Journal of Artificial Intelligence Research 48 (2013) 231 – 252 Submitted 4/2013; published 10/2013

Optimal Implementation of Watched Literals and More General Techniques

Ian P. Gent IAN.GENT@ST-ANDREWS.AC.UK
School of Computer Science, St Andrews University
St Andrews, Fife KY16 9SX, UK

Abstract

I prove that an implementation technique for scanning lists in backtracking search algorithms is optimal. The result applies to a simple general framework, which I present: applications include watched literal unit propagation in SAT and a number of examples in constraint satisfaction. Techniques like watched literals are known to be highly space efficient and effective in practice. When implemented in the 'circular' approach described here, these techniques also have optimal run time per branch in big-O terms when amortized across a search tree. This also applies when multiple list elements must be found. The constant factor overhead of the worst case is only 2. Replacing the existing non-optimal implementation of unit propagation in MiniSat speeds up propagation by 29%, though this is not enough to improve overall run time significantly.

1. Introduction

In many backtrack search procedures, a given list must contain an element satisfying some property. I call this an "acceptable element". If no acceptable element exists in the list, some relevant action is triggered, such as assigning a unit clause in SAT (Boolean Satisfiability). This paper considers only monotonic acceptability properties: i.e. if an element is unacceptable at a node in the search tree, it remains unacceptable at all descendant nodes. This description is general, but applies to vital components in modern search techniques. As well as unit propagation in SAT, a typical application in CSP (Constraint Satisfaction Problem) is support for a variable-value pair in a general purpose arc consistency algorithm, e.g. MGAC2001/3.1 (Bessière, Régin, Yap, & Zhang, 2005). Once no acceptable element (support) is available, the triggered action is that the variable-value pair must be pruned.

A common technique for maintaining acceptability is to keep a pointer to a known acceptable element. If this element becomes unacceptable during the search process, we scan the list for a new acceptable element. If we find one we change the pointer. If not, we can trigger the necessary action. After backtracking, we must guarantee that the pointer points to an acceptable element. There are several methods for achieving this. One is to reset the pointer before each list scan. A second is to store the current value of the pointer and to restore this value on backtracking. This paper focuses on a third method, the "backtrack-stable" approach, as used in watched literals in SAT.

The backtrack-stable approach is very simple: no changes are made to the pointer except when we are scanning the list for a new acceptable element. Watched literals in SAT, the classic example of this approach, have proven to be highly effective in practice (Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001). Its correctness depends on acceptability being monotonic. If we move the pointer to a new acceptable element at a particular node, the new element must be acceptable at all ancestor nodes. When we backtrack we thus still have an acceptable element, even if it is different to the one we entered the node with. Unfortunately, down any branch (i.e. sequence of nodes from root to leaf node), we can no longer guarantee that a single pass of the list is enough. At a given node the pointer may move due to moves at children nodes, meaning we may have missed elements. Indeed, on a single branch we may need to check every value of the list many times.

This paper shows that a simple, 'circular', implementation of the backtrack-stable approach is optimal in big-O terms when amortised across all branches of a tree, a significantly better theoretical property than has previously been suggested. The amortisation applies to any search tree explored in a depth-first style, independent of the size of the search tree. The constant increase in worst case complexity is only 2. Applicability to existing code depends on low-level implementation details. I discuss some such cases in detail. In some cases existing implementations are shown to be optimal, while in others an optimal implementation of watched literals was not used. For example I show empirically that implementing watched literals optimally speeds up unit propagation in MiniSat by 29%, although the effect does not lead to a statistically significant improvement in overall MiniSat solution time.

Section 2 describes a simple framework for scanning lists in backtracking search and gives pseudocode for the three methods compared in this paper, and gives a worked example to motivate the proof that the circular approach is optimal. Section 3 gives that proof and related results, including detailed comparisons with the state restoration method. Section 4 generalises the result to multiple acceptable elements. Later sections discuss applications to constraint satisfaction, and to watched literal unit propagation in SAT, including experiments on MiniSat. Appendices gives proofs omitted from the main text, detailed methodology of the SAT experiments, results on another method of list scanning, and a summary of the Online Appendix for this paper.

2. Simple Framework for List Scanning Algorithms

We have a list LIST of length N.[1] We assume there is a boolean function ACCEPTABLE(LIST,i) to check list elements, returning true if LIST[i] is acceptable and false if not. Throughout this paper, this function is required to be monotonic, in the following sense.

Definition 1 (Monotonicity of Acceptability)**.** *Acceptability is* monotonic *if, whenever* ACCEPTABLE(LIST,i) *fails at a node, then* ACCEPTABLE(LIST,i) *would also fail if called for the remainder of the search process at that node and at all its descendant nodes.*

Note that this definition allows elements in LIST to be moved, as long as unacceptability at each index is maintained: this flexibility will be important in Section 4. When unacceptability is detected we must find a new acceptable element, or guarantee that none exists. This is achieved by one of three variants of a function FINDNEWELEMENT (sometimes abbreviated to FNE). Each variant updates the value of a pointer *last*, and either succeeds with an acceptable value of LIST[*last*], or fails with a guarantee that no acceptable value in LIST exists. The execution environment is assumed to guarantee that, at any

1. In this paper I assume a computational model in which the word size is at least $\log N$ bits, and that standard operations on words take $O(1)$ time. This is not ideal from a complexity-theoretic point of view, but is a standard assumption in the literature, although unfortunately not usually clearly stated.

node of the search tree, if list[*last*] changes from being acceptable to unacceptable, then findNewElement(list) will be called before the next node is visited, *except* when backtracking occurs from this node before any descendant node is visited. For the purposes of this paper, detection of unacceptability is *not* counted as a call to acceptable: whatever cost this detection has is the same for each approach studied here. We assume an initialisation phase in which $O(N)$ calls to acceptable are made to find the initial value of *last*: these calls are considered as preprocessing and not charged to any node in the search tree. We also assume that calls to findNewElement are only made during initialisation and after unacceptability has been detected.

Each variant of findNewElement has to ensure the following invariant is maintained. Note that this invariant means when one of the first two cases holds, the value list[*last*] is unacceptable iff there is no acceptable element in the list.

Invariant 2. *At all times at least one of the following is true:*

- *the value* list[*last*] *is acceptable; or*
- *there is* no *acceptable element in* list*; or*
- *the initialisation process has not completed; or*
- *the value* list[*last*] *has become unacceptable at the current node but* findNewElement *has not yet been called and completed.*

Many calls to findNewElement may happen at a single node, because facts about acceptability may not become known at the same time. For example, in the case of unit propagations in SAT, the current watched literal (i.e. list[*last*]) may become unsatisfiable (unacceptable), so we have to scan for a new watched literal (value of *last*). But later propagations at the same node can make this new value unsatisfiable, leading to new scans.

This paper compares three implementations of findNewElement and shows the good properties of the last one. The differences arise from how *last* is dealt with in between invocations of findNewElement. The simplest variant is shown in Procedure 1: each time it is called the previous value of *last* is discarded and the list searched from the beginning.

Procedure 1: FNE-NoState(list)

```
last := -1
repeat
    last := last + 1
    if acceptable(list,last) then return true
until last = N
return false
```

Procedure 1 is simple, certainly maintains Invariant 2, and is highly space-efficient. It has a significant disadvantage in that its worst case is to require $\Theta(N^2)$ calls to acceptable at every leaf node, stated here as Proposition 3 and proved in Appendix E. (Appendix E is omitted from the main text but is available online, see Appendix D.)

