Watched Literals for Constraint Propagation in Minion *

Ian P. Gent[1], Chris Jefferson[2], Ian Miguel[1]

[1]School of Computer Science, University of St Andrews, St Andrews, Fife, UK
[2]Oxford University Computing Laboratory, University of Oxford, Oxford, UK
{ipg,ianm}@dcs.st-and.ac.uk, Chris.Jefferson@comlab.ox.ac.uk

Abstract. Efficient constraint propagation is crucial to any constraint solver. We show that *watched literals*, already a great success in the propositional satisfiability community, can also be used to provide highly efficient implementations of constraint propagators. We describe in detail three important aspects of watched literals as we apply them to constraints, and we describe how they are implemented in the Minion constraint solver. We show three successful applications of watched literals to constraint propagators: the sum of boolean variables; generalised arc consistency for the 'element' constraint; and generalised arc consistency for the 'table' constraint.

1 Introduction

Efficient constraint propagation is the bedrock of any implementation of constraint programming. We show that *watched literals* can be used to provide faster implementations of constraint propagators. Watched literals are one of the main reasons for the dramatic improvements seen in SAT solvers in this decade [12]. They allow the key progagation algorithm, unit propagation, to run much faster than previous implementations. Used as triggers to fire constraint propagations, watched literals have three features different to triggers as normally used. Watched literals only cause propagation when a given *variable-value pair* is deleted; their triggering conditions can be changed *dynamically* during search; and they remain *stable* on backtracking so do not use memory for restoration. These features can be used separately, and we report on doing so.

We first identify what we consider to be the two key benefits of watched literals: the reduction of work when no propagation is possible, and the reduction in work on backtracking. Then we detail the three aspects of watched literals mentioned above. Following that, we report on implementations of three different constraints using watched literals.

* Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Fellowship. This research was supported by EPSRC Grants GR/S30580 and EP/D032636 and also received help from the EPSRC funded SymNET. We thank Peter Jeavons, Steve Linton, Iñes Lynce, Angela Miguel, Karen Petrie and Judith Underwood.

The first is a slight generalisation of the SAT clause, and so it is no surprise that this allows us to improve the speed of propagating SAT clauses in a constraint solver. In fact, our results show that SAT clauses propagate almost as fast as in a state-of-the-art SAT solver.

Next, we show that watched literals are ideally suited to implementing generalised arc consistency (GAC) for the 'Element' constraint. Using classical triggers, it is hard to achieve GAC efficiently: as a result we are unaware of literature on how to do so. Using watched literals, we develop a GAC algorithm in a very natural progression from the definition of what GAC means for this constraint. Compared to a non-GAC propagator, we can improve run-times several-fold on instances of Langford's problem.

Our third example is to implement GAC for the 'Table' constraint, i.e. an arbitrary constraint specified by an explicit list of allowed tuples. We base our implementation on GAC-2001/3.1 [4]. By using watched literals, we automatically convert this from being a constraint-based to a support-based algorithm. Furthermore, by the nature of watched literals, we do not need to restore the state of support data structures on backtracking.

We report on how we provide the infrastructure to allow watched literals to work in a practical constraint solver. All our implementation and experimentation is based on the MINION constraint solver [7]. MINION's design principles are to reduce user choice to allow greater optimisation; and to maintain a low memory profile, most especially in use of memory for state information that needs to be restored on backtracking. As a result, we have reported MINION running between one to two orders of magnitude faster than existing systems such as ILOG Solver [10] or GeCode [13]. Thus, the run-time improvements we report here are compared to the state of the art. Watched literals have been incorporated into MINION. Since MINION is open source, our implementations are available for use and research.[1]

2 Motivation and Terminology

In this section we motivate why watched literals let algorithms improve the efficiency of constraint propagation. First, we introduce our terminology, to clarify the three ways in which watched literals differ from classical triggers.

Conventionally, constraints are triggered by a particular variable's domain being changed in one of three ways: by being assigned, by having any value in its domain removed, or by having either its lower or upper bound removed. We call these kinds of triggers **classical**. A watched literal can trigger on the removal of any given variable-value pair in the problem. We use the word "literal" for a variable-value pair. This is completely consistent with its usage in SAT, where a literal is a variable together with a polarity. A **literal trigger** is one which causes its constraint to act when the given value is removed from that variable. An important reason for using the word 'literal' is to avoid a clash with the classical 'domain trigger', which will fire when *any* value of its variable is removed. After a watched literal is triggered, its constraint can move the trigger to another literal, delete it, or add a new watched literal. We say that a **dynamic trigger**

[1] `http://minion.sourceforge.net`. We used revision 138 in this paper.

is one which can be moved in this fashion. Dynamic triggers are a valid concept on non-literal triggers. We can have dynamic triggers which fire on domain removal, bounds removal, or variable assignment, and these are implemented in MINION. By contrast, a **static trigger** is one whose firing conditions are guaranteed to remain fixed during the lifetime of a constraint.[2] Watched literals have to remain valid on backtracking, so that we can avoid storing their state information in backtrackable memory. A trigger with this property is a **backtrack-stable trigger**, or just 'stable trigger' for short.[3] We explain this concept in more detail below. Finally, for brevity and to emphasise the historical linkage with the SAT concept, we say that a **watched trigger** is a dynamic and backtrack-stable one. Thus a **watched literal** is just a dynamic, stable, literal trigger.

