

Implementing Tuple Variables in Gecode

Bachelor Thesis

Uppsala University

Patrik Broman

January 27, 2016

Abstract

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

Contents

1	Introduction	2
2	Background	2
2.1	Constraint Programming	2
2.2	Motivation for Tuple Variables	3
2.3	Gecode	4
3	The IntPair Variables	6
3.1	Implementation of IntPairExact	7
3.2	Implementation of IntPairApprox	7
3.3	Implementation of DFA Propagators	7
3.4	Correctness verification	10
3.5	Performance Test	10
4	Related Work	15
5	Conclusions and future work	16

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 Introduction

Almost all constraint programming (CP) problems could be pruned more efficiently if it was possible to remove a combination of values from the domains, rather than removing the values individually. In many of these problems a more efficient pruning would mean faster execution. In the paper *A propagator design framework for constraints over sequences* (Monette, Flener, Pearson) a tuple variable is described. In this thesis it is shown that a tuple variable can do that, and that it is possible to reach better performance with it. A *tuple* is a mathematical object. It is an ordered list of elements. In this thesis a *tuple variable* is a *tuple* in Gecode, which in turn is a library for CP. Currently, it supports boolean, integer, integer set and float variables. The tuple variable described here is an extension to the Gecode library.

To demonstrate this I have implemented two different variants of a tuple variable. Both of them are restricted to 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 optimise the code that handles the tuple variables. Even obvious and simple optimisations have been avoided. For the same reason I have optimised the reference propagator as far as I can, and I also asked more experienced CP-programmers for help with that.

To compare performance I used the tuple variables to find all valid paths through a deterministic finite automaton (DFA) with costs for each transition. It means that the DFA does not only tell if a string is accepted or not, but also that the total cost is valid. The performance is measured by studying how the execution time and size of the search tree depends on four different parameters: number of states, size of alphabet, maximum cost per transition and number of steps. Number of steps can be interpreted as the length of the string.

The results confirm that tuple variables can be used for increasing performance. The tuple variable which has an approximated domain performs slightly better than the exact version. The parameter that gives the greatest advantage to tuple variables over regular integer variables is the number of steps.

2 Background

2.1 Constraint Programming

Constraint programming 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 9×9 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, 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 queens¹. 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

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

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 variable 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.

2.2 Motivation 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 $dom(y) = \{2\}$ in one step, and this is the solution to the problem. If we instead have the same variables and domains but instead use 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 integer variables 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 $x = y + 1$ the benefits are more obvious. Using integer variables only the values $x = 1$ and $y = 1000$ 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. A more realistic example is the constraint $DFA(state_1, cost_1, state_0, cost_0, symbol, statefun, costfun)$. It is a constraint for a DFA with cost and a solution has to satisfy $state_1 = statefun(state_0, symbol)$ and $cost_1 = cost_0 + costfun(state_0, symbol)$. This is the constraint used in this thesis. Furthermore, the constraint is linked in several steps: $DFA(state_n, cost_n, state_{n-1}, cost_{n-1}, symbol_{n-1}, statefun, costfun)$. A solution to this problem would be three vectors, 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: $DFA(P, Q, symbol, statefun, costfun)$ 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 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 is a part of a solution.

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

Everything that follows in this chapter is basically a summary of MPG (Modeling and Programming with Gecode). Especially chapter P, B and V.

When implementing a new variable in Gecode there are several things that should be written. Among them is specification file, various classes and propagators.

2.3.1 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 class, from which the variable implementation class will inherit.

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. Note that other names than the required ones are arbitrary.

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.

2.3.2 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 if the variable is assigned. The implementation class is the class that does all the work. This is where the modification events are defined, and the modification events are the only methods that directly changes the domain. The implementation class implements both the getters and setters that are used by the variable class and view class.

2.3.3 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.

2.3.4 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.

2.3.5 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.

2.3.6 Propagators

A propagator implements constraints and prunes variables. They interact with the views and have four possible return values. A propagator class must have the following methods:

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

The propagate method 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 depends 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.
- 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 can't be guaranteed that the propagator has reached a fixpoint, this should be returned. Furthermore, the propagator needs the following methods:

A propagator must also subscribe to views, that is, telling Gecode when the propagator should be awoken. 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 there is nothing the propagator can do.