Proposition 3. *The procedure* FNE-NoState *maintains Invariant 2. It makes* $O(N^2)$ *calls to* acceptable *per branch of a search tree, but requires* $\Theta(N^2)$ *in the worst case.* (Proof in Appendix E online.)

The remaining two variants both use FNE-NoState to initialise *last*, but make different calls thereafter. The second variant is based on state restoration. Each call continues search from the most recent value of *last* found at the current node or any of its ancestors. Since the value of *last* may be changed by descendent nodes, some mechanism of the backtracking search solver must be exploited to restore the value of *last* when backtracking occurs. What this method is will vary between solvers, and is not critical for this paper. Given this, this method is shown as Procedure 2.

Procedure 2: FNE-RestoreState(list)

```
repeat
   last := last + 1
   if ACCEPTABLE(LIST,last) then return TRUE
until last = N
return FALSE
```

Since acceptability is monotonic and *last* is always restored to the value it had previously, the invariant is guaranteed. We have the following, for which no proof should be necessary.

Proposition 4. *The procedure* FNE-RestoreState *maintains Invariant 2. On any given branch of the search tree from root to leaf node, at most* N *calls to* acceptable *are made.*

In the final, "backtrack-stable", variant, we do not restore former values of *last* on backtracking. We now have the problem that down a branch, or even at a single node, the value of *last* can be moved by later nodes and not restored when we return to the current node. To deal with this correctly, we have to allow every element of list to be checked even if some may have already been checked higher on the current branch or even at the same node. The focus of this paper is the "circular" method for checking all elements: at the end of the list we circle around from the end of the list to zero, and continue checking until we hit the value that *last* had when the call was made. For initialisation we call the above procedure FNE-NoState. All subsequent calls are to the following.

Procedure 3: FNE-Circular(list)

```
last-cache := last
repeat
   last := last + 1
   if last = N then last := 0
   if ACCEPTABLE(LIST,last) then return TRUE
until last = last-cache
return FALSE
```

No claim is made for originality of the circular method: it was used by Gent, Jefferson, and Miguel (2006b) and probably by earlier authors. Unlike the previous variants, the invariant does not hold for all search algorithms. Correctness will be proved in Section 3 below, in the context of 'downwards-explored search trees', as defined in Definition 5.

.

One difference between Procedures 2 and 3 should be noted. At a given node of the search tree, Procedure 2 can make at most N calls to ACCEPTABLE. However, Procedure 3 can, perhaps counterintuitively, require almost $2N$ calls at a single node. For example, suppose at a given node we have $last = 0$. If this value becomes unacceptable and the only other acceptable value is $N - 1$, this requires $N - 1$ calls to set $last = N - 1$. Further propagation may later make $N - 1$ unacceptable also. The resulting new call to Procedure 3 cannot find an acceptable value, but must still check all values from 0 to $N - 2$ as it has no memory of the previous call. (The variable *last-cache* is local to each call of the Procedure.) This is a further $N - 1$ calls to ACCEPTABLE, for a total of $2N - 2$ at the node.

2.1 Worked Example

I present a worked example of Procedure 3, showing how we can amortize the count of calls to ACCEPTABLE in a way which will form the basis of the optimality results of this paper.

Figure 1 shows an example search using the circular approach. In the example we assume

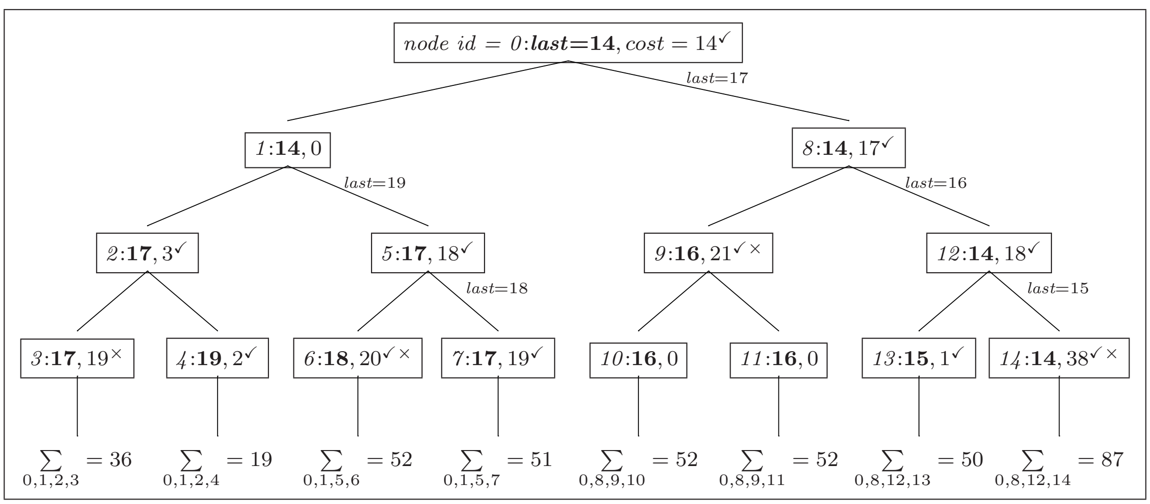

Figure 1: A search using the circular approach. See main text for description.

that the list being searched has 20 elements, indexed from 0 to 19. and that initialisation sets $last = 0$. The box at a node indicates the node number (italics), the value of *last* after any calls to FNE-CIRCULAR (bold), and the cost of those calls in total at the node (measured as number of calls to ACCEPTABLE), including a failed call if there is one. A ✓ indicates a successful call was made, while a × indicates an unsuccessful call. Both can occur at the same node, if the value of *last* is moved by a successful call, while that value later becomes unacceptable, and another call to FNE-CIRCULAR fails. A branch is annotated with a value of *last* if the search to the left changed the value of *last*. For example, at node 5 the value of *last* was 17, but the left child set it to 18, meaning that it had changed when we branched right. The total cost for each branch is listed, e.g. 87 for the last branch.

In the example, the first 14 elements become unacceptable at the root, so calls to FNE must check 14 elements in order to set $last = 14$. If we restored *last* the maximum cost down any branch would be the number of elements in the list, in this case 20. If we used FNE-Circular, but searched only one branch, the maximum cost would be almost double, 38. This could happen if we eventually settle on $last = 19$, this becomes unacceptable and we do a failed call which costs 19. By contrast, the extreme right hand branch costs $14 + 17 + 18 + 38 = 87$, more than twice the maximum if search took only a single branch. Summing the number of checks in FNE on each branch, we get $36 + 19 + 52 + 51 + 52 + 52 + 50 + 87 = 399$, almost 50 times the number of branches, 8. However, this counts many costs twice. By summing at each node, the total number of checks across the tree is $14+0+3+19+2+18+20+19+17+21+0+0+18+1+38 = 190$. This is more than 20 times the number of branches, but less than 40 times the number of branches. I will show that the latter bound is always true: we can never look at more list elements than $2kN$, if N is the number of list elements and k is the number of branches.

A *left branch segment* (LBS) is a segment of a branch with all left branching decisions ending at a leaf node. In this paper, the left child of a given internal node is taken to mean whichever child node is explored first.[2] Figure 2 shows the same search tree as Figure 1, but showing the LBSs and the cost across each LBS. Crucial points to note are: each left branch segment contains consecutively numbered nodes; the cost of each left branch segment is never more than $38 = 2N - 2$; and the total cost summed over left branch segments $(36 + 2 + 38 + 19 + 38 + 0 + 19 + 38 = 190)$ is identical to the total cost across the tree as calculated earlier. All of these observations about the tree are provable in general.