It might be helpful to think of a dynamic trigger being a generalisation of a classical bounds trigger. A bounds trigger fires, say, on the value 2, but after it propagates it will fire on the new lower bound, perhaps 7. For a dynamic trigger, the constraint itself decides where the new trigger will be, rather than being fixed at the new lower bound. The constraint has complete flexibility in assigning triggers. It can move the trigger to another value of the same variable, to any value of any other variable in the constraint. It can move some or all dynamic triggers associated with a constraint to different places. It can even add or delete dynamic triggers.

We see the key advantage of watched literals and dynamic triggers as the flexibility it gives constraint propagators to avoid unnecessary work. In many situations where a constraint cannot propagate, it causes *no work at all*. Doing no work, yet with a guarantee that we are not missing any propagation, is better than doing any work of any kind, even if that work is highly optimised and very cheap. To enable this, each constraint using dynamic triggers must provide what we call the **propagation guarantee**: that its intended level of consistency is satisfied *unless* at least one of its dynamic triggers fires. For a watched literal, when a domain value is deleted, we only need to wake up constraints which are watching that literal (i.e. that value of its variable.) The advantage extends to, for example, dynamic assignment triggers. A constraint on ten variables may be able to set up dynamic assignment triggers on just two of them: this constraint will cause no work when of the other eight are assigned.

Watched triggers (but not dynamic triggers) have another significant advantage because they are backtrack-stable. This lets us reduce memory management overheads greatly. Conventionally, on backtracking, we ensure that the state of all propagators are exactly as they were when we left that node going forwards, necessitating some data structures and maintenance to support this. Backtrack-stability lets us lift this restriction. It is enough to ensure that when we backtrack, all constraints are in a state which lets them provide the propagation guarantee. It is perfectly acceptable for a constraint's set of watched triggers to be completely different on returning to a node to the set when we left the node, as long

[2] Under this definition a classical bounds trigger is *static* although the exact domain removals that trigger it will change.

[3] We thank Steve Linton for suggesting this name.

as both sets provide the propagation guarantee in that search state. The ideal way to achieve this is for constraints to define sets of watched triggers which do not need to be changed on backtracking, i.e. backtrack-stable triggers. Where we can do this perfectly, the constraint needs put nothing into backtrackable memory. This means that no copying is done on backtracking, so once again we have reduced the cost associated with the constraint to zero.

Defining backtrack-stable triggers can require nonintuitive thinking, but some general rules apply and we will see examples in the detailed descriptions to follow. Many cases are simple, because the notion of support monotonic in many constraints: i.e. if a set of values supports some literal, they still support it as we restore values to domains on backtracking. However, more subtle cases arise, as in GAC for the table constraint in Section 5. When we fail to find support for a value and so the constraint forces its deletion, the natural thing to do is to remove the watch on the literal. This is correct for dynamic triggers but not for watched triggers. When we backtrack, the value we have just deleted will be restored to its domain, as will the literal whose deletion caused its removal. If we delete the trigger, we will lose the propagation guarantee on backtracking. Instead, we leave the watched literal in place.

There is one general disadvantage that should be mentioned with watched triggers, although not with dynamic triggers. Because triggers may move arbitrarily between leaving a node and returning to it, it is often not possible to use a propagation algorithm which is optimal in the worst case in terms of propagation work performed down a single branch. An example is our variant of GAC-2001/3.1 below. This has to be set against the great potential for efficiency gains we have outlined, so we do not think this will often be a major disadvantage. The same problem arises in SAT but watched literals are very widely used, and we will see experimentally that our non-optimal watched version of GAC outperforms an optimal but unwatched one.

3 Watched Literals for Sum of Booleans

The first constraint we consider is the sum of an array of boolean variables B being greater than or equal to a constant value c. Where $c = 1$ this is exactly the case of a SAT clause, and indeed in that case our method will be exactly the standard means of using watched literals in SAT clauses [12]. As such it serves as a good introduction to watched literals for constraint programming. Our approach is only novel in being incorporated in a constraint solver: watched literals for the (more general) weighted sum of a boolean array have ben described by Chai and Kuehlmann and implemented in the solver Galena [5].

We will watch $c + 1$ different literals in the domain of $c + 1$ different variables in B. In each case we watch the value 1. If, at initialisation, we can find only c such literals we immediately set them all to be 1 to satisfy the constraint, and if we cannot find even that many we fail: in neither case is any more work on this constraint required. In general, the watched literals are $B[i_1] = 1, B[i_2] = 1, \ldots, B[i_{c+1}] = 1$. The watched literals, on their own, more than satisfy the constraint, so all other literals could be set to 0 without any propagation happening.

```
OneRemovedFromWatchedVar(i)
      Triggered by DomainRemovalOf B[i] = 1
A01   j := Last
A02   repeat j := j + 1 mod n − c − 1
A03   until (j = Last) or (1 ∈ Domain(B[UnWatched[j]])
A04   if (j ≠ Last) then              // We found a new literal to watch, so update data structures
A04.1     MoveWatchFrom B[i] = 1 To B[UnWatched[j]] = 1;
A04.2     UnWatched[j] := i; Last := j
A05   else                            // No new literal found to watch, so at most c 1's possible in B
A05.1     foreach k such that WATCHED[B[k] = 1]
A05.2         if(k ≠ i) then AssignVariable(B[k] = 1)
```

Fig. 1. Propagator for the Boolean Sum constraint. `UnWatched` is an array of integers. Its values are all different and represent the set of unwatched literals: in general it will not remain sorted during search but there is no need for it to be. `Last` points to the array element in `UnWatched` that was set to be watched the last time we updated this constraint. Neither `UnWatched` nor `Last` needs to be restored on backtracking: therefore both reside in non-backtrackable memory. This code assumes services provided by our watched literal infrastructure. AssignVariable sets a variable to a value and triggers necessary propagations while MoveWatchFromTo updates the watched literal store.