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. If the same constraint were applied with domain consistency we would end up with $dom(x) = \{1, 7\}$ and $dom(y) = \{1, 7\}$. Pruning the whole domain does in general cost more execution time.

2.3.7 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 is not strictly needed to implement a variable, but if there is no brancher the propagators must be able to prune the variable until it is assigned.

3 The IntPair Variables

To keep things simple only two-dimensional integer tuples are considered, so called Pairs. The specification files are kept almost as minimal as described in the introduction. None of the propagators utilizes the boundaries anyway. Both implementations uses identical specification files except for the name of the variable.

There are only two things added to the specification file described in section 2.3.1. The first is the DOM modification event and propagator condition. This is used to schedule the propagators. As soon the domain of any variable is changed, all propagators that variable depend on will be scheduled. The second change to the specification file is the line `Dispose: true`. This is needed when using external memory resources outside Gecode, and the domains are stored as vectors from the standard library. 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()`.

The main purpose of this thesis is to show that tuple variables have benefits over regular integer variables for certain problems. For that reason the code for Pairs has purposely been kept unoptimised. The modification events 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 values from the domains. 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>

3.1 Implementation of IntPairExact

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. The domain is stored as `std::vector<struct {int x, int y}>`.

Modification Events

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

3.2 Implementation of IntPairApprox

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 less use of memory and possibly faster calculations for some problems, but the drawback is that it can't be pruned as efficient as the exact version because it can only achieve bounds consistency. The domain is stored as `std::vector<struct {int x, int y_low, int y_high}>`

Modification Events

- **nq(Pair p)** - Remove p from domain. $\mathcal{O}(n)$ where n is the number of different x-values in domain.
- **xeq(PairInterval p)** - Remove all values from domain whose y-value is not in the interval $[p.y_{low}; p.y_{high}]$ iff those values x-values is equal to x. $\mathcal{O}(n)$ where n is the number of different x-values in domain. The name xeq is very badly chosen. The definition of this modification event was altered during development, but the name was never changed.
- **xeq(vector<Pair> v)** - Remove all values from domain whose x-value does not exist in v. $\mathcal{O}(nm)$ where n is number of x-values in domain and m is number of elements in v.

3.3 Implementation of DFA Propagators

The DFA *propagators* are the actual implementations of the DFA *constraints*. These propagators are implemented in three different ways. Two are for IntPair and one is for integer variables.

The latter one is just for reference when comparing performance. The algorithms are described below in pseudo code.

All three implementations work in a similar manner. *PreState* and *PreCost* is current state and accumulated cost so far in the DFA. *PostState* and *PostCost* is the result of reading the symbol in *Symbol*. Since the Pairs makes it possible to bundle cost and state to a single variable they are just called *Pre* and *Post*. The basic principle is: for all symbols and all states in *Pre*, calculate the possible values for *Post*. Remove everything from *Post* that is not reachable. If nothing is reachable from a certain *Pre*, also remove that. This is done by creating lists of everything that should remain and then removing everything 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 Pre-
   Cost, IntVar Symbol, Statefunction S, Costfunction C)
2:   Intdomain newPostState, newPreState, newPostCost, newPreCost, newSymbol :=  $\emptyset$ 
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) then
7:         newPostState := newPostState  $\cup$  state
8:         newPreState := newPreState  $\cup$  pre
9:         newSymbol := newSymbol  $\cup$  symbol
10:      for all cost in PreCost do
11:        int postCost := cost + C(preState, symbol)
12:        if PostCost.contains(postCost) then
13:          newPreCost := newPreCost  $\cup$  cost
14:          newPostCost := newPostCost  $\cup$  postCost
15:   PostState := PostState  $\cap$  newPostState
16:   PreState := PreState  $\cap$  newPreState
17:   PostCost := PostCost  $\cap$  newPostCost
18:   PreCost := PreCost  $\cap$  newPreCost
19:   Symbol := Symbol  $\cap$  newSymbol

```

Figure 1: int

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 :=  $\emptyset$ 
3:   Intdomain newSymbol :=  $\emptyset$ 
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.contains(p) then
8:         newPost := newPost  $\cup$  p
9:         newPre := newPre  $\cup$  pre
10:        newSymbol := newSymbol  $\cup$  symbol
11:   Post := Post  $\cap$  newPost
12:   Pre := Pre  $\cap$  newPre
13:   Symbol := Symbol  $\cap$  newSymbol
```

