Seminar on Constraint Programming

April 2005

Daniel Schreck
Supervisor: Guido Tack
Seminar Constraint Programming
Winter semester 2004/2005
Programming System Lab
Department of Computer Science
Universität des Saarlandes

Table of Contents

Propagator Scheduling

Daniel Schreck

Universität des Saarlandes, Saarbrücken, Germany
s9daschr@stud.uni-saarland.de

Abstract. This paper is about efficiently implementing propagation, which is the key component of a solver for constraint satisfaction problems (CSPs). We first show how propagation algorithm and constraint solver could be implemented and then enhance this proposed model in order to save propagation time. This is on one hand achieved by reasoning and bookkeeping about the propagators that are at a fixpoint. On the other hand we show how to run propagators prioritized, in an order that quickly leads to solution or failure. The content of this paper is based on one by Schulte and Stuckey [SS04b] and covers the same topics, but tries to present them in a manner more suitable for readers with little previous knowledge.

1 Introduction

Constraint satisfaction problems (CSPs) can be formulated from many different application domains. According to Apt [Apt03] these include optimization problems in operations research, business applications, the construction of efficient parsers for natural language processing, the solving of equations in computer algebra and many more often hard combinatorial problems. *Constraint solvers* can solve CSPs of all these domains, although for the hard and complex problems it may take hours or days or even longer. Thus it is important to build constraint solvers that are fast, thereby increasing the number of problems that can be solved in reasonable time and enabling simple problems to be solved instantly. Note that there are techniques to speed up solvers for specific problems, but they are not topic of this paper.

In this first section we introduce the basic concepts of constraint programming. We will define necessary terms including the notion of a propagator for a constraint. In the following section we present a model of a constraint solver and especially of the propagation algorithm that is responsible for the execution of the propagators. The third section "Fixpoint reasoning" deals with avoiding unnecessary calls to propagators, thereby speeding up propagation. After that we will see how changes in the order of execution of the propagators can be beneficial. The fifth and last section will provide a summary of the paper.

A CSP consists of a set of variables V and constraints on those variables. Each variable also has a domain, which we here assume to be a finite set. It is the set of possible values the variable can have. If there is a solution to the CSP each variable can only have a value from it's domain. In the examples given in this paper the domains of all variables will be integer sets. We define D, the *domain of a CSP*, to be the mapping that assigns to each variable x a domain $D(x)$.

A simple example for a CSP: Two variables x and y with domains $D(x) = \{1, 2\}$ and $D(y) = \{2, 3\}$ and a constraint "$x < y$". This means that in a solution of the CSP $x \in \{1, 2\}$ and $y \in \{2, 3\}$. By looking at this very simple problem we find the (only possible) solution $x = 2$ and $y = 3$. Actually when we are "looking", we can either reason about what is possible, or just try out every combination of values for x and y. A constraint solver is capable of both. It uses reasoning and inference during *propagation* phases and *search* whenever propagation alone gets stuck. The propagation is the difference to a brute-force-approach. In general having strong yet effective propagation, that leaves as little work as possible to be done by search, is the key ingredient to any fast constraint solver. Propagation is realised by running propagators for each of the declared constraints.

We say that a *propagator* implements a constraint and denote the set of propagators for a constraint c by $prop(c)$. More precisely propagators are functions that try to modify the domain of the CSP by infering that certain variable assignments are impossible and removing those values from the respective variables domain. In that way an old domain is replaced by a new, narrower domain. This process is sometimes called a *narrowing conversion*, as in [SS04a].

More formally: We define a domain D_1 to be *narrower* or *stronger* than another domain D_2 ($D_1 \sqsubseteq D_2$) iff $\forall x \in V : D_1(x) \subseteq D_2(x)$ and require of every propagator to satisfy the following statement: $\forall D : f(D) \sqsubseteq D$. So a propagator either leaves the domain unchanged or it removes some possible values. If the propagator removes values from a certain domain D, we say that it *contributes*. If the propagator cannot contribute in D ($f(D) = D$), we say that it is at a *fixpoint* in D.

