code the name xeq is used, which may be misleading. The definition of this modification method was altered during development, but the name was never changed.

- **xeq(PairVector v) -** Remove all values from domain whose x-value does not exist in v. The time complexity is $\mathcal{O}(nm)$ where n is number of x-values in domain and m is number of elements in v.

Other methods

- **contains(Pair p) -** Returns true if the domain contains p, and false otherwise. $\mathcal{O}(n)$ where n is number of different x-values in the domain.

3.3 Implementation of Propagators for Constraint cDFA

The cDFA *propagators* are the actual implementations of the cDFA *constraint*. These propagators are implemented in three different ways. Two propagators are for pair variables and one is for integer variables. The latter one is only for reference when comparing performance. The algorithms for the propagators are described below in pseudo code.

All three implementations work in a similar manner. *PreState* and *PreCost* are domains containing current state and accumulated cost so far in the cDFA. *PostState* and *PostCost* are the result of reading the symbol in *Symbol*. Since pair variables make it possible to bundle cost and state into a single variable, they are just called *Pre* and *Post*. The basic principle is the following: For all symbols and all states in *Pre*, calculate the possible values for *Post*. Remove everything from *Post* that is not reachable from any element in *Pre*. If nothing is reachable from a certain *Pre*, then also remove that element from *Pre*. This is done by creating lists containing all elements that should remain in the domains, and then removing everything from the domains that is not in those lists.

Algorithm 1 DFA propagator IntVar

$\mathcal{O}(\#(PostState)\#(PostCost)\#(PreState)\#(PreCost)\#(Symbol))$

1: **procedure** DFAPROP(IntVar PostState, IntVar PostCost, IntVar PreState, IntVar PreCost, IntVar Symbol, Statefunction S, Costfunction C)
2:     Intdomain newPostState, newPreState, newPostCost, newPreCost, newSymbol := ∅
3:     **for all** symbol in Symbol **do**
4:         **for all** preState in PreState **do**
5:             int state := S(pre.state, symbol)
6:             **if** PostState.contain(state) and p.state ≠ garbage state **then**
7:                 newPostState := newPostState ∪ state
8:                 newPreState := newPreState ∪ pre
9:                 newSymbol := newSymbol ∪ symbol
10:                 **for all** cost in PreCost **do**
11:                     int postCost := cost + C(preState, symbol)
12:                     **if** PostCost.contains(postCost) **then**
13:                         newPreCost := newPreCost ∪ cost
14:                         newPostCost := newPostCost ∪ postCost
15:     PostState := PostState ∩ newPostState
16:     PreState := PreState ∩ newPreState
17:     PostCost := PostCost ∩ newPostCost
18:     PreCost := PreCost ∩ newPreCost
19:     Symbol := Symbol ∩ newSymbol

Algorithm 2 DFA propagator IntPairExact

$\mathcal{O}(\#(Post)\#(Pre)\#(Symbol))$

1: **procedure** DFAPROP(IntPairVar Post, IntPairVar Pre, IntVar Symbol, Statefunction S, Costfunction C)
2:     Pairdomain newPost, newPre := ∅
3:     Intdomain newSymbol := ∅
4:     **for all** symbol in Symbol **do**
5:         **for all** pre in Pre **do**
6:             Pair p := (S(pre.state, symbol), pre.cost+C(pre.state, symbol))
7:             **if** Post.contain(p) and p.state ≠ garbage state **then**
8:                 newPost := newPost ∪ p
9:                 newPre := newPre ∪ pre
10:                 newSymbol := newSymbol ∪ symbol
11:     Post := Post ∩ newPost
12:     Pre := Pre ∩ newPre
13:     Symbol := Symbol ∩ newSymbol

Algorithm 3 DFA propagator IntPairApprox
$\mathcal{O}(\#(Post)\#(Pre)\#(Symbol))$