Figure 2: exact

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
```

Figure 3: approx

The differences are just adjustments to make the algorithms work with different variable types. For the IntVarApprox 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 $p.y_{high}$.

At a first glance it may seem that the Pair propagators have better time complexities. This is an illusion, because the size of $\#(Pre)$ and $\#(Post)$ is equal to $\#(PreState)\#(PreCost)$ and $\#(PostState)\#(PostCost)$.

The propagators for IntPairExact and IntVar both achieves domain consistency. The propagator for IntPairApprox does not, but this would be impossible because the domain is only approximated.

3.4 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 solution before the solution 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 and all solutions fulfill the requirements the algorithms work correctly.

For bigger problems it is very hard to pre-calculate the number of solutions. Instead the tests relies on probability. The two IntPairs and one integer implementation are compared to ensure that they put out the same number of solutions, and that the solutions are the same. It is very unlikely that they would be wrong in the exactly same way. These tests are found in `tripletestsrc/` and `multistep-testsrc/`

3.5 Performance Test

To test the performance the same problem is solved in three different ways. Two of them is IntPairExact and IntPairApprox. The last one is solved with the integer variable that comes with Gecode, and is just for reference. The purpose of this thesis is to show that tuple variables can be very useful. For that reason the reference propagator have been optimized, while the Pairs have been left with a very naive implementation. 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.
- symbols - 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.

The test is performed by varying one parameter of *states*, *symbols*, *cost* and *steps*. The others are kept constant while keeping the others fixed. For each method and combination of parameters ten random DFA:s are generated. All three methods generate the same DFA:s for a given seed. Runtime and number of nodes are the total for the ten different DFA:s. For all tests four graphs in total are used. Two are for the number of nodes in the search tree and two are for the runtime. For each of these there is one linear and one logarithmic graph. The maximum total cost is set to $(steps \times cost)/5$.

Varying states

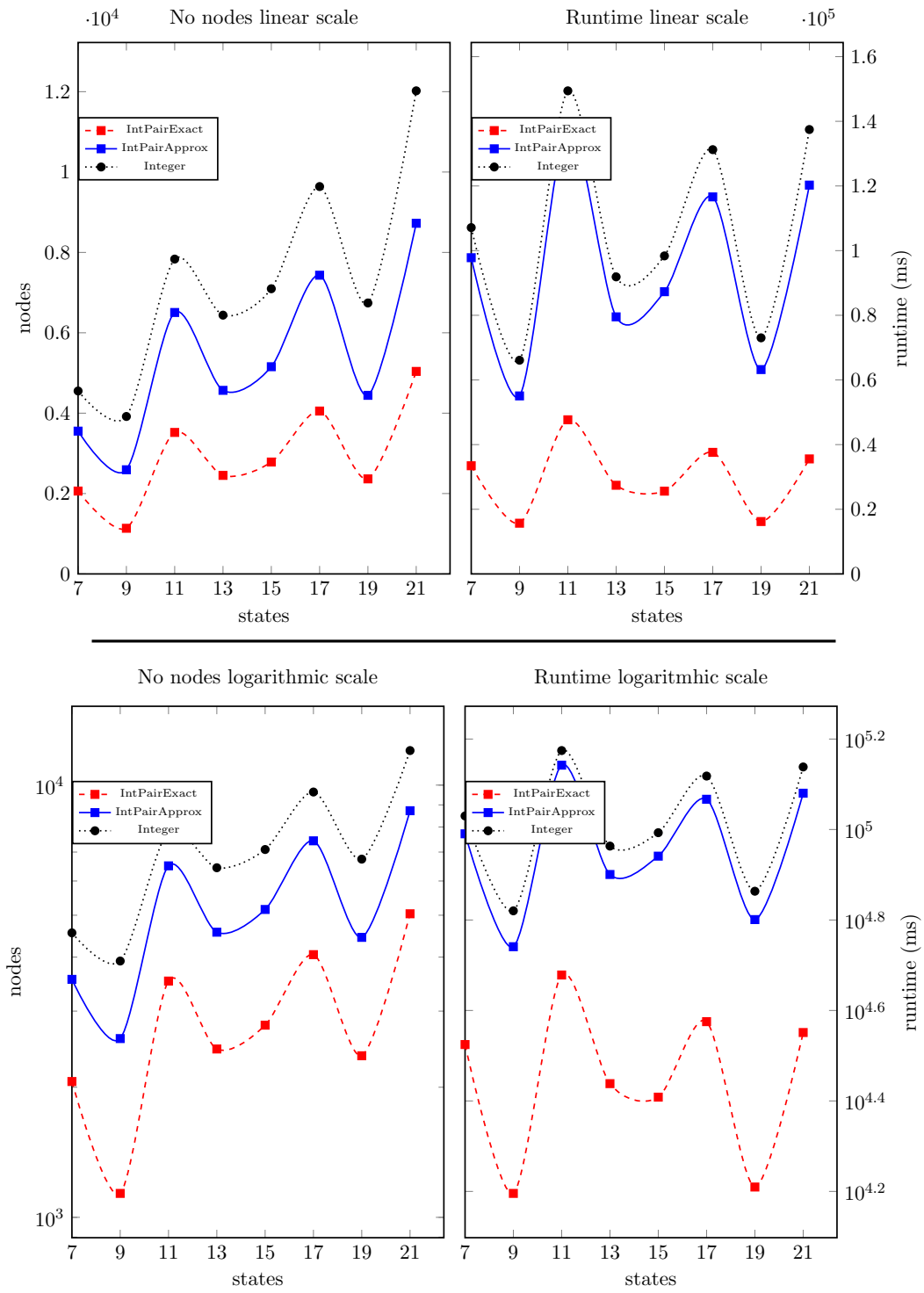


Figure 4: Caption

Varying symbols

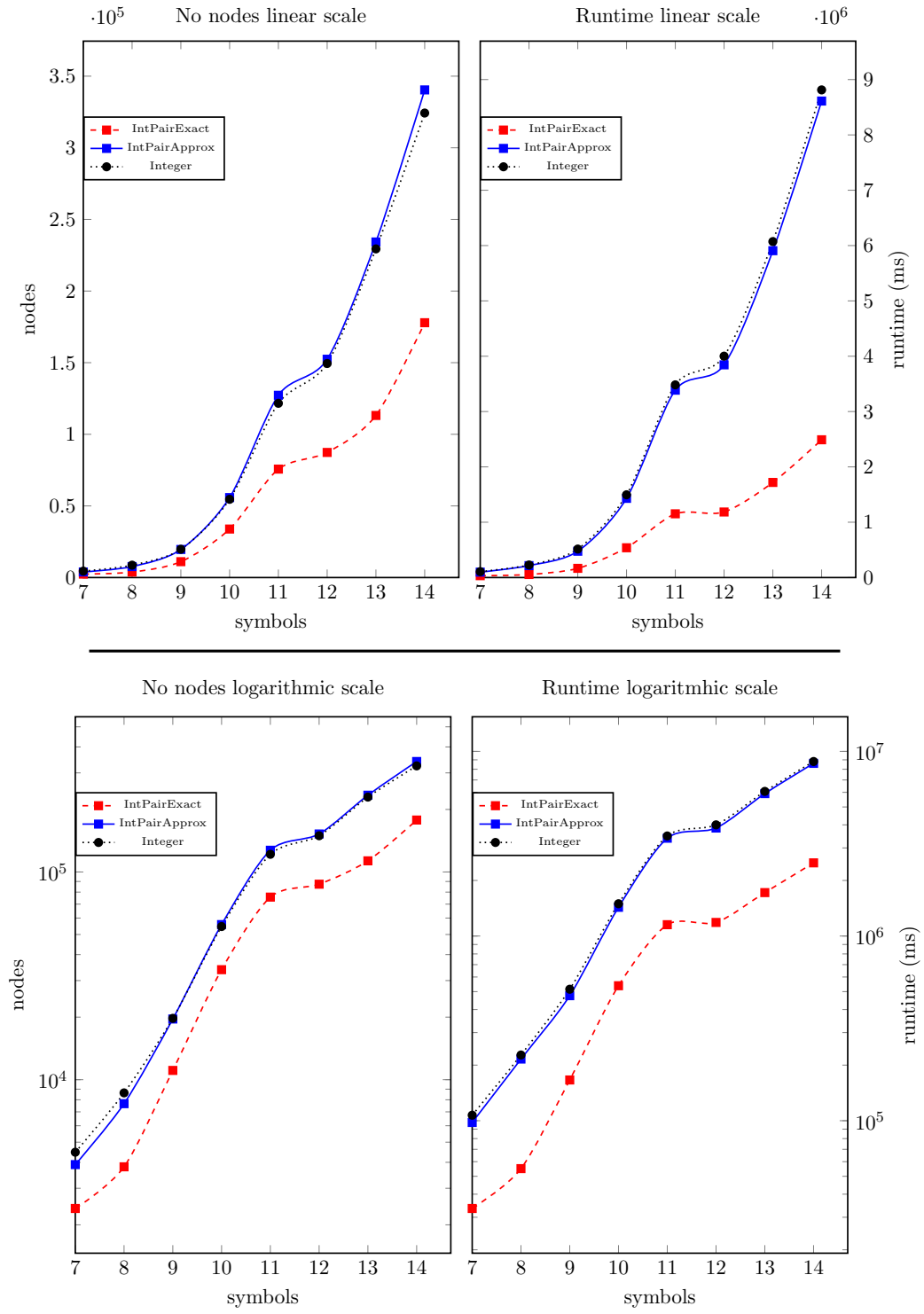


Figure 5: Caption

Varying cost

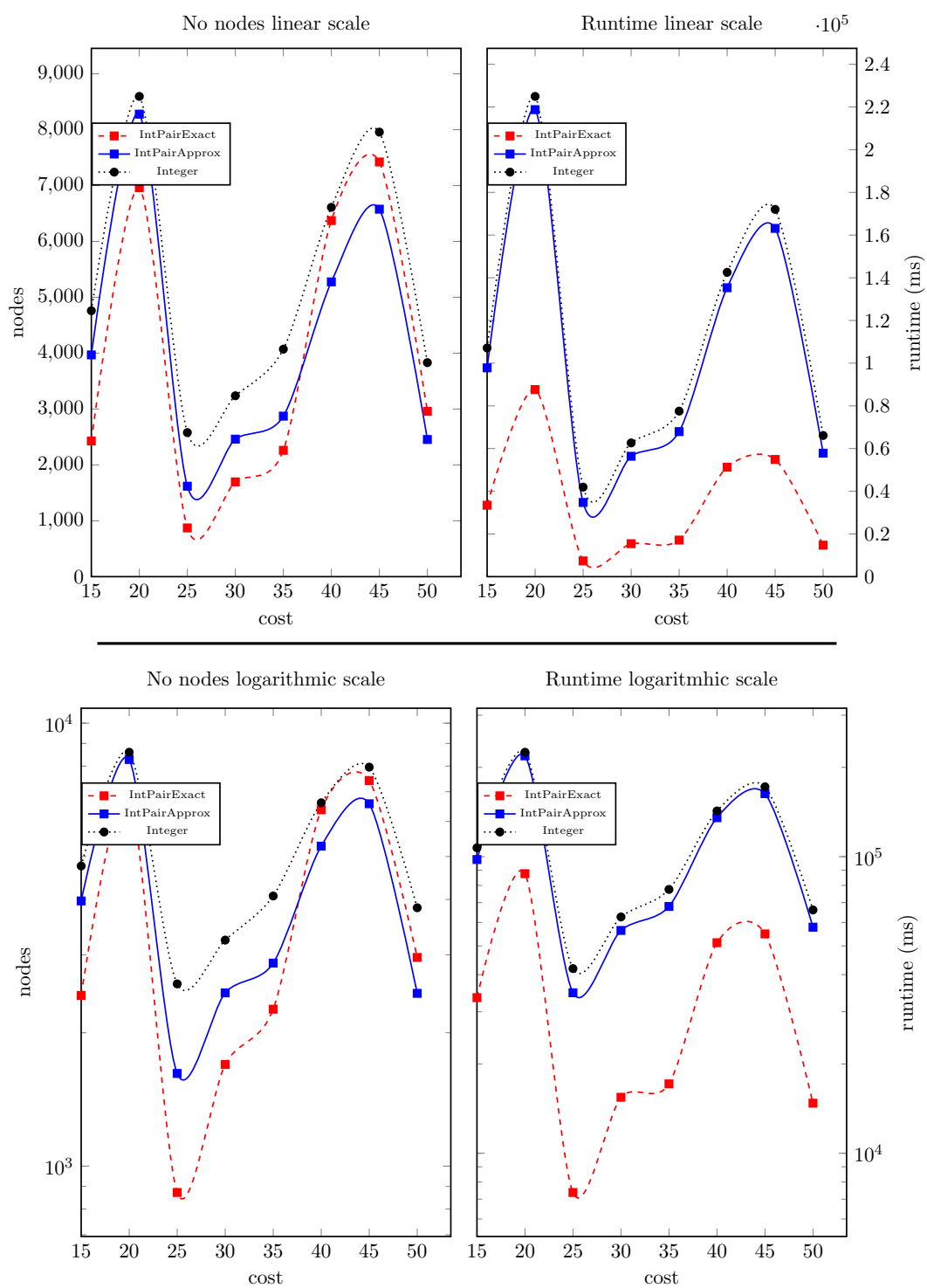


Figure 6: Caption

Varying

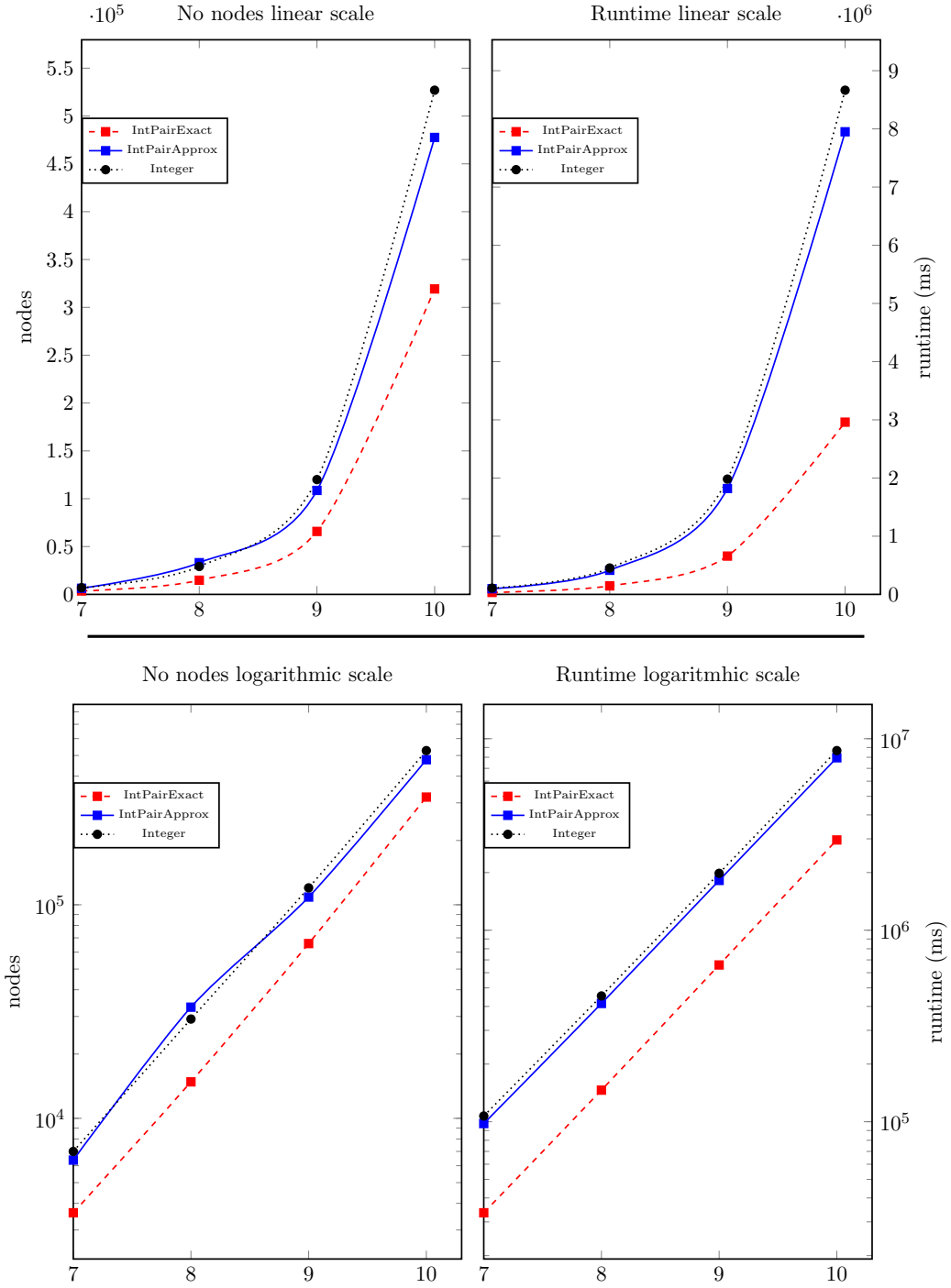


Figure 7: Caption

In the graphs we can see advantages for IntPair. In all tests they win over the regular integer variable solution in both execution time and the 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 variant uses two pair variables and one integer variable. The size of