We now give a simple example for a propagator. Assume we are given the constraint $c =$"$x < y$". A propagator for c could be $f(D)$, defined as

$D(x) := \{u \in D(x) | u < max\{D(y)\}\}$

$D(y) := \{w \in D(y) | w > min\{D(x)\}\}$

If we apply this propagator to an inital domain D with $D(x) = D(y) = \{1, 2, 3\}$, it will remove the "3" from $D(x)$ and the "1" from $D(y)$. The former is done because the currently know maximum value for y is 3 and so all values that are possible for x must be smaller than 3. The latter is just the inverse case. We could say that the first line imposes c onto x and the second line imposes c onto y. To clear things up, we can also achieve the same effect with two propagators. Then $prop(c) = \{g(D), h(D)\}$ with

$g(D)$ defined as $D(x) := \{u \in D(x) | u < max\{D(y)\}\}$ and

$h(D)$ defined as $D(y) := \{w \in D(y) | w > min\{D(x)\}\}$

We will discuss the pros and cons of the aprroach with two propagators later.

Although we pass the whole domain to every propagator for the reason of easy interchangeability, we say that the *input variables* of a propagator are only those on which it's behaviour depends. In this case both g and h depend only on x and y and not on any other variables that may exist in the CSP's domain.

When the domain of variable contains only a single possible value ($\mid D(x) \mid = 1$), we say that the variable is *determined* (or *fixed*). A CSP is *solved* as soon as all variables are determined. It is *failed* when a propagator infers that there is no possible value for at least one variable of the CSP. Goal of propagation is to arrive at a domain D which is a *common* fixpoint of all propagators ($\forall f : f(D) = D$). Of course a failed CSP's domain is a fixpoint, too.

For most problems propagation alone does not suffice to find a solution. If no propagator can contribute any more, the principle of trial and error is used in a search. Several CSPs are derived from the original one, each is augmented with a new constraint. The new constraints must result in a partition of the original CSP. Therefore this step is called *splitting*. The constraint solver then tries to recursively solve each of the new CSPs, possibly splitting each of them again et cetera. When one of the new CSPs is solved, a solution of the original one has been found. When all of the new CSPs fail, the original CSP can not have a solution, because the newly introduced constraints formed a partition thus not ommiting possible solution paths.

A simple search is illustrated in figure 1. Propagation happend in the nodes of the tree. Diamonds indicate a failure. Boxes indicate a solution. The circles inidicate propagation phases that neither reduced in failure nor a solution, but instead in splitting of the CSP.

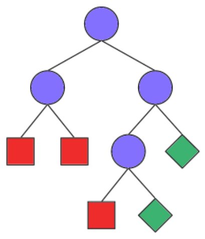

Fig. 1. A search tree (Taken from [SS04a])

2 Constraint Solver

Knowing which propagators are at a fixpoint and which are not is essential to achieve high levels of efficiency. Thus the constraint solver and constraint propagation algorithm we are going to present now subdivide the propagators into two disjoint sets. The set $F_{fixpoint}$ contains the propagators that are known to be at a fixpoint in the current domain. The other set F_{new} contains those that are expected to make new contributions when run. First the pseudo code for the main function, into which search and propagation are embedded. It will initially be called with "`search(`$\emptyset$`,` F`, D)`" where F is the set of all propagators of the CSP and D is the initial domain.

```
@invariant: all propagators in $F_{fixpoint}$ have a fixpoint in $D$
search($F_{fixpoint}$, $F_{new}$, $D$) { // $F_{fixpoint} \cap F_{new} = \emptyset$
   $D$ := isolv($F_{fixpoint}$, $F_{new}$, $D$); // propagation
   if ($\exists x \in V : D(x) = \emptyset$) return false;
   if (all variables in $D$ are determined) return true;
   choose new (‘‘splitting’’-) constraints $\{c_1, \ldots, c_m\}$
   forall($i \in \{1..m\}$) {
      if (search($F_{fixpoint} \cup F_{new}$ , $prop(c_i)$, $D$)) return true;
   }
   return false;
}
```

The code is pretty self-explanatory. The propagation happens inside the `isolv` function. Search begins when propagation does not find a solution. Then the original CSP is split up into several ones as described above and each of those is searched for a solution recursively.