3. Formal Results

While the worked example showed binary branching, this is not necessary. After the left hand child of a node there may be any number of later nodes (including zero at a unary node.) A leaf node is simply a node with no children. I will prove correctness and optimality of FNE-Circular when search algorithms build trees of the following type:

Definition 5 (Downwards-Explored Search Tree)**.** *A search tree is* Downwards-Explored *if: for any node n_0 in the tree, all nodes descending from n_0 are visited later than n_0,* and *for any node n_1 not descended from n_0 and visited later than n_0, all nodes in the tree that descend from n_0 are visited after n_0 but before n_1.*

Depth-first search certainly defines a downwards-explored tree, but so do many variants such as conflict-directed backjumping (Prosser, 1993). A less obvious case is a single restart of a modern Conflict-Driven Clause Learning (CDCL) SAT solver (Marques-Silva, Lynce, & Malik, 2009). When backtracking, search returns to the level of the tree chosen by conflict-analysis, and it is guaranteed that all intermediate parts of the search tree are unsatisfiable, and will never be visited again, resulting in a downwards-explored search tree. CDCL solvers undertake restarts, as do other major algorithms such as Iterative Deepening (Korf, 1985) and Limited Discrepancy Search (Harvey & Ginsberg, 1995; Prosser & Unsworth, 2011). In all three of these examples each iteration considered separately gives a downward-explored tree. The results of the current paper therefore apply to a single iteration or restart of each algorithm. They will also apply to multiple restarts or iterations if one counts separately

2. An algorithm may regard its first branching choice as the right branch, e.g. limited discrepancy search going against its heuristic, but for this paper the left child is defined as whichever one is explored first.

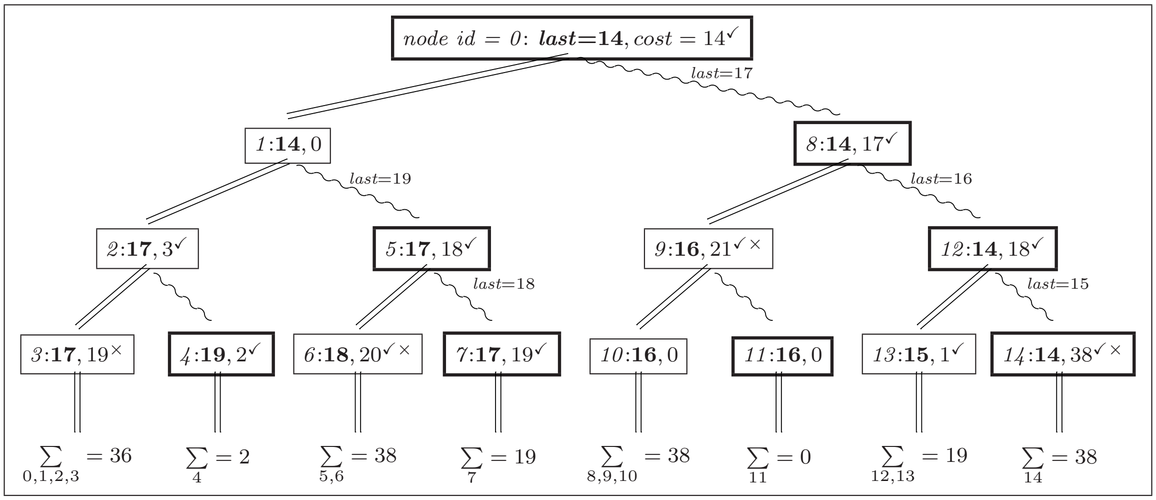

Figure 2: The example from Figure 1 with additional annotations. A double line indicates a left hand branch, while a wavy line the right hand branch. A sequence of double lines indicates a left branch segment. A solid box around a node indicates the start of a left branch segment. Finally, the sums of costs are made only over the left branch segment ending at each leaf node.

for each iteration, i.e. counting a branch twice even if it is a duplicate of a branch from a previous iteration. Some major algorithms do not explore downwards: e.g. breadth-first or best-first search. In such algorithms, at consecutive nodes *last*, which is not contained in the search state, moves from pointing to an acceptable to an unacceptable value. The results of this paper therefore do not apply in any way to such algorithms.

The first result is correctness of the FNE-circular method, slightly generalised so that it will also apply to the "middle-out" Procedure 5 in Appendix B.

Definition 6 (Locally Correct)**.** *A* findNewElement *procedure is* locally correct *iff:*

1. *if any element of* list *is acceptable then the procedure sets* last *to point at an acceptable value and succeeds;*

2. *if no element of* list *is acceptable then the procedure fails and exits with* last *set to the same value it had on entry.*

Specifically, FNE-Circular is locally correct by design, and therefore its global correctness is a corollary of the following theorem.

Theorem 7. (Correctness) *If the search algorithm defines a downwards-explored search tree, and if procedure* findNewElement *is locally correct, Invariant 2 is true at all times.* (Proof in Appendix E online.)

Definition 8 (LBS)**.** *The* ***left branch segment*** *ending at a leaf node n $LBS(n)$ is defined recursively as follows:*

- *$n \in LBS(n)$*
- *if $m_1 \in LBS(n)$ and m_1 is the left hand child of a parent node m_2, then $m_2 \in LBS(n)$. For convenience, if a parent node has only one child we call this the left hand child.*
- *$LBS(n)$ is the minimal set of nodes satisfying the above properties.*

Note that for a leaf node m which is not the left hand child of its parent, the definitions trivially give $LBS(m) = \{m\}$. Every internal node has exactly one left child, so we can proceed as follows.

Lemma 9. *Every node in the tree is contained in exactly one left branch segment. In a downwards-explored search tree, nodes in a left branch segment are visited consecutively without search visiting any other nodes.* (Proof in Appendix E online.)

Theorem 10. *In a downwards-explored search tree, there are no more than $N - 1$ calls to* ACCEPTABLE *made by successful calls to* FNE-CIRCULAR *in a left branch segment.*

Proof. The proof relies on the monotonicity of acceptability down the tree. Because all nodes in an LBS are explored sequentially without interruption, from Lemma 9, all calls to FNE-CIRCULAR and therefore changes to *last*, are consecutive. The definition of FNE-CIRCULAR means that to check more than $N - 1$ elements on a single LBS, *last* must be incremented at least N times, meaning that every value is checked at least once in the LBS, including the original value *last* at entry to LBS. Call this value i. Say the root of the LBS is m_1, and the element i is checked again at node m_2 with either $m_1 = m_2$ or m_1 an ancestor of m_2. Between entry to m_1 and the call to ACCEPTABLE(LIST, i) at m_2, every list element must have been checked. Furthermore, for each value j, either the check of j in FNE-CIRCULAR failed, or it succeeded and later on the value had become unacceptable causing another call to FNE-CIRCULAR. If this was not the case then the value of *last* would not have moved on from j. Therefore, for any j, the list element j cannot be acceptable when the check of i was made at node m_2. Therefore, the call to FNE-CIRCULAR which makes the second check of i must fail. As required, we have shown that if there are more than $N-1$ calls to ACCEPTABLE in an LBS, there is an unsuccessful call to FNE-CIRCULAR. □

Corollary 11. *In a downwards-explored search tree, calls to* FNE-CIRCULAR *in an LBS make no more than $2N - 2$ calls to* ACCEPTABLE.

