Implementing Tuple Variables in Gecode
Bachelor Thesis
Uppsala University

Patrik Broman

December 16, 2015

Abstract

Constraint programming is a programing paradigm focusing on high performance computing. There exists several different frameworks for constraint programming. Gecode is one of them, and is the one chosen for this project. In the present version of Gecode the available variable types are booleans, integers, floats and integer sets. In this thesis a new variable type - tuples - is implemented and analyzed. It is shown that for some problems, this variable type can improve performance significantly.

Contents

Acknowledgements

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

1 Introduction

Intro...

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. 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, 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[1]. 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 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 (shrinked) 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 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.

---

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

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

2.3   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 wit h two integer variables. An 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 $x = y + 1$ 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 $DFA(state_1, cost_1, state_0, cost_0, token, statefun, costfun)$. It is a constraint for a DFA with cost and a solution has to satisfy $state_1 = statefun(state_0, token), cost_1 = cost_0 + costfun(state+0, token)$. This is the constraint used in this thesis. Furthermore, the constraint is linked in several steps: $DFA(state_n, cost_n, state_{n-1}, cost_{n-1}, token_{n-1}, statefun, costfun)$

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 is a part of a solution.

2.4 Creating a New Variable in Gecode

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.4.4 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 virtual functions takes Deltas as arguments.

2.4.5 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.

2.4.6 Propagators

A propagator implements constraints and prunes variables. They interact with the views and have four possible return values.

- ES_FAILED - There are no 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 the 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 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.

- 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.

A propagator must also subscribe to views, that is, telling Gecode when the propagator should be awaken. For instance, a propagator for the constraint $x \leq y$ 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.

2.4.7   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   The IntPair Variables

To keep things simple only two-dimensional integer tuples are considered, so called Pairs. The specification file is kept almost as minimal as described in the introduction. None of the propagators utilizes the boundaries anyway, and there is only one propagator working for each execution in the performance test. The only thing added 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 should explicitly call `vector::~vector()`.

The main purpose of this thesis is to show that tuples have benefits over regular integers for certain problems. For that reason the code for Pairs has purposely been kept unoptimised. The modification events do not use the fact that the domains are sorted. Just fixing this would decrease their time complexity from linear to logarithmic.

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. $\mathcal{O}(x)$ where x is the number of different x-values in domain.

- **xeq(Pair p) -** Remove all values from domain whose y-value is not in the interval $[p.y\_low; p.y\_high]$ iff that values x-value is equal to p.x. $\mathcal{O}(x)$ where x is the number of different x-values in domain.

- **xeq(vector<Pair>v) -** Remove all values from domain whose x-value does not exist in v. $\mathcal{O}(xm)$ where x is number of x-values in domain and m is number of elements in v.

3.3 Implementation of DFA Propagator

This propagator is implemented in three different ways. Two are for IntPair and one is for integer variables as reference. The algorithms are described in pseudo code:

Algorithm 1 DFA propagator IntPairExact

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

1: **procedure** DFAPROP(IntPairVar Post, IntPairVar Pre, IntVar Token, Statefunction S, Costfunction C)
2:     Pairdomain newPost := $\emptyset$
3:     Pairdomain newPre := $\emptyset$
4:     Intdomain newToken := $\emptyset$
5:     **for all** token in Token **do**
6:         **for all** pre in Pre **do**
7:             Pair p := (S(pre.state, token), pre.cost+C(pre.cost, token))
8:             **if** Post.contain(p) **then**
9:                 newPost := newPost $\cup$ p
10:                 newPre := newPre $\cup$ pre
11:                 newToken := newToken $\cup$ token
12:     Post := Post $\cap$ newPost
13:     Pre := Pre $\cap$ newPre
14:     Token := Token $\cap$ newToken