When one of the values being watched is removed, we have some $B[i_j] = 0$. In this case, we have to find a new value i'_j so that $B[i'_j]$ still has 1 in its domain and is not one of the other k literals currently being watched. In the worst case, we have to scan each unwatched variable in the array to see if it has 1 in its domain. If we find such a literal, we simply stop watching $B[i_j] = 1$ and start watching $B[i'_j] = 1$: we call this moving the watch from one literal to the other. ¿From the propagation guarantee, we cannot propagate until one of these literals is deleted. On backtracking, we can only make the constraint looser by enlarging domains, so the watched literals are backtrack-stable. The more interesting case is if we cannot find any other literal to watch. In this case, we know there are at most c literals with 1 in their domain, and all must be set to 1 to satisfy the constraint. So we can immediately set all of them to be 1. This will either cause immediate failure (if another one of the watched literals can no longer be 1) or guarantees to satisfy the constraint. However, we continue to watch all the $c+1$ literals we were watching previously. All watched literals are now set and will cause no propagation, so there is no cost in leaving them in place. Pseudocode for this is shown in Fig. 1. As mentioned above, it is essential to leave the watches in place so that they will be correct on backtracking. It may not be a general rule, but certainly in the cases studied in this paper it is always correct to leave watches in place when a constraint is either satisfied or unsatisfied. The last time the constraint was fired, the propagation guarantee held and the set of watched triggers were backtrack-stable. Therefore, if we backtrack past the current point

in search, we return to a state where we know a set of backtrack-stable triggers satisfied the propagation guarantee.

Our first set of experiments are on SAT problems. These are encoded into a constraint problem with a boolean variable for each SAT variable and a boolean sum greater than 1 for each clause. In no way can MINION compete with specialist SAT solvers because it does not have specialised heuristics or techniques such as nogood learning. However, these experiments demonstrate the value of incorporating watched literals into a constraint solver for this kind of constraint. We compare MINION using its sum-$\geq$ constraint using classical triggers against our new implementation using watched literals. For reference we also compare these against the same model encoded into ILOG Solver 5.3 and a state-of-the-art SAT solver, MiniSAT [6]. We ran on two sets of SAT benchmarks from SATLib [9], uniformly unsatisfiable random instances with 150 variables, and the QG benchmark set. Results are shown in Table 1. We ran experiments on a variety of machines for this paper, but any comparison between methods on an identical instance were run on a single machine for consistency to enable valid comparisons. Our SAT experiments were run under Windows (XP SP2), with a Pentium 4 3GHz and 2GB of RAM, compilation being done using g++ 3.4.4 under cygwin.

On random instances, watched literals are a slight overhead, presumably because all clauses have only 3 literals. Both versions of MINION outperform Solver on random instances. MiniSAT easily outperforms both versions of MINION, but note that MINION does in fact search more nodes per second than MiniSAT: however the very small runtimes for MiniSAT means that these figures may be unduly affected by setup times.

For the structured QG benchmarks, we see that in all cases the watched literal propagator outperforms the classical version by up to 6 times in run time. ILOG Solver does outperform MINION on a number of instances, but only on easy instances: we suspect that this is because of MINION's larger initialisation costs. MINION searches faster on all instances where either solver takes even 10 seconds. On hard instances MINION can search many times faster than Solver, up to 82 times in the case of QG2.8. MINION is again outclassed easily by MiniSAT. However, on instances where it takes 10 seconds, MINION, is never as much as 6 times slower than MiniSAT in terms of decisions per second. Certainly MINION is not a competitive SAT solver. However, the efficiency with which it propagates clauses does suggest that it is ideally suited for solving instances containing a hybrid of significant numbers of SAT clauses together with the more expressive constraints usually used by constraint programmers.

4 Element Constraint

The "Element" constraint is a great addition to the expressivity of a constraint language. It allows one to use an integer variable for the index of an array [8]. Suppose that A is an array of n integer variables and that $Index$ and $Result$ are two more integer variables. The Element constraint is that $A[Index] = Result$.