Proof. By Theorem 10, the maximum number of list elements checked in successful calls in any LBS is $N - 1$. The first unsuccessful call to FNE-CIRCULAR will check $N - 1$: all elements of the list except LIST[*last*]. No more calls are necessary in the LBS, since no element can become acceptable again. So the total cost is bounded above by $2N - 2$. □

Theorem 12. *For a downwards-explored search tree containing k branches, calls to* FNE-CIRCULAR *make at most $k(2N - 2)$ calls to* ACCEPTABLE.

Proof. Any call to FNE-CIRCULAR occurs at some node in the search tree. Each node in the search tree is in exactly one LBS. Therefore every call to FNE-CIRCULAR occurs in exactly one LBS. There are at most $2N - 2$ acceptability checks in each LBS. In a tree with k branches there are exactly k LBSs, meaning that the total number of list elements checked in the tree is bounded above by $k(2N - 2)$. □

We therefore have optimality in the following sense:

Theorem 13. (Optimality) *In any downwards-explored search tree, the circular approach requires space for one* last *pointer and has a worst case of $O(N)$ calls to* ACCEPTABLE *per branch of the tree, and no algorithm can require $o(N)$ calls per branch.* (Proof in Appendix E online.)

While these results show that restoring state and circular are equivalent in worst case time complexity in big-O terms across a tree, we can compare them more precisely.

Proposition 14. *For circular, there can be as many as $2k(N-2)$ calls to* ACCEPTABLE *in a downwards-explored search tree. For state restoration, the number of calls to* ACCEPTABLE *is bounded above by kN and there can be $k(N-1)$ calls to* ACCEPTABLE *in a tree.* (Proof in Appendix E online.)

Thus we have that the worst case number of list element checks where we do not backtrack *last* is twice the worst case number of list checks where we do backtrack it. But this does not apply on an instance by instance basis, as the following result shows.

Proposition 15. (Non Dominance) *Both techniques can check less than the other across a downwards-explored search tree. The circular method can take $\Omega(k)$ times fewer calls to* ACCEPTABLE *across the tree than state restoration, while state restoration can need $\Omega(N)$ times fewer calls than circular.* (Proof in Appendix E online.)

In the language of Likitvivatanavong, Zhang, Shannon, Bowen, and Freuder (2007), in any LBS, circular can have no "positive repeats" (duplicate successful calls to ACCEPTABLE). It can have "negative repeats" (duplicate failed calls) only if the last call to FNE-Circular fails, so there will be no failed call to FNE when there is an acceptable element at a leaf node. This reduces significantly the chance of more than N calls in an LBS. For example, in SAT, consider a clause with r literals. Under a random boolean assignment, there is a $\frac{1}{2^r}$ chance of no literals being valid under a full assignment, and $\frac{r}{2^r}$ of exactly one literal being valid. With two or more valid literals, there can be no failed call. So the chance of a call to FNE failing is no more than $\frac{r+1}{2^r}$ on any LBS. If clauses have 10 literals then under random assignments there is a maximum of a just over 1% chance of negative repeats in each LBS.

4. Generalisation to Multiple Acceptable Elements

An important case is that we need to maintain multiple acceptable elements in a list. Specifically, we must ensure that at least W different elements of LIST are acceptable, or trigger some action if less than W are. The classic example is watched literals in SAT, where $W = 2$ and the action is unit propagation, but examples arise in constraints with higher W as discussed in Section 5. Not only can the circular approach be generalised, but the bound on number of calls to ACCEPTABLE per branch is independent of W.

The implementation technique is to maintain two lists. The first, called WATCHED, is of length $W-1$, and the second list, UNWATCHED, is of length N. The union of these two lists is the original list LIST of length $N+W-1$. Initialisation is assumed to either make all elements in WATCHED plus UNWATCHED[*last*] acceptable, or (if that is not possible) to trigger the necessary action. The solver infrastructure is assumed to ensure correct notification of findNewElement: we must maintain the position of each element in watched, but this can be done in $O(1)$ time per move of an element using $O(N + W)$ space. We now assume that if any element in watched changes from being acceptable to unacceptable, or the single value unwatched[*last*] does, then FNE-Circular-W will be called with the appropriate parameters before the next node is visited, unless backtracking occurs before then. We also assume that if more than one such event happens at a node, a separate call happens for each event. Given these assumptions, implementation is almost trivial, as follows.

Procedure 4: FNE-Circular-W(list,*elt*,*i*)

```
1:                    // elt is the value in LIST which is newly unacceptable
2:            // i is the index of elt in WATCHED unless elt = UNWATCHED[last]
3: if elt ≠ watched[i] then
4:    watched[i] := unwatched[last]
5:    unwatched[last] := elt
6: return FNE-Circular(unwatched)
```

Everything that follows depends on the fact that, despite swapping elements, acceptability is monotonic in the list unwatched.

Proposition 16. *Acceptability in* unwatched *is monotonic if acceptability in* list *is.*

Proof. Because acceptability in list is monotonic, the only way nonmonotonicity could occur is when an element of unwatched is replaced with one from watched. This can only happen in FNE-Circular-W at Line 5. But the value unwatched[*last*] is replaced by *elt*, and *elt* becoming unacceptable is the reason that FNE-Circular-W was called. Therefore, a call to acceptable(list,*last*) must return false, whether or not it would have succeeded before the replacement. By Definition 1, monotonicity is therefore respected. □

Proposition 16 and Corollary 11 make the following immediate. It is remarkable that the bound in Corollary 17 is independent of W, the number of acceptable elements required. Initialisation does need $O(N + W)$ calls to acceptable, but this is done only once.

Corollary 17. *In a downwards-explored search tree, calls to* FNE-Circular-W *in an LBS make no more than* $2N - 2$ *calls to* acceptable.

We must also show correctness, and this is done with a revised invariant. If the first clause holds, we have the W required acceptable elements, while if the second does, there can be no more than $W - 1$ acceptable elements so we can trigger the necessary action.

Invariant 18. *At all times at least one of the following is true:*

- *all elements in* watched $\cup$ {unwatched[*last*]} *are acceptable; or*
- *there is* no *acceptable element in* unwatched*; or*
- *the initialisation process has not completed; or*
- *at the current node, at least one element in* watched$\cup${unwatched[*last*]} *has become unacceptable but the corresponding call to* findNewElement *has not yet completed.*

Lemma 19. *In a downwards-explored search tree, each call to* FNE-Circular-W *either returns false or reduces by one the number of unacceptable elements in the set* watched ∪ {unwatched[*last*]}*.* (Proof in Appendix E online.)

Theorem 20. (Correctness) *In a downwards-explored search tree,* FNE-Circular-W *maintains Invariant 18.* (Proof in Appendix E online.)

Chai and Kuehlmann (2003) described multiple watches for a pseudo-boolean solver, although the current results do not apply because the number of watches varied during search. Chai and Kuehlmann did not give implementation details: the implementation described here follows Gent et al. (2006b) for a sum of boolean variables.

5. Application to Constraint Satisfaction

The first application is in constraint propagation, specifically maintaining generalised arc consistency. The optimal algorithm GAC2001/3.1 can easily be turned into the algorithm MGAC2001/3.1 which maintains GAC during search (Bessière et al., 2005).

Corollary 21. *For a constraint of arity r where each variable is domain size d, in a downwards-explored search tree the circular approach to maintaining the last pointer in MGAC2001/3.1 can be achieved using space to store $O(dr)$* last *pointers (beyond the storage space for the constraint itself), and requires time to check $O(rd^r)$ tuples per branch.* (Proof in Appendix E online.)