Algorithm 2 DFA propagator IntPairApprox
$\mathcal{O}(\#(Post)\#(Pre)\#(Token))$

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 Token, Statefunction S, Costfunction C)
9:     Pairdomain newPost := ∅
10:     Pairdomain newPre := ∅
11:     Intdomain newToken := ∅
12:     **for all** token in Token **do**
13:         **for all** pre in Pre **do**
14:             Pair p := (S(pre.state, token), pre.low+C(pre.low, token), pre.high+C(pre.high, token))
15:             **if** p.state ≠ garbage **then**
16:                 Merge(newPost, p)
17:                 Merge(newPre, pre)
18:                 newToken := newToken ∪ token
19:     Post := Post ∩ newPost
20:     Pre := Pre ∩ newPre
21:     Token := Token ∩ newToken

Algorithm 3 DFA propagator IntVar
$\mathcal{O}(\#(PostState)\#(PostCost)\#(PreState)\#(PreCost)\#(Token))$

1: **procedure** DFAPROP(IntVar PostState, IntVar PostCost, IntVar PreState, IntVar Pre-Cost, IntVar Token, Statefunction S, Costfunction C)
2:     Intdomain newPostState := $\emptyset$
3:     Intdomain newPreState := $\emptyset$
4:     Intdomain newPostCost := $\emptyset$
5:     Intdomain newPreCost := $\emptyset$
6:     Intdomain newToken := $\emptyset$
7:     **for all** token in Token **do**
8:         **for all** preState in PreState **do**
9:             int state := S(pre.state, token)
10:            **if** PostState.contain(state) **then**
11:                newPostState := newPostState $\cup$ state
12:                newPreState := newPreState $\cup$ pre
13:                newToken := newToken $\cup$ token
14:                **for all** cost in PreCost **do**
15:                    int postCost := cost + C(preState, token)
16:                    **if** PostCost.contains(postCost) **then**
17:                        newPreCost := newPreCost $\cup$ cost
18:                        newPostCost := newPostCost $\cup$ postCost
19:     PostState := PostState $\cap$ newPostState
20:     PreState := PreState $\cap$ newPreState
21:     PostCost := PostCost $\cap$ newPostCost
22:     PreCost := PreCost $\cap$ newPreCost
23:     Token := Token $\cap$ newToken

All three implementations work in a similar manner. PreCost and PreState is current state in the DFA. PostCost and PostState is the result of reading the token in Token. 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 tokens and all states in Pre, calculate the possible 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.

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 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 $\#(PostState)\#(PostCost)$.

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 fullfill the requirements the algorithms works 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 the same solutions. It is very unlikely that they all are wrong in the exactly same way. These tests are found in `tripletestsrc/` and `multisteptestsrc/`

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 tuples 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.

- 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.