	Classical			Watched		Ilog Solver		MiniSat		
Problem	Time(s)	Nodes	Nodes/s	Time(s)	Nodes/s	Time(s)	Nodes/s	Time(s)	Decisions	Dec./s
Random	96.20	621,798.1	**63,360**	100.0	62,180	462.9	13,225	0.0858	3,577.1	41,614
QG1.7	8.13	58	7	7.83	7	2.77	**21**	0.14	121	864
QG1.8	628.47	188,270	300	101.89	**1,848**	*3604.09*	27	0.75	7,951	10,601
QG2.7	8.16	32	4	7.89	4	2.20	**14**	0.125	54	432
QG2.8	1,137.31	340,747	300	165.16	**2,063**	*3604.09*	25	10.328	48,933	4,737
QG3.8	1.08	150	139	1.06	141	0.36	**415**	0.046	283	6,152
QG3.9	18.42	82,404	4,474	14.09	**5,847**	107.63	766	6.703	55,003	8,205
QG4.8	1.06	909	855	1.03	882	0.92	**987**	0.078	1,003	12,859
QG4.9	1.56	461	295	1.52	304	0.94	**491**	0.046	28	608
QG5.9	2.83	187	66	2.75	68	1.30	**143**	0.62	26	419
QG5.10	5.44	1,453	267	4.63	**314**	10.84	134	0.093	74	795
QG5.11	6.77	506	75	6.20	**82**	7.97	63	0.14	79	564
QG5.12	269.84	139,581	517	76.33	**1,829**	2132.67	65	0.296	1,440	4,864
QG5.13	4,847.28	1,798,176	371	1,125.83	**1,597**	*3602.88*	48	6.156	30,776	4,999
QG6.9	2.19	28	13	2.13	13	0.53	**51**	0.046	16	348
QG6.10	3.53	313	89	3.33	94	2.05	**153**	0.109	458	4,202
QG6.11	7.58	2,522	333	5.66	**446**	21.03	120	0.296	2,632	8,892
QG6.12	50.09	26,847	536	20.95	**1,281**	358.88	75	4.484	22,519	5,022
QG7.9	2.19	8	4	2.14	4	0.50	**14**	0.046	8	174
QG7.10	3.81	816	214	3.47	**235**	4.58	178	0.093	124	1,333
QG7.11	16.64	11,616	698	8.16	**1,424**	104.94	111	0.187	1,866	9,979
QG7.12	277.41	159,907	576	70.53	**2,267**	2370.37	68	0.593	5,731	9,664
QG7.13	786.55	312,108	397	180.66	**1,728**	*3602.19*	52	0.265	1,307	4,932

Table 1. Experiments with Random SAT instances (mean of 100) and QG SAT instances. Bold indicates which of the three constraint solvers searched most nodes per second – nodes searched in each case were identical. Italics in the Solver column indicate timeouts after 1 hour. The number of SAT variables in QG$i.n$ is n^3.

Although the constraint is implemented in constraint toolkits such as ILOG Solver, GeCode, and Choco [11], we are not aware of a literature on how to propagate it. This may be because, using classical triggers, some aspects of Element are hard to propagate efficiently enough to repay the overheads.[4] In contrast, using watched literals, establishing and maintaining GAC is straightforward to implement and efficient. In particular, where little propagation occurs little overhead is incurred. We start by establishing the exact definition of GAC:

Theorem 1. *The domains of variables in an Element constraint are Generalised Arc Consistent if and only if all of the following hold:*

$$Dom(Index) = \{i\} \implies Dom(A[i]) \subseteq Dom(Result) \tag{1}$$

$$i \in Dom(Index) \implies Dom(A[i]) \cap Dom(Result) \neq \emptyset \tag{2}$$

$$Dom(Result) \subseteq \bigcup_{i \in Dom(Index)} Dom(A[i]) \tag{3}$$

[4] We believe that Choco does GAC on Element, but that Solver 5.3 and GeCode 1.0.1 do not.

Proof: (If) Suppose all the conditions hold. By (2) there is a value to support each value of $Index$. If $Dom(Index)$ is a singleton than (1) and (3) show that $Dom(A[Index]) = Dom(Result)$, and any value of one variable is supported by the same value of the other. If $|Dom(Index)| > 1$, every value for each $A[i]$ is unconstrained since it is supported if $Index \neq i$. Also, (3) ensures that each value j of $Result$ is supported by a pair $Index = i, A[i] = j$. (Only If) Suppose the domains of $Index$, $Result$, and each $A[i]$, are GAC. If $Dom(Index) = \{i\}$ then certainly any value in $Dom(A[i])$ must be in $Dom(Result)$. If $i \in Dom(Index)$ then there must be at least one value in the domain of both $Dom(A[i])$ and $Dom(Result)$. Finally, for any value $v \in Dom(Result)$ there must be an index $i \in Dom(Index)$ such that $v \in Dom(A[i])$. ***QED***

This proof shows the close link between the conditions and the *support* for each value of each variable in the constraint. We make this explicit, by describing support for each value of $A[i]$, $Index$ and $Result$. Notice that each kind of support involves at most two other variables, instead of a tuple involving all variables in the constraint. There are three cases, corresponding exactly to(1), (2) and (3) above.

Support for $A[i] = j$**: either** $|Dom(Index)| > 1$ **or** $j \in Dom(Result)$.
Support for $Index = i$**:** any j such that $j \in Dom(A[i])$ and $j \in Dom(Result)$.
Support for $Result = j$**:** any i such that $i \in Dom(Index)$ and $j \in Dom(A[i])$.

We now describe how we proceed if the support is lost, i.e. when values are removed which were supporting a variable-value pair in one of the cases above.

1. There are two cases where we might have lost support for $A[i] = j$. The support in either case is independent of i, depending only on the values of $Index$ or $Result$ respectively.
   (a) If $Index$ is assigned to i (i.e. its domain is the singleton $\{i\}$) then we have to remove all values from $Dom(A[i])$ not in the domain of $Result$. This can be achieved by a classical trigger on the assignment of $Index$.
   (b) If a value j is removed from the domain of $Result$, and $Index$ has already been assigned to i, then we remove j from the domain of $Dom(A[i])$. There are a number of ways to achieve this: in our pseudocode and implementation we use a classical domain reduction trigger on $Result$.
2. If the value j is removed from either $Dom(A[i])$ or $Dom(Result)$, we try to find another value j' in the domain of both. If we succeed, this is the new support for $Index = i$. If not, then we remove i from the domain of $Index$. We implement this by *watching* the literals $A[i] = j$ and $Result = j$. If we find j' then we move the watched literals to be on $A[i] = j'$ and $Result = j'$. If we do not find a new support, we leave the watched literals in place, so that they will be correct when j returns to the domain on backtracking.
3. If the value i is removed from $Dom(Index)$, or j is removed from $Dom(A[i])$, we try to find another pair of values i', j' with $i \in Dom(Index)$ and $j \in Dom(A[i])$. These will be the new support for $Result = j$, or else we insist that $Result \neq j$. We implement this by *watching* $Index = i$ and $A[i] = j$, moving these to $Index = i'$ $A[i'] = j'$ if possible. If not, we again leave the watches in place.