 1: **procedure** MERGE(Pairdomain dom, Pair p)
 2:     find d in dom such that d.state = p.state
 3:     **if** not found **then**
 4:         add p to dom
 5:     **else**
 6:         d.high := max(d.high, p.high)
 7:         d.low := min(d.low, p.low)
 8: **procedure** DFAPROP(IntPairVar Post, IntPairVar Pre, IntVar Symbol, Statefunction S, Costfunction C)
 9:     Pairdomain newPost, newPre := $\emptyset$
10:     Intdomain newSymbol := $\emptyset$
11:     **for all** s in Symbol **do**
12:         **for all** pre in Pre **do**
13:             Pair p :=(S(pre.state,s),pre.low+C(pre.state,s),pre.high+ C(pre.state,s))
14:             **if** p.state $\neq$ garbage state **then**
15:                 Merge(newPost, p)
16:                 Merge(newPre, pre)
17:                 newSymbol := newSymbol $\cup$ s
18:     Post := Post $\cap$ newPost
19:     Pre := Pre $\cap$ newPre
20:     Symbol := Symbol $\cap$ newSymbol

The differences between the algorithms are adjustments to make the algorithms work with different variable types. For approximate integer variables we can not freely remove any value we want. That is why it has the function MERGE, which checks if there is an element d in the domain dom such that $d.x = p.x$. If so $d.y\_low$ and $d.y\_high$ get expanded to fit the interval $p.y\_low$ to $p.y\_high$.

The `contain` functions' time complexity is $\mathcal{O}(n)$ where n is the size of the domain for the Pair variables. It is fairly safe to assume that this is faster for the integer variables on avarage. The time complexity for the intersection operations is at most quadratic, but it does not matter since they are performed outside the loop. The union operation inside the loop is constant in time.

At a first glance it may seem that the IntPairExactVar propagator has better time complexity. This is an illusion, because the size of $\#(Pre)$ and $\#(Post)$ is equal to $\#(PreState)\#(PreCost)$ and $\#(PostState)\#(PostCost)$ respectively. However, this is not the case for the IntPairApproxVar propagator. Here $\#(Pre) = \#(PreState)$ and $\#(Post) = \#(PostState)$.

The propagator for IntPairExactVar achieves domain consistency. The propagator for IntPairApproxVar does not, but this would be impossible because the domain is only approximated.

4   Results

Some sentence just to fullfill requirements...

4.1 Correctness verification

The correctness is verified in several steps. For all problems an `assert` is used to ensure that the solution printed actually is a valid solution before it is printed. This ensures that no false solutions occur. There are a number of trivial tests in `testsrc/` with pre-calculated number of solutions. If the number of solutions are correct, all solutions are unique, and all solutions fullfill the requirements the tests are as good as they can get.

For bigger problems, it is very hard to pre-calculate the number of solutions. Instead the tests rely on probability. The two pair variants and the integer variant are compared to ensure that they put out the same number of solutions, and that the solutions are the same. It is unlikely that they would be wrong in the exact same way. These tests are found in `tripletestsrc/` and `multisteptestsrc/`.

4.2 Performance Test

To test the performance the same problem is solved in three different ways. Two of them are IntPairExactVar and IntPairApproxVar. The last one is solved with the IntVar that comes with Gecode, and is for reference. The problem is finding strings of a fixed length accepted by a cDFA. Branching is done on the symbols only, since this is enough. Indeed, if the cDFA is in a state s and has a current cost c and the symbol x is read, then the next state and cost can be uniquely calculated. The executables accept six arguments:

- seed - The seed for the random generator to generate the same DFA for all executables.

- states - Number of states.

- symbols - Size of the alphabet.

- cost - Maximum cost per transition in the cDFA. This is decreased by one, so setting this to 1 means no cost.

- max cost - Maximum cumulative cost for the string.

- steps - Length of the string.