In all examples, the maximum total cost is set to max total cost per transition times number of steps.
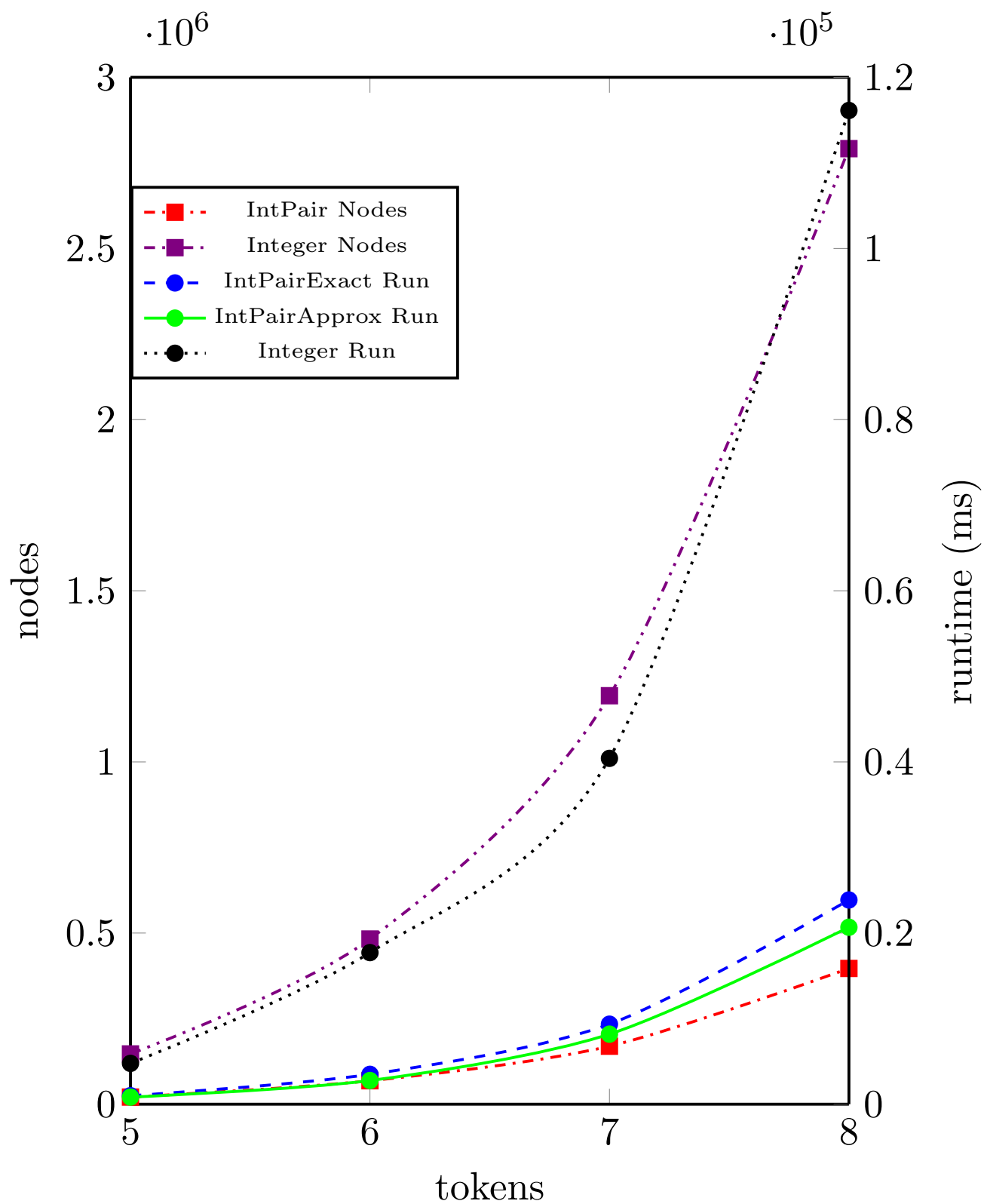