The `search` function returns a boolean value that indicates whether there exists a solution for the CSP given by the propagators and the intial domain (value is `true`) or not (value is `false`). In case a solution was found, the current domain contains the variable assignment. The mechanism for the retrieval of the current domain by the caller was omitted from this pseudo code.

Note that splitting is realized by posting special constraints and not by modifying the domain directly. The new CSPs differ from the original only in the additional propagator(s) for the newly introduced constraints. This way all changes on the domain happen in the propagation function `isolv`, where the propagators are executed. So it is easier to keep track of the propagators at fixpoint, at least from the presentation point of view. The pseudo code of the propagation function `isolv` is below.

The propagation algorithm works it's way through an agenda of possibly contributing propagators stored in the set Q. Initially Q contains the propagators from F_{new}. In each iteration of the main loop, a propagator f is chosen from Q by means of the -yet unspecified- function `choose`. f is removed from Q and applied to the domain D. Although not stated in the pseudo code, the propagator flags if it made the CSP fail by removing all values from a variables' domain and in this case `isolv` returns immediately. Otherwise, to maintain the loop invariant that Q contains all possibly contributing propagators, we call the method `new`.

`new` checks which of the propagators that where at a fixpoint in D are no longer at a fixpoint in the new domain resulting from the application of f to D. This is a crucial task, because if we miss any propagators that are not at a fixpoint any more, the propagation algorithm may miss possible chances for inference. The overall correctness of the constraint solver will be maintained, but

```
@invariant: again $D$ is fixpoint of all propagators in $F_{fixpoint}$
isolv($F_{fixpoint}$, $F_{new}$, $D$) { // $F_{fixpoint} \cap F_{new} = \emptyset$
   $F$ := $F_{fixpoint} \cup F_{new}$;
   $Q := F_{new}$;
   while ($Q \neq \emptyset$) {
      $f$ := choose($Q$); // selects a propagator to apply
      $Q$ := $Q \setminus \{f\}$
      $D'$ := $f(D)$;
      // find out which propagators are no longer at a fixpoint
      // and add them to $Q$
      $Q$ := $Q \cup$ new($f$, $F$, $D$, $D'$);
      $D$ := $D'$;
   }
   return $D$;
}
```

propagation is weaker than possible. Thus the runtime will increase with high probability because of the need to search more. If, on the other hand, we add a lot of propagators that are still at a fixpoint, then all these propagators will be run although they do not contribute. Thereby we will loose time as well, and in the worst case the algorithm will not terminate.

We can ensure termination by checking if a propagator that has just been executed did change the domain. If not, we do not need to add any propagator to Q. Together with the fact that propagators perform narrowing conversions and thus always return stronger domains, we can prove termination. Proof sketch: If a propagator f removes values from the domain, the domain gets stronger; if it does not, we know that f is at a fixpoint and do not run it again until another propagator has removed a value from the domain. In that way we never run a propagator twice on the same domain. As the propagators are sort of monotoncially decreasing functions, the current domain will be strengthened until either the CSP fails or a common fixpoint of all propagators is reached.

Obviously propagators whose input variables have not been changed by the latest propagator's run can not have changed their fixpoint state. Thus the following implementation of `new` will ensure termination of the propagation algorithm:

$$new_{input}(f, F, D, D') = \{e \in F | \exists x \in input(e) : D(x) \neq D'(x)\}$$

Read: we need to add all propagators to Q where the domain has changed on at least one of the input variables.

On the following pages we will explore how to speed up the basic constraint propagation algorithm just presented. We will discuss possibilities for improving new_{input} by doing some fixpoint reasoning. After that we will investigate in what way the choice for the next propagator should be made by the function `choose`. The effectiveness of the presented techniques will be illustrated by results from the evaluation of Schulte and Stuckey.

3 Fixpoint reasoning

Improving new_{input} to requires a closer look at the properties of the propagators.