We find this development very natural. We started from the definition of domains being GAC, moved to what is needed to support each value of each domain, and finally showed how new supports could be sought when old supports are lost. This last stage can be implemented in MINION. Pseudocode for this implementation is in Fig. 2, where the four constituent functions correspond exactly with the four cases above. We have not discussed initialisation because it is a straightforward variant on the pseudocode of Fig. 2. Finding the initial watched values is essentially the same as finding new ones from old, and values are removed if we cannot find initial supports for them.

There are powerful efficiency reasons for using watched literals for the Element constraint, instead of on any of the conventional trigger types. For each i in $Dom(Index)$ we are watching two things, and for each j in $Dom(Result)$ we are watching two things. If the array is size 500 and each domain is of size 100, we therefore have 1,200 literals to watch in addition to the conventional triggers on $Index$ and $Result$. Yet in total in $A[i]$, $Index$, and $Result$, there are 500,600 domain elements, so we watch less than 0.25% of all domain elements. This argument also shows why it is important that we do not need to restore trigger data on backtracking when using watched literals. If we did have to, we would have to restore the values of 1,200 triggers on backtracking, an overhead which would outweigh the benefits we have gained. The only genuine efficiency loss compared to conventional methods happens when we fail to find new support, for example in line C01 in Fig. 2. When we return to a node on backtracking, our watched literals may have moved any number of times since we left the node going forwards. We have to search the whole domain instead of just the remaining part of the domain.

To compare with our watched literal implementation, we also wrote two other propagators using classical triggers. Both are triggered any time any literal is removed from a variable, and are told the literal removed. One of these accomplishes GAC, by looking at all literals which could have lost support by the removal of this literal, and checking if support for them still exists.[5] The code for this is very similar to that for the watched literal case, but without storing support when it is found. This has the advantage of being very space efficient, using no memory which must be backtracked at all, at the cost of extra computation at each node. The second, non-GAC implementation, checks only each time any variable is assigned, and performs only those checks which are possible in O(1) time.

Langford's Problem is problem 24 in CSPLib. Given k and n, $L(k, n)$ requires finding a list of length $k*n$, which contains k sets of the numbers 1 to n, such that for all $m \in \{1, 2, \ldots, n\}$ there is a gap of size m between adjacent occurrences of m. For example, `41312432` is a solution to $L(4, 2)$. We modelled the family of instances $L(2, n)$ in MINION using two vectors of variables, V and P, each of size $2n$. Each variable in V has domain $\{1, 2, \ldots, n\}$, and V represents the result. For each $i \in \{1, 2, \ldots\}$ the $2i^{th}$ and $2i + 1^{st}$ variables in P are the first and second positions of i in V. Each variable in P has domain $\{0, 1, \ldots, 2n-1\}$,

[5] This approach is similar to the GAC algorithm in Choco, as seen in its source code.

GAC for Element Constraint: $A[Index] = Result$

```
SupportLostForArrayValue-a(i)
    Triggered by AssignmentOf Index = i
                        // Remove from domain of A[Index] any values not in domain of Result
A01   foreach k in the current domain of A[i]
A01.1     if k ∉ Domain(Result) then
A01.2         RemoveFromDomain(A[i],k)

SupportLostForArrayValue-b()
    Triggered by AnyDomainRemovalOf Result
                        // If Index is assigned to i then Dom(A[i]) ⊆ Dom(Result)
B01   if VariableIsAssigned(Index) then
B02       foreach k in the current domain of A[i]
B02.1         if k ∉ Domain(Result) then
B02.2             RemoveFromDomain(A[i],k)
```

```
SupportLostForIndexValue(i, j)
    Triggered by RemovalOf A[i] = j or RemovalOf Result = j
        // Previously we supported Index = i because j ∈ Domain(A[i]) ∩ Domain(Result)
                        // Now we must find a replacement for j or insist that Index ≠ i
C01   foreach k in the current domain of A[i]
                        //See caption for details of how we step through domains
C02       if k in the current domain of Result then
C02.1         MoveWatchFrom A[i] = j To A[i] = k;
C02.2         MoveWatchFrom Result = j To Result = k;
C03.3         return                    // We are finished, leave function
C04   endforeach                       // We failed to find a new support so Index ≠ i
C05   RemoveFromDomain(Index,i)
```

```
SupportLostForResultValue(i, j)
    Triggered by RemovalOf Index = i or RemovalOf A[i] = j
    // Previously we supported Result = j because i ∈ Domain(Index) and j ∈ Domain(A[i])
                        // Now we must find a replacement pair for (i, j) or insist that Result ≠ j
D01   foreach k in the current domain of Index
D02       if j in the current domain of A[k] then
D02.1         MoveWatchFrom A[i] = j To A[k] = j;
D02.2         MoveWatchFrom Index = i To Index = k;
D03.3         return
D04   endforeach                       // We failed to find a new support so Result ≠ j
D05   RemoveFromDomain(Result,j)
```

Fig. 2. Propagator for the Element constraint. Note that we mix types of triggers freely. The first function triggers when a variable is assigned, and the second triggers on a domain removal. The final two functions both trigger when one of two literals being watched is removed. At line C01 search the domain starting from j, and looping back to j if we reach the end of the domain. A similar approach is taken at D01.