A reasonable assumption is that it takes time $O(r)$ to check a tuple since it is arity r (Bessière et al., 2005).[3] On this basis we get time $O(r^2d^r)$ per branch. Bessière et al. report the same time complexity $O(r^2d^r)$ for GAC2001/3.1 and require the same number of *last* pointers. This shows that the amortized worst case big-O time per branch for MGAC2001/3.1 is the same as that needed simply for the one-off algorithm GAC2001/3.1, and using the same space. Bessiere (2004) reports using state restoration techniques in his implementation of MAC2001 (Bessière & Régin, 2001). This therefore used additional space, since many copies of *last* must be stored instead of just one.

My results improve on those given by van Dongen (2004). For binary constraints ($r = 2$), he gives an upper bound for space complexity of $O(d \min(n, d))$ per constraint for a time optimal implementation using time $O(d^2)$ per branch, where d is domain size and n is the number of variables in a problem. Corollary 21 gives the same $O(d^2)$ time and improved $O(d)$ space, although van Dongen's results remain valid as they were given as upper bounds.

There are several studies of the time complexity of maintaining arc consistency down a branch, but no suggestion that leaving *last* pointers alone can be so good theoretically. Some existing circular implementations can now be seen to be optimal. For example, Gent et al. (2006b, p. 185) wrote: "There is one general disadvantage that should be mentioned with watched triggers, …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

3. If constraints are stored extensionally, then this will be true, and space requirement to store the constraint will be $O(rd^r)$. However constraints may also be stored intensionally or procedurally, in which case checking time can be either larger or smaller, and space requirement can be arbitrarily small.

example is our variant of GAC-2001/3.1 below." In fact, this implementation of MGAC-2001/3.1 was time optimal amortized across branches and had better space complexity than van Dongen (2004) reports for optimal implementations of MAC-2001/3.1.

Régin (2005) studied maintaining arc consistency during search without backtracking the last pointer. He writes: "If the last values are not restored after backtracking then the time complexity of AC-6 and AC-7 algorithms is in $O(d^3)$" (Régin, 2005, p. 528), and he gives an example similar to the right hand branch of Figure 1. In this context d is domain size and constraints are binary so the maximum length of list $N = d^2$. Régin is entirely correct. My contribution is to show that the lack of optimality down a branch is compensated by amortization by a factor of d. Régin writes also: "Currently [i.e. before his paper], there is no MAC version of these algorithms capable to keep the optimal time complexity on every branch of the tree search ($O(d^2)$ per constraint, where d is the size of the largest domain), without sacrificing the space complexity". His method can recompute a correct value of *last* on backtracking without storing it, by comparing the current value of *last* with values restored to domain on backtracking. This is elegant but does not generalise in an obvious way to non-binary constraints as the number of combinations of restored values can be exponential.

Likitvivatanavong et al. (2007) discuss the cost of Arc Consistency during search. They give ACS-resOpt, which uses an instance of the list scanning framework given here, and report that while it is optimal at a given node, it is not optimal down the branch of a tree (which they call "path-forward complexity"). By reordering domains during search, "Adaptive Domain Ordering" (ADO) enforces MAC for binary constraints with the optimal property of $O(ed^2)$ worst-case time complexity on any branch of the search tree (Likitvivatanavong, Zhang, Bowen, & Freuder, 2005). Unfortunately it did not perform well in empirical tests (Likitvivatanavong et al., 2007), and combined with its reordering of domains during search, this militates against widespread adoption in constraint solvers.

There are other applications in constraints. Nightingale, Gent, Jefferson, and Miguel (2013) use the circular technique to avoid restoring state in GAC algorithms exploiting short supports. Jefferson, Moore, Nightingale, and Petrie (2010) use it for a propagator for a generalised 'or' constraint. Gent et al. (2006b) used the circular approach for propagating the element constraint. They also used the generalisation to W literals for a sum of booleans constraint. All these examples can be seen in the code for the Minion constraint solver (Gent, Jefferson, & Miguel, 2006a), version 0.15, as can some not described in papers such as the constraint 'litsumgeq'. All are now seen to have excellent theoretical properties.

6. Application to Satisfiability

The second application is watched literal unit propagation in SAT (Moskewicz et al., 2001). A clause is a list of literals. An acceptable element is one that represents either an unassigned or a satisfied literal. In standard two-literal watching we must maintain two acceptable elements. Modern CDCL SAT solvers quickly learn large numbers of large clauses, and thus benefit greatly from having to maintain only two pointers in a clause. The approach of Section 4 can be applied easily. However, this is not normally done in SAT solvers.[4] Successful solvers often implement the search for new watches in a non-optimal way. To see this, we

4. I am grateful to a reviewer of this paper for pointing this fact out to me.

must examine their code, because the necessary level of implementation detail is not given in papers. For example MiniSat (Eén & Sörensson, 2003) and tinisat (Huang, 2007) implement watched literals in the following non-optimal way. The core of the implementation is shown as Procedure 5: a key change is that all watched elements are in the list WATCHED instead of all but one in Procedure 4. Some details of MiniSat's implementation are not in Procedure 5: the most important is the maintenance of "blocked" literals, which I will discuss further below. Proposition 3 does not technically apply, but nevertheless I state without proof that this approach leads to $\Omega(N^2)$ calls to ACCEPTABLE per branch in the worst case.

Procedure 5: FNE-NoState-W(LIST,*elt*,*i*)

```
                    // elt is the value in LIST which is newly unacceptable
                                        // i is the index of elt in WATCHED
result = FNE-NoState(UNWATCHED)
if result then
   WATCHED[i] := UNWATCHED[last]
   UNWATCHED[last] := elt
return result
```

To compare practical performance, I adapted MiniSat version 2.2.0 with optimal unit propagation using Procedure 4. I call this "Circular" MiniSat and the original "Stock" MiniSat. Appendix A gives full methodology and more detailed results. The key results are as follows. First, Circular unit propagates notably faster than Stock when features of MiniSat not related to unit propagation were removed. For searching 594 instances up to 10 million conflicts, Circular's mean time was 141.6s, compared to a mean 182.6s for Stock, so Stock takes 29.1% more time. Median performance is much closer, 73.1s for Circular compared to 78.1s for Stock. The larger disparity in mean is because Stock is never more than 15% faster, but Circular can be as much as 9.5 *times* faster than Stock. Second, when all features of MiniSat were restored, the improved propagation speed of Circular did not translate into improved performance. On 330 instances, there was no statistical support to reject the null hypothesis that the two solvers have equivalent performance.

It is interesting to look at how many watched literal scans are 'blocked', a term used in MiniSat's code. A clause scan is 'blocked' if the other watched literal (at the time this watch was set up) is a valid literal now, so the clause is satisfied and no new watch is needed. This saves accessing the memory relating to the clause. For each instance I measured the ratio b/u between blocked and unblocked watched scans. A higher b/u is better since it results in less watched scans. I computed the ratio ρ between the b/u values obtained for Circular and Stock MiniSat, $\rho = (b_c/u_c)/(b_s/u_s)$. Figure 3 shows ρ plotted against obtained speedup. Behaviour is very different in two regions. For $\rho \geq 1.2$ (123 instances) the median speedup is 1.48 and mean is 1.86. There is a high correlation between ρ and speedup, $r^2 = 0.88$. For $\rho < 1.2$ (471 instances) we get no correlation between ρ and speedup, $r^2 = 0.01$. Compared to Stock MiniSat the median speedup is 0.97 and mean of 0.98, i.e. slight slowdowns. This analysis indicates that most of the speedup occurs in instances where Circular is much better than Stock MiniSat at setting watches on literals which are likely to be valid later in search and thus block watched literal scans. It would be interesting to investigate this result theoretically, since it does not follow from the results of this paper. It may be related