3.1 Idempotence

One important issue is idempotence, the abillity of a propagator to make an immediate re-execution of itself obsolete. We define a propagator f to be *idempotent* iff $\forall D f(f(D)) = f(D)$. Thus for any domain f arrives at a fixpoint after one execution. This is also called *static* idempotence. If we know the propagator just executed is idempotent, we do not have to add it to Q immediately after it's execution, even if it has changed it's input variables' domain. So we improve `new`:

$$new_{sidem}(f, F, D, D') = new_{input}(f, F, D, D') \setminus \{f | f \ idempotent\}$$

This should have quite some impact, because many of the common propagators are idempotent and these would often be re-inserted into the queue when using just new_{input}. For example g, one of our propagators for "$x < y$", would, after we applied it to $D(x) = D(y) = \{1, 2, 3\}$, be re-added to the queue. This is because it changed $D(x)$ to $D'(x) = \{1, 2\}$ and thus also changed one of it's own input variables' domain. Nevertheless it will not contribute anything when run on D', as it is an idempotent propagator. The idempotence of g can be seen as it does narrow the domain of x depending on the maximum value in $D(y)$, which it does not change itself.

There are of course propagators that are not idempotent. Consider the equation constraint c_2="$3x = 2y$". A propagator for this equation is $k(D)$ defined as
$D(x) := D(x) \cap [\lceil(2\, min\, y)/3\rceil..\lfloor(2\, max\, y)/3\rfloor]$
$D(y) := D(y) \cap [\lceil(3\, min\, x)/2\rceil..\lfloor(3\, max\, x)/2\rfloor]$
When run once on the initial domain D with $D(x) = [0 \dots 3]$ and $D(y) = [0 \dots 5]$, it will change the domain of y to $D'(y) = [0 \dots 4]$ and leave $D(x)$ unchanged ($D'(x) = D(x)$). When run again imediately afterwards it will leave $D'(y)$ unchanged but will change $D'(x)$ to $D''(x) = [0 \dots 2]$. Thus this propagator, called the bounds propagator, is provably not idempotent.

Note that every propagator can be made idempotent by running it in a loop until a fixpoint is reached. The constraint solver in *Mozart* uses this and forbids non idempotent propagators altogether. This simplifies the engine design, but in my opinion it is questionable as there may be circumstances where time is wasted in a loop that could be better invested into running other constraints' propagators inbetween iterations.

Idempotence can be handled dynamically, that is depending on the current domain. To make use of dynamic idempotence, we just need to provide means for a propagator to flag when the domain it returns is a fixpoint. For example $k(D)$ when run on D'' above will change $D''(y)$ to $D'''(y) = [0 \dots 3]$ and there will be no rounding involved then. According to theorem 8 in [HS98] we then know that this propagator is at a fixpoint (i. e. $D^{(4)} = D'''$). So under these circumstances, we can do without re-adding k to Q. We call this strategy new_{didem}.

Evaluation Schulte and Stuckey evaluated the improvements achievable through consideration of idempotence on eleven examples, most of them standard ones. For the implementation they used the Gecode C++ library. It turns out that new_{sidem} uses only 79.0% of the propagation steps of new_{input} on the most favourable problem. Due to the higher overhead for checking the knowledge base whether a propagator is statically idempotent or not, the runtime in Schulte's and Stuckey's implementation improves to just 90.3% in the best case and 104.3% in the worst case. On most problems a little improvement can be noticed. Thus the consideration of statical idempotence brings the chance of reduction of the runtime with a tolerable risk of increasing it.

Dynamic idempotence (new_{didem}) behaves very similar to the static case. The runtime ranges from 90.4% to 102.6% of that of new_{input}. new_{didem} is never more than 3.6 percentage points better or worse than new_{sidem}. So the difference is negligible.

3.2 Events

Another very promising approach is to investigate the actual changes in the domain that must happen before a propagator changes it's fixpoint state. We call a change in the domain of a variable x an event. Typical events are:

det(x) x becomes determined; ($\mid D'(x) \mid = 1$ and $\mid D(x) \mid > 1$)
lbc(x) lower bound of x changes ($min\{D'(x)\} > min\{D(x)\}$)
ubc(x) upper bound of x changes ($max\{D'(x)\} < max\{D(x)\}$)
dmc(x) domain of x changes

Of course these events overlap e.g.: $det(x) \Rightarrow dmc(x)$ and $lbc(x)$ may coincide with $det(x)$.