The test is performed by varying one parameter of *states*, *symbols*, *cost* and *steps*, while keeping the others fixed. For each method and combination of parameters one hundred random cDFA are generated, except for when varying steps where it is only ten. All three methods generate the same cDFA for a given seed. Runtime and number of nodes are the total for the random generated DFA. The maximum total cost is set to $(steps \cdot cost)/5$.
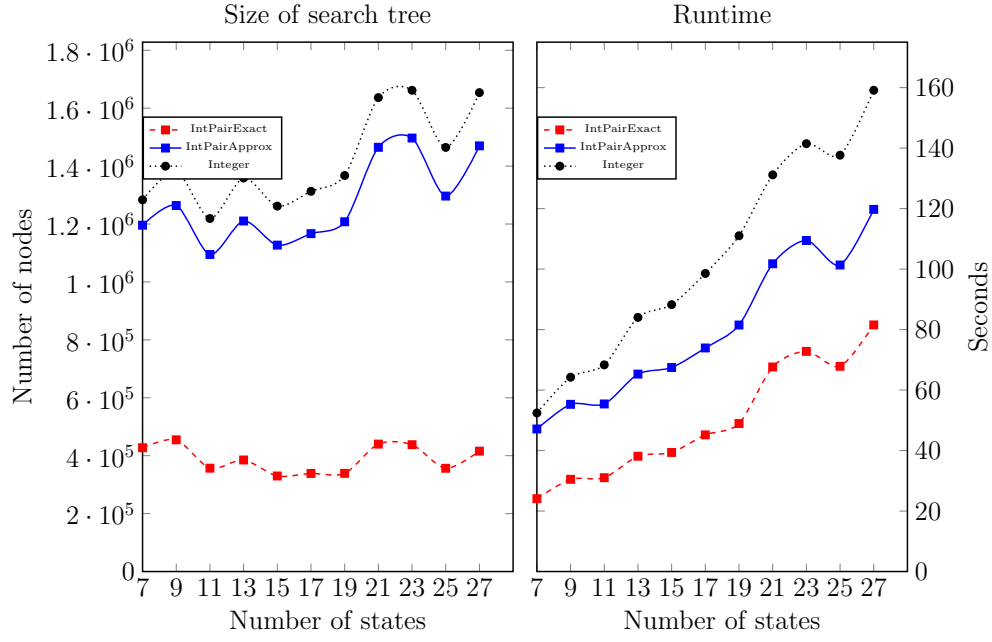
Figure 6: Varying the number of states. The other parameters are fixed. Size of alphabet=7, max cost per transition=15, and number of steps=7.
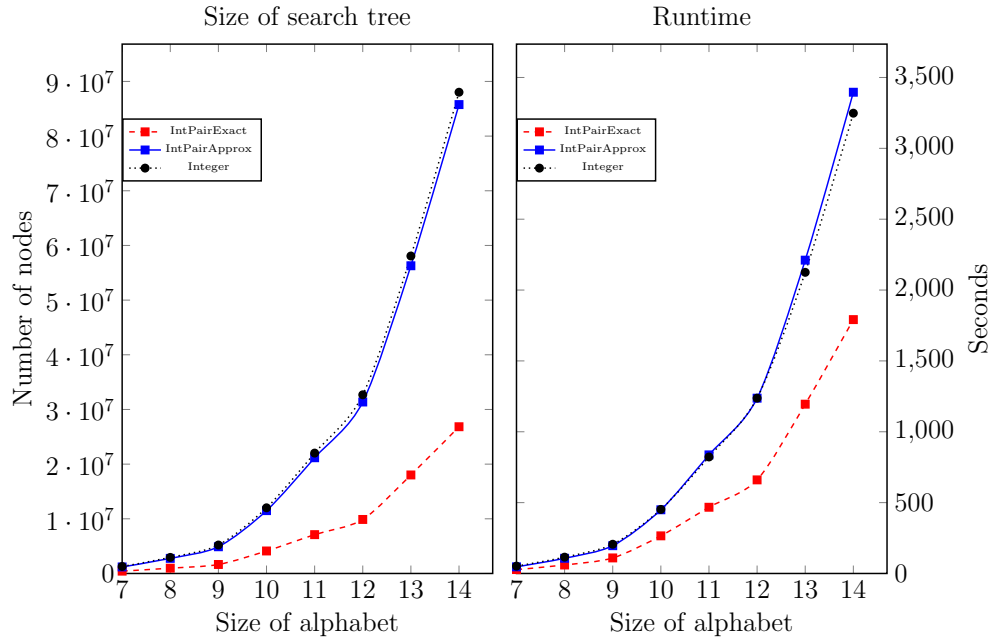