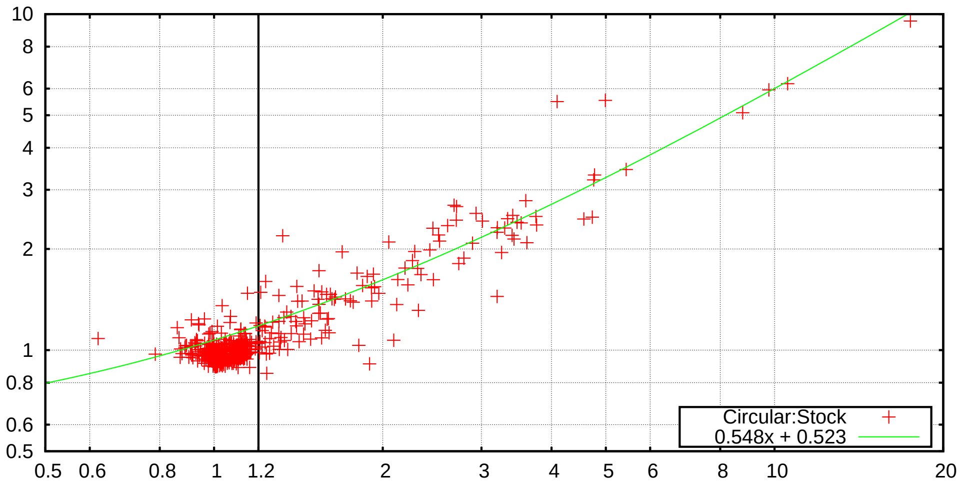

Figure 3: Scatterplot of ρ (x-axis) against speedup ratio in conflicts per second of Circular over Stock MiniSat (y-axis). The vertical line shows $\rho = 1.2$. The line $0.548x + 0.523$ is the best-fit line in the region $\rho \geq 1.2$.

to the fact that Circular sets watches to arbitrary literals in a clause, while Stock MiniSat will tend to set watches to literals appearing early in a clause. Therefore if a good blocking literal is late in a clause, Circular has more chance of watching it.

Head-tail lists were an important advance in implementation of unit propagation in SAT (Zhang & Stickel, 2000). Pointers to the first unassigned literal in a clause (head) and the last (tail) are maintained by state-restoration. Head-tail lists led to "watched literals" (Moskewicz et al., 2001), in which there are two pointers to arbitrary (but different) unassigned literals. Watched literals (or variants thereof) have become the standard technique for efficient implementation of unit propagation in SAT solvers. When implemented as described in this paper, the theoretical properties of watched literals are now seen to be very nearly as good as head-tail lists in time, with much reduced space overheads.

A variant implementation of watched literals is in JQuest (Lynce & Marques-Silva, 2005). Scans go from *last* to the end of the list, but restart from *last* backwards to 0. This "middle-out" search can be big-O optimal *provided* that the current direction of search is persistent between calls. However, if the direction of search is always initially in one direction, it can require $\Omega(N^2)$ checks per branch, and this is the case in JQuest. Full algorithmic details and proofs of these statements are in Appendix B. Lynce and Marques-Silva (2005) also introduced literal sifting to attempt to get the best of both worlds: literals in a clause are reordered during search to avoid repeating checks and backtracking pointers. Lynce and Marques-Silva report slightly better empirical performance with literal sifting (although as just noted the comparison was with a non-optimal watched literal implementation). Generalising literal sifting to arbitrary number of acceptable elements and performing further theoretical and experimental comparisons with the circular approach is open as future work.

Another application which is now seen to be optimal, if implemented in a circular style, is Van Gelder's (2002) three literal watching to detect binary clauses.

7. Conclusions

I have shown that the circular approach to scanning lists in backtracking search has desirable theoretical properties. It is big-O optimal in time (measured as number of acceptability checks) when amortized across a search tree. The worst case constant is only a factor of two. The results are not an average case but apply to every search tree with any number of branches. The results generalise to maintaining multiple acceptable elements in a single list, and the complexity is independent of the number of elements required. This result is relevant to practically important algorithms in applications such as SAT and CSP. Techniques like watched literals in SAT are known to be successful in practice, and certainly have reduced space overheads compared to state restoration methods. When implemented appropriately, they can be newly understood to have essentially no theoretical disadvantages in time either. Some existing implementations are now seen to be optimal even though this was not realised by their implementers. Some implementations of unit propagations in real world SAT solvers are not optimal, e.g. MiniSat. Replacing with an optimal implementation in MiniSat can improve propagation speed by a mean of 29%. Experiments suggested that the circular approach was better able to find watched literals likely to be true at future nodes. However, improved propagation speed did not result in improved speed in the full solver.

Acknowledgments

I thank Chris Jefferson and Peter Nightingale for help with this paper in many ways, for example C++ coding advice and suggestions on how to implement variants of watched literals in MiniSat. I thank the JAIR editor of this paper, Holger Hoos, and anonymous reviewers for suggestions leading to the comparison with MiniSat and study of the "middle-out" approach, and for requiring much more precise presentation of my results. I thank authors of MiniSat, tinisat, and JQuest for making their code available for study.

Appendix A. Experiments in MiniSat

This appendix describes methodology and gives detailed results for experiments on watched literal implementation in MiniSat version 2.2.0. Two variants of MiniSat were implemented. The first, Circular, implements two literal watching algorithm using the approach described in Section 4. The second, TwoPointer, is a variant in which two independent pointers are maintained, based on an method in a preprint of this paper. Since it has worse properties both in practice and theory, TwoPointer is described further only in Appendix C.

Timings reported here were performed on a single Apple MacPro (MacPro4,1), with two Quad-Core Intel Xeon chips 2.26GHz, L2 Cache 256KB per core, L3 Cache 8MB per processor, 32GB DDR3 RAM 1066MHz, 7200 RPM hard drive, MacOS 10.6. MiniSat 2.2.0 was used as the codebase, with compile time flags as in the distribution. Instances from the

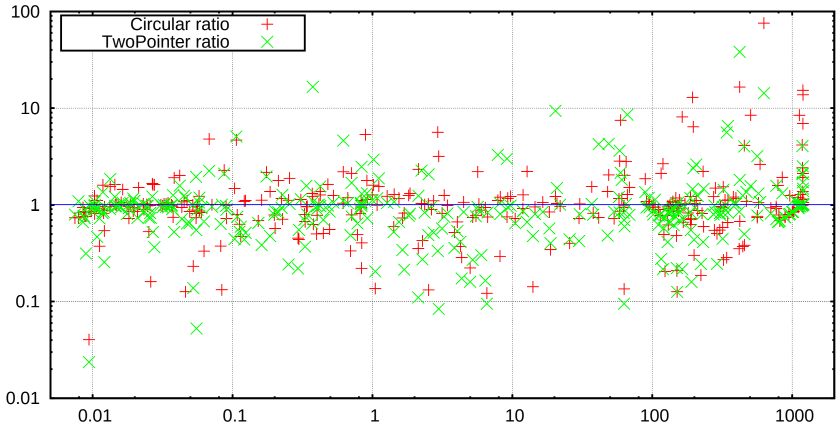

Figure 4: Scatterplot of relative performance of Circular and TwoPointer variants compared with Stock MiniSat. The x-axis gives run time of Stock MiniSat in seconds. The y-axis gives ratio of Stock time to Circular/TwoPointer time. Ratios above 1 mean that the alternative was faster, and below 1 that Stock MiniSat was faster.