		Non-GAC			Classical-GAC		Watched-GAC		
	Solutions	Time(s)	Nodes	Nodes/s	Time(s)	Nodes	Time(s)	Nodes	Nodes/s
L(4,2)	2	<0.1	378	-	<0.1	46	<0.1	46	-
L(5,2)	0	0.1	6,956	-	<0.1	694	<0.1	694	-
L(6,2)	0	2.3	169,275	73,000	0.6	15,388	0.3	15,388	59,000
L(7,2)	52	76.8	4,912,580	64,000	16.7	413,573	7.5	413,573	55,000
L(8,2)	300	3,276.9	176,320,552	54,000	635.7	13,471,366	267.2	13,471,366	50,000

Table 2. Comparison of propagators in MINION on Langford's problem.

indexing matrices from 0. The constraints are given below, where i ranges from 1 to n. We write $=_{\texttt{elem}}$ to distinguish the usage of the element constraint from ordinary indexing of a vector by a constant.

$$\begin{aligned} V[P[2*i]] &=_{\texttt{elem}} i \\ V[P[2*i+1]] &=_{\texttt{elem}} i \\ P[2*i] &= i + P[2*i+1] \end{aligned}$$

We found all solutions to Langford's problem up to $n = 8$ using this model. Experiments for increasing values of n were performed for our three propagators for element presented in this paper. These were run under Mac OS X (10.4.6) running on a 1.2Ghz PowerPC G4 with 768MB RAM compiled with g++ 4.0.1. Results are presented in Table 2. Performing GAC on the element constraints improves solving time by an order of magnitude. The watched GAC propagator is over twice as fast as the non-watched GAC propagator, and in terms of nodes searched per second is only slightly slower than the non-GAC propagator. We also performed experiments on constraint encodings of Quasigroup construction problems, shown in Table 3. Experiments were performed under Windows as described earlier, and under Linux (Fedora Core 4) with a Pentium 4 3.4 GHz dual processor with 8GB RAM, with compilation by g++ 4.0.2. On QG3, GAC performs no additional propagation, but remarkably our GAC algorithm using watched literals is *faster* than the non-GAC algorithm. We suggest this is because of the benefit of watched literals avoiding unnecessary work. On QG7, we get significant search reductions, but here speed per node is greatly reduced by the extra work and the overheads are not repaid. In both cases the watched literal GAC propagator is much faster than the classical version.

5 Watched-GAC for the Table Constraint

The 'Table constraint' provides generalised arc consistency for any user-defined constraint, given by a list of acceptable tuples of the variables involved in the constraint. This can be very useful where critical parts of a problem have no natural expression in primitive constraints, but which need to be propagated effectively. The table constraint can be implemented using any GAC algorithm, provided it is suitably adapted to work correctly in a backtracking environment. We base our algorithm on GAC-2001/3.1 [4].

	Non-GAC			Classical-GAC		Watched-GAC		
Problem	Time(s)	Nodes	Nodes/s	Time(s)	Nodes	Time(s)	Nodes	Nodes/s
QG3.6	0.016	51	-	0.031	51	0.031	31	-
QG3.7	0.031	11	-	0.047	11	0.016	11	-
QG3.8	7.031	105,414	14,992	43.453	105414	6.453	105,414	16,335
QG3.9	0.047	26	-	0.141	26	0.031	26	-
QG3.11	0.078	132	-	0.375	132	0.078	132	-
QG7.7	<0.1	844	-	0.03	311	0.02	311	-
QG7.8	0.08	12,450	155,625	0.5	4,628	0.33	4,628	14,024
QG7.9	<0.1	233	-	0.02	83	0.01	83	-
QG7.10	205.56	31,383,717	152,674	840.33	3,408,114	329.8	3,408,114	10,333.9

Table 3. Comparison of propagators in Minion on Quasigroup Existence Problems. QG3.10 was not solved within an hour by any of the three approaches. QG3 performed under Windows, QG7 under Linux.

This section shows the third significant benefit of using watched literals. We have already seen it greatly speed up propagation of SAT-like constraints in a CP environment; and enable the simple implementation of GAC for the element constraint more efficiently than any other method we know of. Here, we show that we can convert an easy-to-implement but coarse-grained GAC algorithm into a fine-grained algorithm where the required data structure maintenance is provided entirely by an already-implemented central infrastructure. GAC-2001/3.1 was presented as a *coarse-grained* algorithm, i.e. it is a constraint-oriented propagation algorithm [4]. By using watched literals, we convert it into a *fine-grained* algorithm, i.e. "the deletion of a value in the domain of a variable will be propagated only to the affected values in the domains of other variables" [4]. The watched literal infrastructure provides the services to make this happen automatically, so we need only minimal adaptations to a non-watched version of the algorithm. This is enormously much more straightforward than the complicated data structures which need to be implemented to enable classical fine-grained GAC algorithms such as AC-6, AC-7, or GAC-Schema [3, 2, 1] to work correctly and efficiently on backtracking. There is a penalty to pay, which is that our version of GAC-2001/3.1 is no longer time-optimal in the worst case.