Figure 7: Varying the number of symbols. The other parameters are fixed. Number of states=7, max cost per transition=15, and number of steps=7.
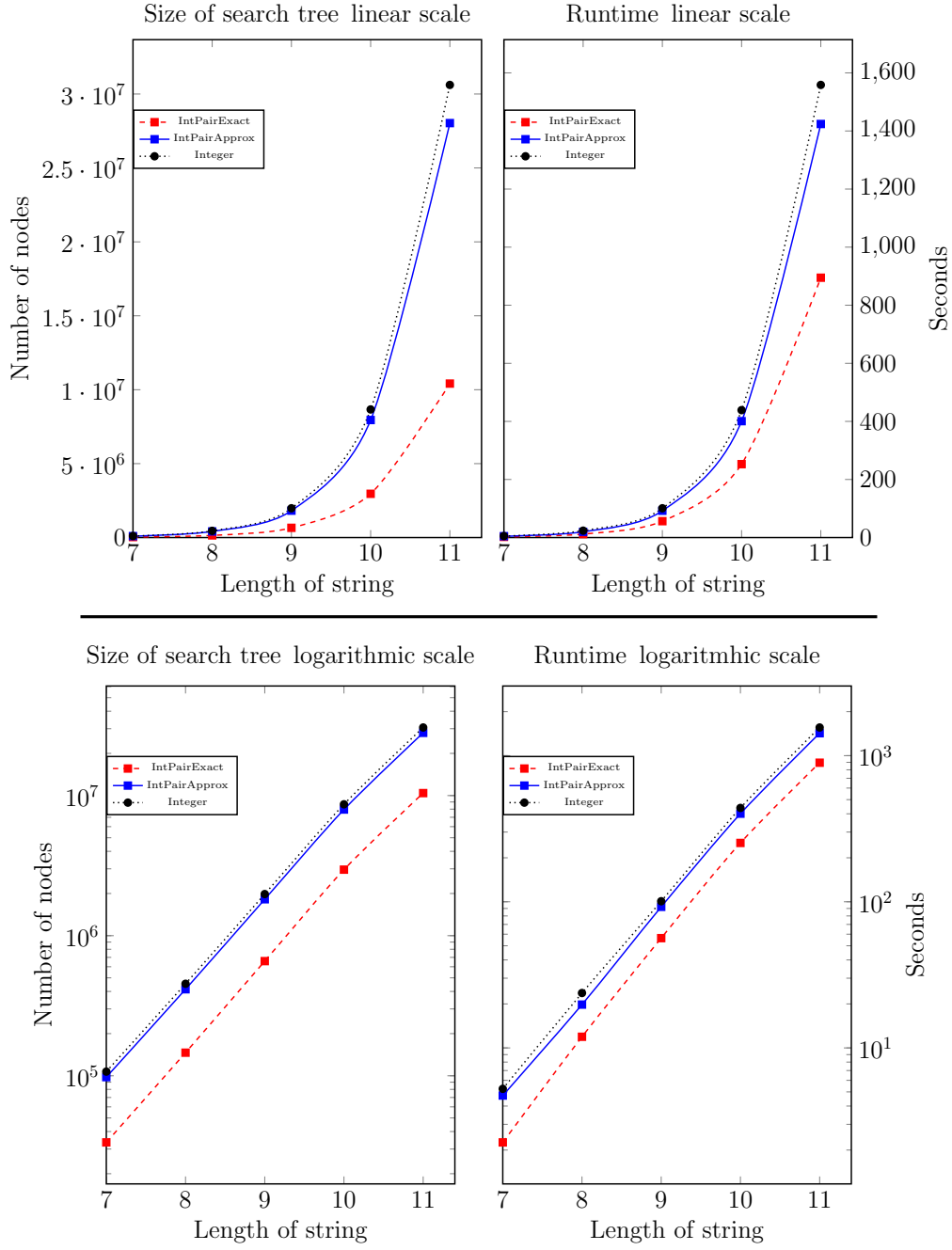
Figure 8: Varying the length of the string. The other parameters are fixed. Number of states=7, size of alphabet=7, and max cost per transition=15. Here we also show graphs in logarithmic scale.
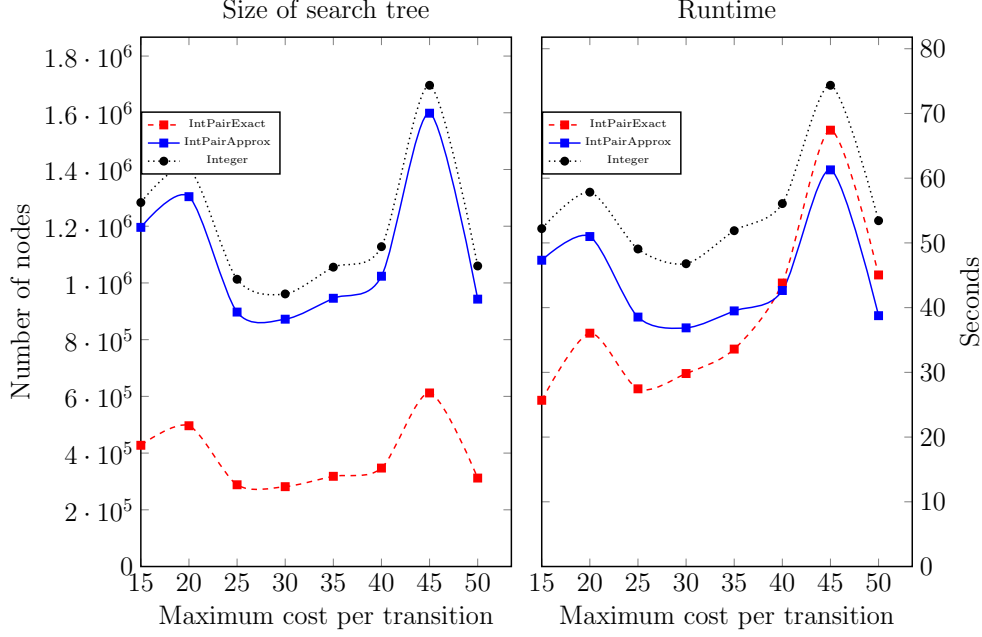
Figure 9: Varying the maximum cost per transition. The other parameters are fixed. Size of alphabet=7, size of alphabet=7, and number of steps=7. The lack of an obvious dependence pattern between the x and y axis is because the maximum total cost also varies with the cost per transition.