SAT 2005 competition (Le Berre & Simon, 2006) were used.[5] The reasons for this choice were that MiniSat did very well in that competition, so is unlikely to be a straw man, and that it provides a large but manageable set of benchmarks: by using the entire available set there is no chance of selection bias. While there have been significant advances in SAT solvers since 2005, I am not aware of major changes in propagation, which are the focus of the current paper. Code and results are available online, see Appendix D.

In the first experiment, most features of MiniSat were cut out, e.g. clause learning, conflict analysis, heuristics. By eliminating all other aspects of the solver, each variant searches identical spaces and the differences in speed must be due to differing speeds of propagation. No optimisations were applied to exploit the cut-down solver, so that the propagators being tested are the same as those used in the second experiment. However, since they must be propagated when running full MiniSat, a set of learnt clauses should be included. To do this, standard MiniSat was run on each instance for 60s (on a different Linux machine). For the 594 instances unsolved after 60s, the clause set was saved to give a realistic but static instance for the cut-down versions of MiniSat. Two versions of each propagator were created: one in which all instrumentation was switched off, to maximise speed; and one in which several additional counters were added to provide more metrics on the nature of the search for watched literals. For reporting cpu times the first version was used (with times being the median of three runs). Search was performed until a limit of 10^7 conflicts was reached (excepting one instance solved in 7.75×10^6 conflicts).

For tests of unrestricted MiniSat, three runs were performed for each algorithm-instance combination. MiniSat default settings were used with a cpu timeout of 1200s and memory-

5. `http://www.lri.fr/~simon/contest/results/download/distrib-benchs-random-sat2005.tar.bz2`, `distrib-benchs-crafted-sat2005.tar.bz2`, and `distrib-benchs-industrial-sat2005.tar.bz2`.

out of 1GB. For 87 instances, taking more than 0.01s and max-to-min deviation for some algorithm more than 10% of its median, another 18 runs were performed for each algorithm, and the median of all 21 runs used. Instances which no algorithm solved within the timeout, or all did in less than 0.01s, were discarded. 330 instances remained. Results are shown for both Circular and TwoPointer in Figure 4. There is huge variation between runtimes on the same instance, up to almost 100 times. Each propagation method can find different conflicting clauses, leading to different sets of learnt clauses and heuristics. The instances vertically above and diagonally below $x = 1200$ are those where one method timed-out and the other did not. A paired t-test was performed between Circular and Stock MiniSat, with null hypothesis that the distributions have the same mean. This gave $t = -0.127$, $p = 0.899$, i.e. a highly insignificant result. Because the assumption of normality is invalid, the t-test was randomised 100,000 times (Cohen, 1995). Of these, 48.3% gave a lower t-value and 51.7% a higher value. Similar results were obtained with TwoPointer and Stock MiniSat. The conclusion must be that there is no statistical evidence that either Circular or TwoPointer is either better or worse than Stock MiniSat in the fully featured solver.

Appendix B. Middle-Out List Scanning

This appendix gives formal presentation of algorithms and proofs for middle-out scanning for watched literals as discussed in Section 6.

Procedure 5: FNE-MiddleOut-Helper(list,*delta*)

Require: *delta* equals -1 or $+1$

```
last-cache := last
repeat
   last := last + delta
   if ACCEPTABLE(LIST,last) then return TRUE
until last = 0 or last = N
last:= last-cache
return FALSE
```

Procedure 6: FNE-MiddleOut(list)

Require: *delta* is persistent between calls and equals -1 or $+1$, initialised to either

```
if FNE-MiddleOut-Helper(delta) then
   return TRUE
else
   delta := −delta
   if FNE-Helper(delta) then
      return TRUE
   else
      return FALSE
```

First, note that FNE-MiddleOut is locally correct (Definition 6). Therefore by Theorem 7, FNE-MiddleOut maintains Invariant 2 at all times. FNE-MiddleOut turns out to be optimal in big-O terms, as follows from the analogue of Theorem 10.

Theorem 22. *In a downwards-explored search tree, the total number of calls to* ACCEPTABLE *made by successful calls to* FNE-MiddleOut *in an LBS is no more than* $2N$*.* (Proof in Appendix E online.)

From this result we can follow similar development as for circular, with analogous results, I omit these results except for the most important, which I state without proof.

Theorem 23. (Optimality) *In any downwards-explored search tree, the Middle-Out approach requires space for one* last *pointer and has a worst case of* $O(N)$ *calls to* ACCEPTABLE *per branch of the tree.*

The persistence of *delta* between executions is critical. If we add a line $0 : \textit{delta} = +1$ to Procedure 6 to give FNE-MiddleOut-Fixed, we get the following worse result.

Proposition 24. *In a downwards-explored search tree, the total number of calls to* ACCEPTABLE *made by* FNE-MiddleOut-Fixed *can be* $\Omega(N^2)$ *per branch of the search tree.* (Proof in Appendix E online.)

The solver JQuest by Lynce and Marques-Silva (2005) implements watched literals in the style of FNE-MiddleOut-Fixed, so is non-optimal. This cannot be deduced from the cited paper but can be seen from `http://sat.inesc.pt/sat/soft/jquest/jquest-src.tgz` in file ClauseSCImplWL.java: a flag controls which direction to move first in, but this is swapped at most once in a search for the first watch in a clause and never for the second.

Appendix C. Maintaining Multiple Pointers: Theory and Experiment

Compared with that described in Section 4, a more naive approach to implementing multiple watches is to have a separate *last* pointer for each one. To unit propagate correctly in SAT we have two watched literals. Crucially, we cannot allow two pointers to settle on the same element. A correct method to achieve this for unit propagation is as follows. If a pointer becomes unacceptable, then store its current value i and call FNE-Circular. If it fails the clause is entirely false. If it succeeds with a different value to the other pointer do nothing. If it succeeds with the value of the other pointer then call FNE-Circular again. If this second call fails then we must reset the value of the first pointer to the stored value i and unit propagate with the literal represented by the second pointer. To prove the optimality of this approach I need a more general version of Theorem 10.

Theorem 25. *Suppose that* W *pointers* $\mathrm{last}_1, \mathrm{last}_2, \ldots \mathrm{last}_W$ *to the same list are maintained simultaneously, with the same definition of acceptability, and that calls to* FNE *for a pointer* last_i *are made only when it points to an unacceptable element or to the same value as another pointer currently has. Then: if more than* cN *calls to* ACCEPTABLE *are made in any LBS in a downwards-explored search tree, either at least one of the calls to* FNE-Circular *fails or at least two of the pointers take the same value.* (Proof in Appendix E online.)