Implementing Watched-GAC is straightforward using watched literals. Pseudocode is given in Figure 3. Each literal has associated with it a set of triggers. These are at the start attached to the literals which provide the first support for the literal that can be found. During search, if any of these literals are deleted, the trigger is activated, and either a new support is found, or if support can be found the literal no longer has support and is deleted. As with earlier examples, there is no need to backtrack these supports. If a new support is found, that support will also be valid on backtracking all the way from the current node to the root node. If a new support cannot be found, then we leave the triggers where they are. When search backtracks past this search node, the deleted literal will be restored, as will the literals in the old support, as they must have all been present at the end of the previous node, else they would have been moved then.

This does introduce one problem when compared with GAC 2001. In GAC 2001 as the tuple supporting each literal is backtracked during search, then at any node we know that as we search for support through the tuples in a fixed order, then no earlier tuple in the ordering could possibly be a valid support, as we have already checked it. This means that every tuple is checked at most once. The watched implementation of GAC-2001 does not have this property, as down one branch of the search tree it may be necessary to search far through the list of tuples to find a valid support. When search then backtracks and continues down another branch, it may be that an earlier tuple than the current one provides support. Therefore when checking for support, it is necessary to scan through all tuples. The current implementation always begin searching from the current support, which means that at any particular node each possible tuple will be checked at most twice, as after one pass through the tuples there must be no support, which may require one more pass through to prove. Also, this behaviour could be repeated at several nodes down a branch. We also implemented GAC for the table constraint using dynamic (i.e. non-watched) triggers. Code is almost identical to that for watched literals, except that we no longer have to loop in searching for support, therefore we retain time optimality down a branch. The penalty is the overhead of storing dynamic triggers and support information in backtrackable memory.

The prime queen attacking problem (number 29 at www.csplib.org) is, given n, to put a queen and the numbers 1, ... n^2 on the cells of an an $n \times n$ chess board such that any number i is reachable via a knight's move from the cell containing $i-1$. Furthermore, the number of primes not attacked by the queen should be minimised (the queen does not attack its own cell). To model this problem, we introduce a one-dimensional matrix V of n^2 variables, each with domain $0..(n^2-1)$ to indicate the cell to which each value is assigned. To constrain consecutive values to be placed a knight's move apart, we use a binary table constraint between each adjacent pair of elements of V. We also introduce a variable to represent the cell to which the queen is assigned, also with domain $0..(n^2-1)$. For each prime value between 2 and n^2, we introduce a 0/1 variable. A ternary table constraint ensures that this 0/1 variable is set to 1 if and only if the queen is attacking the corresponding value. We maximise the sum of these 0/1 variables. Experiments for the table constraint were performed under Linux as described earlier. Results are shown in Table 4. They show that the watched table constraint is faster than the dynamic one: i.e. the overhead of additional search are less than the overhead of restoring dynamic data structures. The limited range of experiments, and especially the lack of comparison against other techniques, means that we cannot draw extensive conclusions. We do at least conclude that watched literals provide a realistic way to implement GAC-table while being relatively straightforward.

6 Implementing Watched Literals in Minion

In this section we report briefly on how we provide the infrastructure for watched literals in Minion. This common infrastructure is used by each of the propaga-

GAC Propagators for Table Constraint $\langle X_1, X_2, \ldots, X_n \rangle \in Table$

Global Variables: $tupleList, Last$

```
Setup(inputTupleList, Vars)
A01       tupleList = inputTupleList
A02       foreach v ∈ vars
A02.1         foreach i ∈ Dom(v)
A02.1.1           Last(v,i) = tupleList[0]
A02.1.2           if SUPPORTED(Last(v,i))
A02.1.2.1             Last(v,i) = FindNextSupportingTuple(v,i,τ)
A02.2             if Last(v,i) ≠ nil
A02.2.1               REMOVEFROMDOMAIN(v,i)
A02.3             else
A02.3.1               foreach v' ∈ vars
A02.3.1.1                 ATTACHNEWTRIGGERTO(v', Last(v',i))
```

```
SupportingTupleLost(i,j)
B00       Triggered by DOMAINREMOVALOF some X_k = l in Last(X_i,j)
                                  // X_i = j was supported by the tuple Last(X_i,j)
                                  // We must find new supporting tuple, or set X_i ≠ j
B01       τ = FindNextSupportingTuple(i,j,Last(X_i,j));
B02       if τ ≠ nil
B02.1     then
B02.2         for k = 1 to n
B02.2.1           MOVEWATCHFROM Last(X_i,j)[k] TO τ[k]
B02.3         Last(X_i,j) = τ
B02.4     else                            // We failed to find a new support so X_i ≠ j
B02.5         REMOVEFROMDOMAIN(X_i,j)
```

```
FindNextSupportingTuple(i,j,τ)
C01       if(check = τ+1; check < sizeof(TupleList); check = check+1)
C02.1         if(SUPPORTED(tupleList[check]))
C02.1.1           return tupleList[check]
C03       if(check = 0; check < τ; check = check+1)
C04.1         if(SUPPORTED(tupleList[check]))
C04.1.1           return tupleList[check]
C05       return nil
```

Fig. 3. Propagator for the Table constraint. We write $\tau[k]$ for the variable-value pair at position k in the tuple.

tors reported in this paper and is available for future propagators to be implemented. Since the intention is to use watched literals to make propagation and search faster in practical constraint solvers, it is important that implementation is done in a space and time-efficient manner.