In the graphs we can see advantages for pair variables. In all tests they are more efficient than using regular integer variables in both execution time and size of the search tree. However, it does so at the cost of higher memory usage per node. A node in the integer variable variant uses five integer variables, while the pair variants use two pair variables and one integer variable. The size of a node for the IntVar variable case should be roughly proportional to $Steps \cdot (States + Maxtotalcost)$ plus some overhead, but for the exact pair case the domain sizes of the pairs is proportional to the number of states times the max total cost. This means that the size per node is proportional to $Steps \cdot States \cdot Maxtotalcost$. With approximate pairs, the size per node is proportional to $Steps \cdot States$. For two-dimensional tuples with relatively small domain sizes, this should not be of any concern. However, higher dimensions could cause troubles. The cDFA problem could be modeled with a five-dimensional tuple variable. For the case in Figure 6 with 50 states, 5 symbols, 6 steps and a maximum total cost of 18, the size per node would be around 100MB. Note though that the tuple variables have a higher memory usage *per node* and that this does not necessarily mean that the total memory usage is higher, since the number of nodes in the search tree shrinks. The case where the memory would be a problem is if the search tree became to deep, because a parent node can not be freed from memory until its children have been propagated to either a failure or a solution.

When tests were run (this is not shown in any graph) with the maximum total cost being equal to max cost per transition times the number of steps, the number of nodes is exactly equal for both exact pairs and approximate pairs. The reason is that if the starting state s_0 and the starting cumulative cost c_0 are assigned (which they are for a cDFA), then everything else follows automatically if we branch on the symbols first. This is easy to see because $s_1 = S(s_0, t_0)$ and $c_1 = c_0 + C(s_0, t_0)$. This is also confirmed by the fact that there were no failures during the execution. As can be seen in all graphs, this is not the case when restricting the maximum total cost to a lower value.

18

In Figure 8, the logarithmic graphs are close to straight lines. This indicates that all algorithms are exponential in both time and size of the search tree with respect to the length of the string. Since the lines are parallel, we can also see that they have the same exponential base. More importantly, this confirms that there was no complexity difference between pair variables and integer variables when solving this particular problem. Pair variables are faster, but not necessarily asymptotically faster.