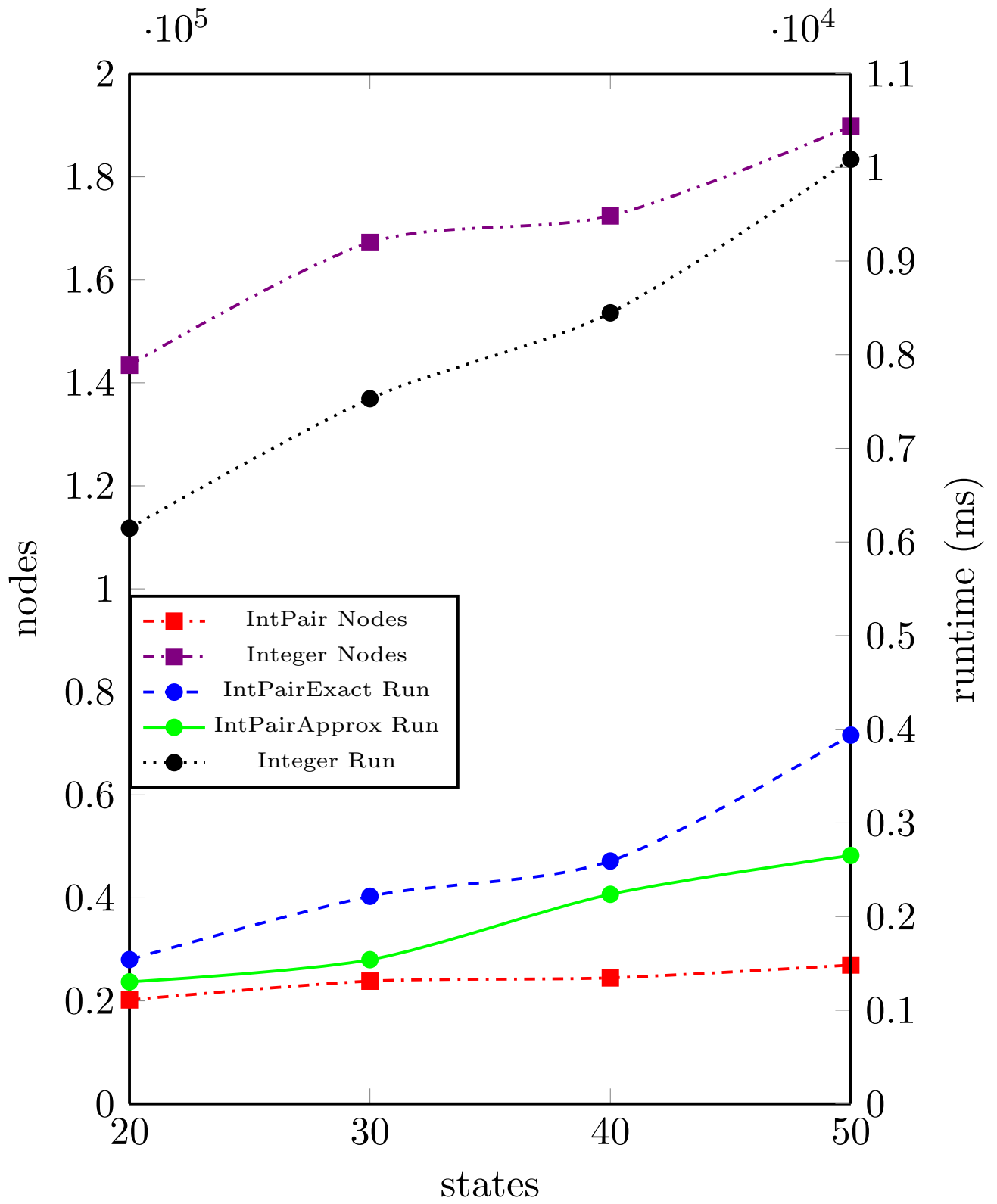
Figure 1: States=15, Maxcost=3, Steps=6
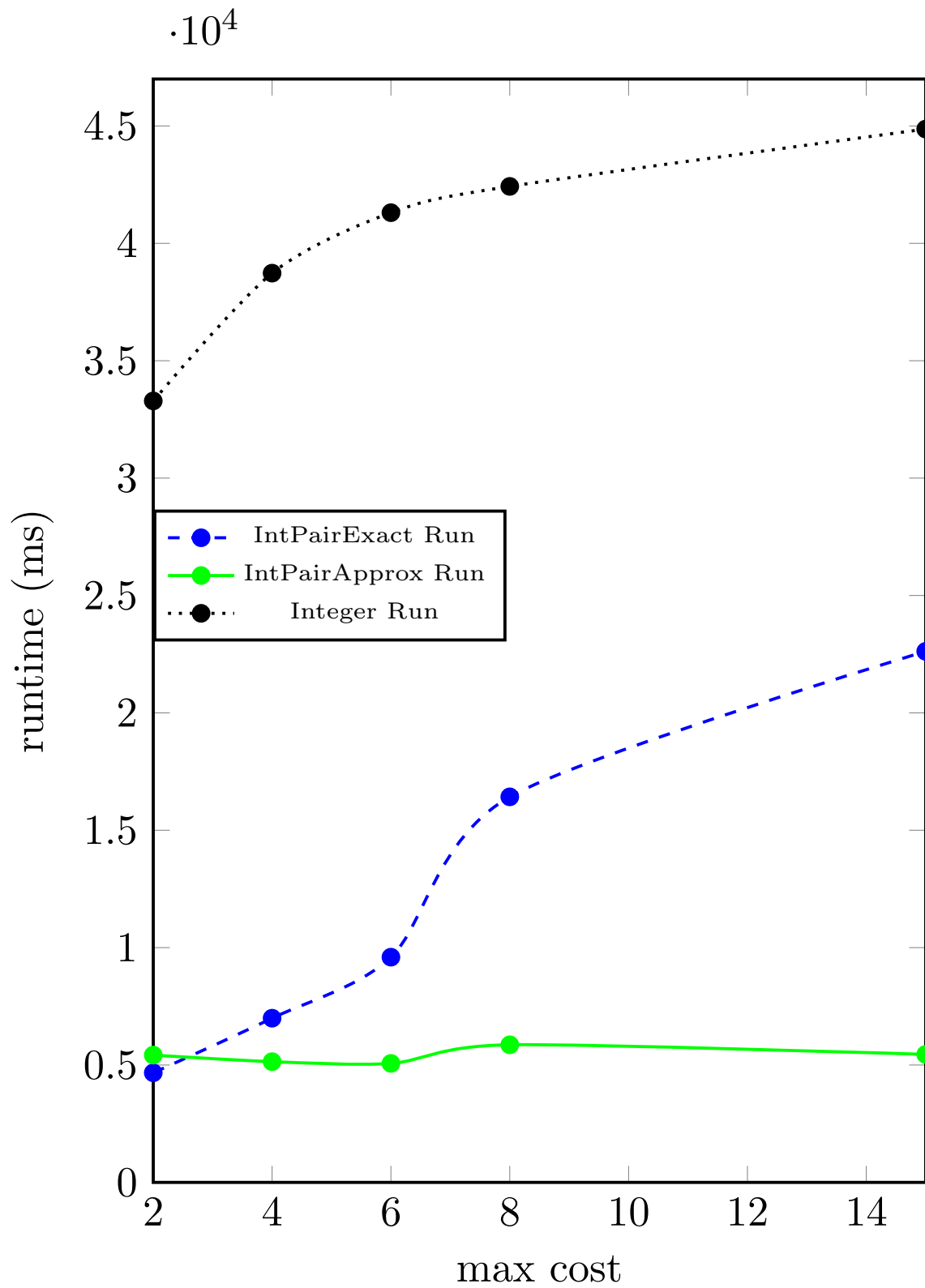
Figure 2: Token=5, Maxcost=3, Steps=6