By incorporating watched triggers into MINION for our implementation and experiments, we ensure that the speed gains we have reported are against a state-

	Dynamic			Watched	
Problem	Time(s)	Nodes	Nodes/s	Time(s)	Nodes/s
4	0.21	3,373	16,062	0.1	33,730
5	0.17	954	5,612	0.07	13,629
6	69.94	268,113	3,833	21.01	12,761
7	7,146.25	10,354,130	1,449	4637.11	2,233

Table 4. Comparison of propagators for the table constraint in Minion on the Prime Queen Attacking problem. For size 7, the problem was not solved to optimality in reasonable time. The time given is to reach the same sub-optimal value.

of-the-art solver. Our implementation is not as highly optimised as the rest of MINION, but respects two primary goals. These are: that the maintenance of watched literals requires constant space after initialisation; and that key data access and update operations are fast. The key operations are finding the location of literals being watched when a value is removed, and changing which literals are being watched. We also provide infrastructure for dynamic (non-stable) triggers, but do not discuss it further as it is almost identical except that triggers are stored in backtrackable memory.

In MINION, watched literals can be created and destroyed at any time during search, but each constraint has to declare at initialisation the maximum number of watched literals it will need at any one time. Each watched trigger is associated with a particular variable and constraint and also with a unique identifier within its constraint, while a watched literal also stores value being watched. The constraint is responsible for maintaining any other information it needs for the trigger. For example the table constraint we describe below requires the current supporting tuple associated with a watched literal: this is stored in an array indexed by the trigger identifier.

A watched trigger consists of four values. These are: the small piece of space mentioned previously to allow constraints to identify the trigger; a pointer to the constraint associated with this trigger; and two pointers which are used to splice the trigger in and out of doubly linked lists. Once search begins, no trigger is ever moved or copied to another place in memory. Instead the pointers are used to change which list the trigger is in. Every literal in the CSP has a doubly linked list which contains the list of watches currently attached to it, while each variable also has a list for watched triggers for domain, bounds, and assignment triggers. When a literal is deleted, the solver moves through this list, triggering each constraint in turn. Thus, no work is done for these literals with no watches currently assocaiated with them, unlike in a traditional solver where each constraint the literal is in would have to be notified. This is one of the key features behind watched literals. When a constraint moves a watched trigger, we access the trigger's location and in the general case execute `trigger.next.prev = trigger.prev` and `trigger.prev.next = trigger.next`, the standard way to splice an element out of a doubly linked list. Thus a watched trigger movement is achieved in $O(1)$, and the space used by the trigger is free to be reused for its new location, with its pointers updated accordingly. Constraints are free to move their watched literals from watching one literal to a different one. This needs to be a

fast operation, so we need random access to their location in the list associated with a variable value pair. This again is achieved by a pointer.

There are further complications. First, constraints can leave triggers on variable-value pairs which have been removed, so we cannot just assume the trigger list will be emptied. Second, while a list of triggers is being processed, constraints may delete or move some of the triggers on it, or even move other triggers from other literals onto this list. This complicates the process of propagating all constraints attached to a literal, as the obvious methods have problems when triggers are removed from the list while the constraints are being triggered. While we raise the issue as an important implementation issue, we do not discuss our solution in detail as being of too low a level to be of general interest.

Our infrastructure for dynamic and watched triggers will make it straightforward to adapt MINION to add constraints dynamically. A new constraint set up with dynamic triggers will automatically be retracted on backtracking past it, while a constraint using watched triggers will automatically persist for the rest of search. Enabling this in MINION requires further new infrastructure, but will enable techniques such as learning nogoods during search, another technique that has proved vital in SAT.

7 Further Work and Conclusions

We have demonstrated the utility of watched literals in constraint solving. In particular, we have shown how three propagators, Sum of Booleans, Element, and Table can benefit from their use. It is important to emphasise, however, that watched literals do not render classical propagation triggering mechanisms useless. Classical triggers have a lower overhead than watched literals and so are more efficient when their use is appropriate. Many are still used in Minion.

A natural and important piece of future work is to explore the integration of nogood learning into Minion. Learning is also a crucial component of a modern SAT solver, and there is every reason to believe that it can also be of great benefit to constraint solving. Minion's ability to manage large numbers of nogoods (essentially table constraints) efficiently is clearly a substantial advantage in pursuing this goal.

References

1. C. Bessière and J.C. Régin. Arc consistency for general constraint networks: Preliminary results. *IJCAI*, 398–404, 1997.
2. Christian Bessière. Arc-consistency and arc-consistency again. *AIJ*, 65(1):179–190, 1994.
3. Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. Using inference to reduce arc consistency computation. *IJCAI*, 592–599, 1995.
4. Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *AIJ*, 165(2):165–185, 2005.
5. Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. *DAC*, 830–835. ACM, 2003.
6. Niklas Eén and Niklas Sörensson. An extensible sat-solver. *SAT*, 502–518, 2003.
7. I.P. Gent, C.A. Jefferson, and I. Miguel. Minion: Lean, fast constraint solving. *ECAI*, 2006.

8. Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity: An experience with ai and or techniques. *AAAI*, 660–664, 1988.
9. Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. *SAT*, 283–292, 2000.
10. ILOG S.A. *ILOG Solver 4.3 User's Manual*. ILOG, 1998.
11. Francoise Laburthe. CHOCO: implementing a CP kernel. In *Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, 2000.
12. M. Moskewicz, C. Madigan, Y. Zhao, S. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. *DAC*, 2001.
13. Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. *CP*, 619–633, 2004.