Static event sets We store for each propagator a set of events $es(f)$ on which the propagator's fixpoint state depends, and only add a propagator to Q if at least one of the events has happened. We also bear in mind idempotence, thus making the function $new_{sevents}$ an improvement of new_{didem}.

As an example we can again use $g \in prop(\text{“}x < y\text{”})$ which we defined as $D(x) := \{u \in D(x) | u < max\{D(y)\}\}$. It turns out that it only depends on the event $ubc(y)$, because it only removes values from x according to the current maximum value of y. When x changes, g will not be able to infer anything and the same holds for any change on $D(y)$ that is not a change of the upper bound. By splitting up the original f above into g and h, we can avoid unnecessary execution of code. Because $es(f) = \{ubc(y), lbc(x)\}$, if only $ubc(y)$ occured and f is run subsequently, the second line of f, which checks the lower bound of x is executed needlessly. This is a pro for the two-propagator approach – we will see the cons later.

Dynamic event sets We can achieve even better results by handling the event sets dynamically, removing events from $es(f)$ as more information on the variables becomes available. This can be used to remove propagators from the propagation process when they are *entailed*, meaning that $\forall D' \sqsubseteq D : f(D') = D'$. We can then set $es(f) := \emptyset$ and f will never be added to Q again.

Here is an example using g again: Consider the domain D with $D(x) = \{1, 2\}$ and $D(y) = \{4, 5, 6\}$. Obviously any change on the upper bound of y can not lead to any contribution of g because even the smallest value in $D(y)$ is still greater than any value in $D(x)$. If we realise this, for example during the first call to g we can remove $ubc(y)$ from $es(g)$. Then the event set of g is empty and we find it is entailed.

In this case having one propagator f for "$x < y$" can be advantageous, because g and its twin h are entailed at the same time. f can just set $es(f) := \emptyset$, but making g care about h and letting it access the event set of h is at least very difficult. In fact in real constraint solvers one uses only a single propagator for this simple kind of constraints. This is because of the advantages when dealing with entailment and also more importantly because every propagator uses quite some amount of storage. As there will be a lot of simple constraints halving the number of propagators for them can reduce memory footprint a lot.

Entailment is not the only case where dynamic event sets are useful. Consider a propagator for the exactly constraint, which enforces that exactly m of n variables are equal to a value k. As soon as k is removed from the domain of one of the said variables, all events considering that variable can be ignored.

We call this new implementation $new_{devents}$ and let it take into account dynamic event sets as described, as well as dynamic idempotence, like explained for new_{didem}.

Evaluation The evaluation shows significant improvements in runtime. Compared to new_{input}, $new_{sevents}$ uses only 74.2% of the time in the best case and 102.4% in the worst case. It seems especially strong for problems with propagators that depend on $det(x)$ events. $new_{devents}$ even achieves 28.1% on one problem and 102.5% in the worst case. Notably neither of the two is significantly slower than both new_{input} and new_{sidem}/new_{didem} on any problem. Also the dynamic event sets are never significantly slower than the static ones. So the implementation using dynamic event sets dominates all others, which should not be a big surprise since it incorporates the strategies from all others.

Static event sets and handling of entailment are common features of current constraint solvers, while dynamic event sets, despite their apparent strengths, have not been widely adopted yet.

4 Choosing a propagator to execute next

In the previous section we discussed how to avoid adding propagators to the set Q which cannot contribute. This section is primarily concerned with the question how we should choose the next propagator that is to be removed from Q and applied to the domain.

The name Q indicates that when actually implementing, we probably will use a queue. A simple and fair policy would be FIFO. This has been used in the evaluation so far. Yet a FIFO queue has the disadvantage that it may delay the detection of a close failure or solution.

For example assume we are in the mid of a propagation phase. Several propagators have been run. Several are not at a fixpoint any more and are now queuing for beinging run: $Q =< d, a, b, f >$. Next the propagator d is run and it determines the domain of variable x to be $D(x) = \{42\}$. This event triggers propagator c which depends on the events $dmc(x)$ and $dmc(y)$. c is inserted into Q. Say running c will make the CSP fail. But because we have a FIFO queue, c is inserted after the last element in the queue and it will be run only after all the other, maybe very time-consuming propagators have been run. So in this case it would be better to favour cheap propagators by prioritising them. So our Q becomes a priority queue and the evaluation will show if it generally leads to faster results.