Figure 6 and 9 fluctuate very much. The difficulty of the problems does not seem to depend very much on the size of the varied variable. When changing the number of states, the random generator generates completely different cDFAs. Changing the number of symbols also generate different cDFAs, but not in such a high degree. In the experiments, there are one hundred random cDFAs for each measure point. Increasing this, would probably smoothen out the curve. The reason why the graphs for varying the cost per transition fluctuates so much is different. The maximum total cost is set to the maximum cost per transition times number of steps divided by 5, so there are actually two parameters along the x-axis. One way to solve this would be to fix the maximum total cost to a fixed value. However, this is not a good solution, because a low value would make the problem easier when the cost per transition grows. On the other hand, a high value would make the problem to easy. This does not matter so much, since all the curves have the same shape, which means that if a cDFA was hard to solve, it was hard for all three methods, making the comparison fair. In Figure 9 we can also see that the runtime graphs for exact and approximate pairs crosses each other. This is expected, because the approximate version is more suitable for removing ranges of values.

All in all, the graphs indicate that pair variables can give a better performance for certain problems. The only possible source of error with this experiment that I can think of is that the reference propagator may not be optimised enough, but I find that highly unlikely. After all, the implementation of IntVar has to be assumed to be very optimised, and the reference propagator is much more optimised than the ones for pairs.

5 Related Work

Beldiceanu et al. [1] describe a propagator for a counter-DFA. It is defined as a regular DFA with only accepting states and such that the transition function not only returns the next state, but the increase for a counter. The cDFA used in this thesis are almost identical. The main difference is that we also use non-accepting states. The other difference is more philosophical. Instead of counting, we talk about cost, but it makes no difference in practice. Doing a performance comparison with their implementation would be complicated to do, because their propagators are implemented in Prolog, while everything in this thesis is implemented in C++.

Kellen Dye has implemented a bitvector variable for Gecode for his master thesis [2]. His report was useful for getting more understanding of how to implement a new variable in Gecode.

Monette et al.[5] show that tuple variables can be used to make pruning much more efficient. This thesis is a practical demonstration of their results, and that it translates into better performance.

6 Conclusions and future work

The experiments show that tuple variables have potential. Both versions perform better than the integer variable variant on a cDFA problem, in both number of nodes and total execution time, and there is still much room for further improvement. Especially the vector operations are very inefficient and can easily be improved. Using the exact pair variable improved the speed by 80-100 % in these experiments.

All large software libraries have coding standards. There is plenty of work to rewrite the code for pairs in such a way that it satisfies Gecode standards. For instance, vectors from the standard library are not allowed. Furthermore, the code is far from ready in other aspects. It contains only the methods needed to run the tests in this thesis. One example of this is that it exists methods to get the minimum and maximum value for the first dimension, but no methods for the second. Also, the propagators currently only accepts arrays.

In this thesis, there is one exact and one approximate version. The approximation is exact in one dimension and only stores the boundaries for the second. This is just one way of doing it and there are several other ways of approximating the domain. Another relevant change that could be made is to extend the pairs to arbitrary dimensions. A tuple of higher dimension would make it possible to shrink the search tree for a cDFA problem even further.

References

[1] Nicolas Beldiceanu, Pierre Flener, Justin Pearson, and Pascal Van Hentenryck. Propagating regular counting constraints. In *Proc. 28th AAAI Conference on Artificial Intelligence : Volume 4*, pages 2616–2622, 2014.

[2] Kellen Dye. Implementation of bit-vector variables in a cp solver, with an application to the generation of cryptographic s-boxes. Master's thesis, Uppsala University, Department of Information Technology, 2014. `http://uu.diva-portal.org/smash/get/diva2: 761927/FULLTEXT01.pdf`.

[3] T. Frühwirth, L. Michel, and C. Schulte. Constraints in procedural and concurrent languages. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 4.3. Elsevier, 2006.

[4] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley series in computer science. Addison-Wesley, 2001.

[5] Jean-Noël Monette, Pierre Flener, and Justin Pearson. A propagator design framework for constraints over sequences. In *Proc. 28th AAAI Conference on Artificial Intelligence : Volume 4*, pages 2710–2716, 2014.

[6] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with Gecode. `http://www.gecode.org/doc/4.3.3/MPG.pdf`.