Figure 3: Steps=20, Tokens=5, Steps=7. Number of nodes is omitted because it is constant. The number of nodes for Pairs are 93000 and for Integer 760000.
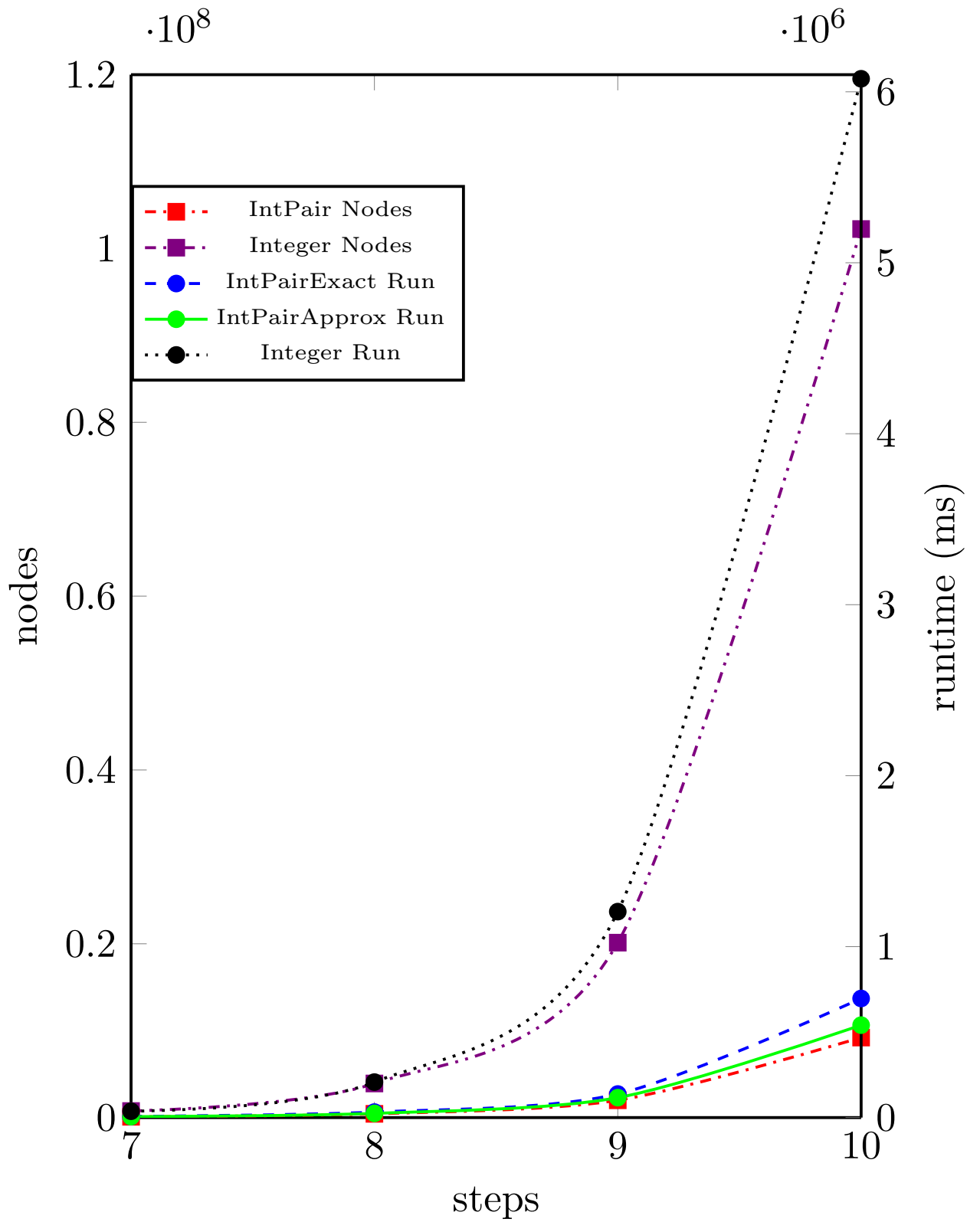
Figure 4: States=20, Tokens=5, Maxcost=4