4.1 Priorities

We assign a *static priority* $n \in \{1, \ldots, k\}$ to each propagator. The priority should reflect the cost of applying the propagator. By "cost" we mean time in this case. Using heuristics tailored to the application domain, we could also try to let the priority reflect the expected impact, that it's execution will have. But the latter idea shall not be topic of this paper, as it is an open research field.

For example we assign priorities imitating complexity classes for algortihms e.g. (CONST, LOGARITHMIC, LINEAR, QUADRATIC, EXPONENTIAL). To get finer granularity on the CONST class, we could differentiate according to the number of variables involved in a propagator's inference process (UNARY, BINARY, TERNARY, LINEAR, QUADRATIC, VERY_SLOW). We could also keep it simple (FAST, SLOW). Most of the current constraint solvers like *SICStus* and *Mozart* stick with the simple variant, while *Choco* has seven priority levels.

Our example propagators g and h could have priority BINARY. A propagator for "$x > 42$" could have priority UNARY. More complex propagators like Régin's for the `alldifferent` constraint [Rég94] would have lower priorities, in this case QUADRATIC.

Once a variable has been determined, propagators may be able to execute faster. Sometimes we can also switch the implementation of the propagator to a faster one in that case. Then of course the priority should be adjusted. We speak about *dynamic priority*.

4.2 Multiple and staged propagators

With a priority queue we can use *multiple propagators* with different complexity and propagation strength for one constraint. Assume we have two propagators r and p with cost(r)$\gg$cost(p), but $r(D) \sqsubseteq p(D)$ for many D. Then we would assign p a higher priority, thereby letting it be executed before r. r and p could be two different propagators for `alldifferent`, e.g. Régin's and Puget's [Pug98] which have complexity $O(n^{2.5})$ and $O(n \cdot logn)$ respectively.

The advantage of multiple propagators is that we can feed other propagators with the results of p. When r is finally run, it can benefit of the propagation that has happened already, including that of p. Thus we may be able to save executions of the costly propagator r.

To make multiple propagators more efficient and reduce management overhead, we can combine them into one *staged propagator*. The staged propagator has an internal "stage" variable which indicates the algorithm that is to be used when the propagator is invoked next.

Figure 2 illustrates how staged propagators work. Call the staged propagator q and let it use one fast but comparatively weak algorithm, call it g, and another slow but strong one (h). Those two algorithms correspond to two propagators when using a multiple propagator approach. On the first run of q, we will execute the cheaper algorithm g (from state B). Then q sets the internal stage variable to use the costlier algorithm h when invoked next (State C). If the current domain is no longer a fixpoint of g, although h may not have been called yet, we set the stage variable to use g when executed next (State B again). If a fixpoint of h is reached, then g is at a fixpoint, too (State A).

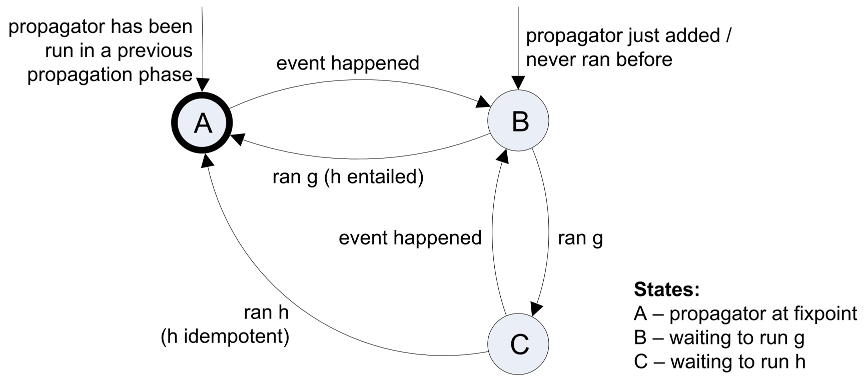
propagator has been run in a previous propagation phase
propagator just added / never ran before
event happened
A
B
ran g (h entailed)
event happened
ran g
ran h (h idempotent)
C
States:
A – propagator at fixpoint
B – waiting to run g
C – waiting to run h