a node for the integer variable case should be roughly proportional to $Steps(States + Maxtotalcost)$ plus some overhead, but for the 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 $Steps \times States \times Maxtotalcost$. For two-dimensional tuples with "normal" domain sizes this should not be of any concern. However, higher dimensions could cause trouble. The DFA problem could be modeled with a five-dimensional tuple variable. For the case in figure ?? with 50 states, 5 symbols, 6 steps and max total cost 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 shrinks. The case where the memory would be a problem is if the search tree became too deep, because a parent node can not be freed from memory until its children have been propagated to either a failure or a solution.

For all situations the number of nodes are exactly equal for both IntPairExact and IntPairApprox. This may be a bit surprising. It would be reasonable that the number of nodes were, if ever so slightly, higher for IntPairApprox, since the domain is approximated. The reason is that if the starting state s_0 and the starting cumulative cost c_0 is assigned (which they are for a DFA), 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. If we removed some values from the domains of t_1 , s_1 and c_1 and above things would have been different.

In all figures, except figure ?? the logarithmic graphs are close to parallel straight lines. This indicates that all algorithms are exponential in both time and size of search tree. More importantly, this confirms that there was no complexity difference between tuple variables and integer variables when solving this problem. Tuple variables are faster, but not asymptotically faster.

In figure ?? IntPairApprox seems to be almost constant in time. Even though this may seem very good it is not something that shows that tuple variables work, since the same result probably would be achievable by using an approximate version of integer variables for the cost. In the same graph we can see that the number of nodes were constant for all three methods. This is because how the DFA:s are generated. The random generator first generates the state matrix and then the cost matrix, so the automaton looks exactly the same except for the transition costs.

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

4 Related Work

The paper Propagating Regular Counting Constraints (Beldiceanu, Flener, Pearson, Van Hentenryck) describes constraints for a counter-DFA (cDFA). 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 DFA in this thesis is 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 practise. I wanted to do a performance comparison with that implementation, but it would be complicated to do that in a fair way, because the constraints are implemented in Prolog, while everything in this thesis is in C++.

In Implementation of bit-vector variables in a CP solver (Dye) Gecode is used, and a new variable is implemented. That paper was useful for getting more understanding of how to

implement a new variable in Gecode.

The paper A propagator design framework for constraints over sequences (Monette, Flener, Pearson) shows that tuple variables can be used to make pruning much more efficient. This thesis is basically a practical demonstration that this indeed is true, and that it translates directly into better performance.

5 Conclusions and future work

The tests shows that tuple variables have potential. Both of them perform better than the integer variable variant 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.

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 standards. For instance, vector from std is 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, that is, proving that tuple variables actually is a good idea. One example of this is that it exists methods to get the minimum and maximum x-value for the domain, but no methods for y-value. Also, the propagators and the brancher currently only accepts arrays. The brancher for IntPairApprox has a bug, which manifested itself by making the program producing the same solution several times which should not be possible. This does not matter in this thesis since it is not used.

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.

References

- [1] Beldiceanu, Flener, Pearson, Van Hentenryck, *Propagating Regular Counting Constraints*,
- [2] Dye, *Implementation of bit-vector variables in a CP solver*,
- [3] Monette, Flener, Pearson, *A propagator design framework for constraints over sequences*,
- [4] Schulte, Tack, Lagerkvist *Modeling and Programming with Gecode*,