In the graphs we can see advantages for IntPair. In all tests they win over the regular integer 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 variant uses five integer variables, while the pair variant uses two pairs and one integer. The size of a node for the integer 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 $Steps*States*Maxtotalcost$. 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. For the case in figure 2 with 50 states, 5 tokens, 6 steps and max total cost 18 the size per node would be around 100MB. Note though that the tuples have a higher memory usage per node and that this does not necessarily mean that the total memory usage is higher because the number of nodes shrinks.

For all situations the number of nodes are exactly equal for both IntPairExact and IntPairApprox. This is 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.

In figure 3 IntPairApprox seems to be constant in time. Even though this may seem very good it is not something that shows that tuples work, since the same result probably would be achie

4 Related Work

- **Implementation of bit-vector variables in a CP solver**

- **A propagator design framework for constraints over sequences**

5 Conclusions and future work

The tests shows that tuples show potential. Both of them perform better than the integer variant in both number of nodes and total execution time. The only flaw with this experiment is that the integer solution may not be optimized enough. There is still some room for further improvement, like implementing back propagation, but the integer variant only has one unoptimized part, which is the propagator. The IntPair variants have two. Both the propagator and the variable implementation is far from optimized.

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 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 tuples 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 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.

5.2 Test different implementations with different domain representations

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.

5.3   Extend the tuples to arbitrary dimensions and not just pairs

This would require a complete rewrite of the variable implementation.

References

[1] Calculus, Adams

[2] Operating systems and its concepts, Galvin