Fig. 2. The stages/states of a staged propagator

If the system supports dynamic priorities, we can adjust the priority of q according to the current stage. If the cheaper algorithm g is going to be run next, we give q a higher priority. This would happen upon entering state B in the diagram. When h is the next to be run, q should have a lower priority. So we set the priority to a low value, corresponding to the cost for running h, upon entering state C.

Apart from the reduced administration effort needed because of fewer propagators, staged propagators are also superior to multiple propagators because of their better handling of entailment. Assume g knows after it's execution that h is entailed. Then it can remove the whole propagator q from the queue. With multiple propagators this would not be possible.

4.3 Evaluation

Schulte and Stuckey again compared the runtime of the resulting new propagation algorithms to that of the simple new_{input} without priorities. They tested simple, multiple and staged propagators, with and without static and dynamic priorities on fifteen benchmarks. We try to summarise the 135 single results into a few rules of thumb.

The introduction of priorities is nearly always beneficial. In the best case they achieve a reduction to 51.8%, but in the worst case the time needed grows to 125.5%[1]. For each static and dynamic priorities there were just five benchmarks for which the runtime decreased. When combining priorities with staged propagators, there was only one problem remaining where the runtime stayed at 108.8% with dynamic priorities. It is difficult to decide whether the static or the dynamic priority approach is better. Combined with staged propagators the runtime of both is often very similar although the difference between the two goes up to 25%.

Staged propagators as well as multiple propagators need priorities to work well. Nevertheless staged propagators could even without priorities reduce the runtime in the best case to 82.2% without inceasing the runtime in the worst case. When using priorities and staged propagators are applicable to the problem, they are never worse and often better than simple propagators (best case: 46.4%, worst case: 99.6%). Multiple propagators are never significantly better than staged propagators, while staged propagators strictly dominate multiple propagators on some problems.

Schulte and Stuckey claim in their summary of the measured data that "dynamic priorities are slightly advantageous over static priorities in most cases", yet their measured data seems to suggests that neither one is superior to the other. The interested reader may see for herself in [SS04b].

[1] both with static priorities (dynamic very similar)

5 Summary

After some introductory words and terminology we have presented a model of a constraint solver. We looked in detail at the constraint propagation algorithm and demonstrated that it does find a common fixpoint of all propagators. We saw how to speed up propagation by keeping track of the propagators' fixpoint state and trying to only execute those propagators currently not at a fixpoint with the help of a queue.

After that we identified the two main starting points for performance improvements, the choice of the next propagator to execute and the addition of only the necessary propagators to the queue. We reasoned about when propagators are no longer at a fixpoint, as well as we introduced and used the concepts of idempotence and propagator's dependence on events. The evaluation done by Schulte and Stuckey showed us that both are useful tools to save propagation steps and time. For choosing the next propagator for execution we assigned priorities to the propagators and showed that it is very often beneficial to run cheapest propagators first. Staged propagators were introduced as a useful method for speed-up, provided that the problem structure allows having them.

As seen in the evaluation the techniques covered can sometimes reduce the number of steps drastically, although the best improvement we saw is a reduction of the runtime by factor four. So no miracles should be expected, even though it surely makes a big difference whether one has to wait one or four hours for the solution to a CSP.

References

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[HS98] Warwick Harvey and Peter J. Stuckey. Constraint representation for propagation. *Lecture Notes in Computer Science*, 1520:235–249, 1998.

[Pug98] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 359–366, Menlo Park, July 26–30 1998. AAAI Press.

[Rég94] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence*, volume 1, pages 362–367, Seattle, Washington, July–August 1994. AAAI Press.

[SS04a] Christian Schulte and Gerd Smolka. Finite domain constraint programming in oz. a tutorial, 2004.

[SS04b] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.

The author wishes to thank Guido Tack for providing expert-knowledge and many useful comments.