Theorem 25 leads to the correctness of the unit propagation procedure described above. It guarantees that when only one satisfiable literal remains in a clause, both pointers will settle on it and so unit propagation can be performed. The space requirement is $O(1)$ per *last* pointer. Detection of unacceptability can also be done in $O(1)$ time by maintaining a list of all occurrences of literals, to be consulted when a literal is set false. This gives:

Corollary 26. *Unit propagation using watched literals in a clause with N literals can be implemented in $O(1)$ space using $O(N)$ time per branch of a search tree.*

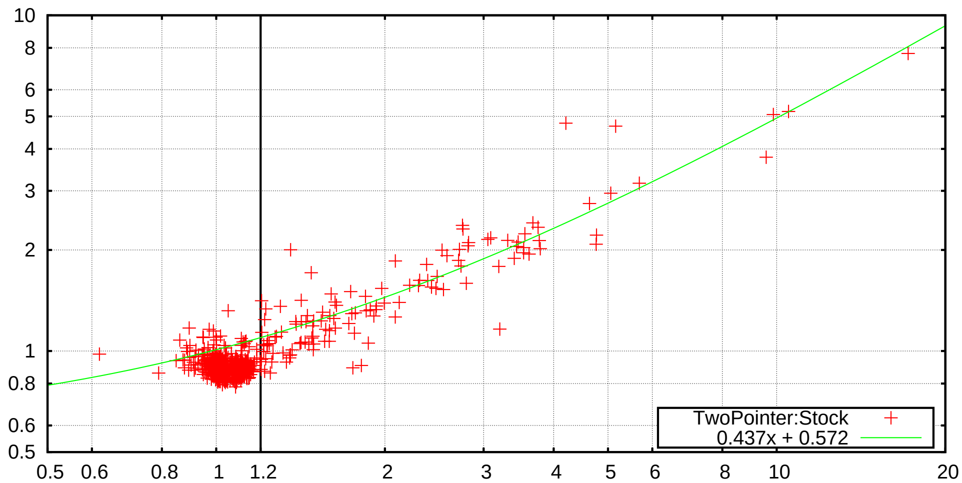

Figure 5: Scatterplot of ρ (x-axis) against speedup ratio of conflicts per second of TwoPointer over Stock MiniSat (y-axis). The line $0.437x + 0.572$ is the best-fit line in the region $\rho \geq 1.2$. The vertical line shows $\rho = 1.2$.

The development following Theorem 10 now follows as before. I state without proof:

Theorem 27. *In the conditions of Theorem 25, for a search tree containing k branches, calls to* FNE-Circular *make at most $k((c+1)N-1)$ calls to* acceptable.

TwoPointer unit propagates faster than Stock. When searching 594 instances in cut-down MiniSat up to 10 million conflicts, TwoPointer took a mean of 155.9s against 182.6s for Stock, so Stock takes 17.1% more time. This was about 10% slower than Circular's mean time of 141.6s. Circular is never more than 13% slower than TwoPointer or more than 35% faster. Both mean and median speedups of Circular over TwoPointer are 1.10. The median performance of Stock was slightly better than TwoPointer (78.1s to 81.1s) but Stock is never more than 22% faster while TwoPointer can be as much as 7.7 *times* faster.

We see similar results on the effect of blocked watches as with Circular. Results for Circular and definition of ρ are given in the main paper in Section 6. Where $\rho \geq 1.2$ (122 instances), ρ correlates very strongly with speedup in conflicts per second, with correlation coefficient $r^2 = 0.86$. Median speedup in this region is 1.32 and mean is 1.65. The best fit line is shown in Figure 5. For $\rho < 1.2$ (472 instances), there is no correlation between ρ and speedup, with $r^2 = 0.07$. Median and mean speedups are 0.88 and 0.90 (so are slowdowns not speedups.) In all regions, there is an extremely high correlation between TwoPointer and Circular, $r^2 > 0.996$. As with Circular, this analysis indicates that most of the speedup occurs in instances where TwoPointer is much better than Stock MiniSat at setting watches on literals which are likely to be valid later in search and thus block watched literal search.

Results for TwoPointer in the full version of MiniSat were similar to those with Circular. Under the methodology described in Appendix A, the raw t-value is -1.54, $p = 0.124$. Randomisation 100,000 times gave 26.2% lower t-values and 73.8% higher values.

Appendix D. Description of Online Appendices

Two Online Appendices are available. The first is a textual Appendix E with proofs omitted from the main text (Gent13a-appendix1.pdf).[6] The second contains the results tables, full MiniSat outputs, and graphs used for this paper (Gent13a-appendix2.tgz).[7] A fuller version of this appendix, including code for each variant of MiniSat and scripts to run and analyse experiments, is available separately.[8] The file is about 4MB and unpacks to about 12MB. Separately, a 2.4GB compressed tar file is available containing the clausesets written out after 60s failed search.[9]

References

Bessière, C., Régin, J.-C., Yap, R., & Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, *165*, 165–185.

Bessiere, C. (2004). Personal communication to Marc van Dongen.. Described by (van Dongen, 2004).

Bessière, C., & Régin, J.-C. (2001). Refining the basic constraint propagation algorithm. In Nebel, B. (Ed.), *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, pp. 309–315. Morgan Kaufmann.

Chai, D., & Kuehlmann, A. (2003). A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pp. 830–835. ACM.

Cohen, P. R. (1995). *Empirical methods for artificial intelligence*. MIT Press.

Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In Giunchiglia, E., & Tacchella, A. (Eds.), *SAT*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518. Springer.

Gent, I. P., Jefferson, C., & Miguel, I. (2006a). Minion: A fast scalable constraint solver. In Brewka, G., Coradeschi, S., Perini, A., & Traverso, P. (Eds.), *ECAI*, Vol. 141 of *Frontiers in Artificial Intelligence and Applications*, pp. 98–102. IOS Press.

Gent, I. P., Jefferson, C., & Miguel, I. (2006b). Watched literals for constraint propagation in Minion. In Benhamou, F. (Ed.), *CP*, Vol. 4204 of *Lecture Notes in Computer Science*, pp. 182–197. Springer.

Harvey, W. D., & Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95,*

6. Also available at `http://ipg.host.cs.st-andrews.ac.uk/JAIR/Gent13a-appendix1.pdf`
7. Also available at `http://ipg.host.cs.st-andrews.ac.uk/JAIR/Gent13a-appendix2.tgz`
8. `http://ipg.host.cs.st-andrews.ac.uk/JAIR/Gent13a-appendix2-full.tgz`
9. `http://ipg.host.cs.st-andrews.ac.uk/JAIR/writtenclausesets.tgz`

Montréal Québec, Canada, August 20-25 1995, 2 Volumes, Vol. 1, pp. 607–615. Morgan Kaufmann.

Huang, J. (2007). A case for simple SAT solvers. In Bessiere, C. (Ed.), *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, Vol. 4741 of *Lecture Notes in Computer Science*, pp. 839–846. Springer.

Jefferson, C., Moore, N. C. A., Nightingale, P., & Petrie, K. E. (2010). Implementing logical connectives in constraint programming. *Artificial Intelligence*, *174*(16-17), 1407–1429.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

Le Berre, D., & Simon, L. (2006). Special volume on the SAT 2005 competitions and evaluations. *JSAT*, *2*(1-4).

Likitvivatanavong, C., Zhang, Y., Bowen, J., & Freuder, E. C. (2005). Maintaining arc consistency using adaptive domain ordering. In Kaelbling, L. P., & Saffiotti, A. (Eds.), *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pp. 1527–1528. Professional Book Center.

Likitvivatanavong, C., Zhang, Y., Shannon, S., Bowen, J., & Freuder, E. C. (2007). Arc consistency during search. In Veloso, M. M. (Ed.), *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pp. 137–142.

Lynce, I., & Marques-Silva, J. P. (2005). Efficient data structures for backtrack search SAT solvers. *Ann. Math. Artif. Intell.*, *43*(1), 137–152.

Marques-Silva, J. P., Lynce, I., & Malik, S. (2009). Conflict-driven clause learning sat solvers. In Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.), *Handbook of Satisfiability*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, pp. 131–153. IOS Press.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pp. 530–535, New York, NY, USA. ACM.

Nightingale, P., Gent, I. P., Jefferson, C., & Miguel, I. (2013). Short and long supports for constraint propagation. *J. Artif. Intell. Res. (JAIR)*, *46*, 1–45.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, *9(3)*, 268–299.

Prosser, P., & Unsworth, C. (2011). Limited discrepancy search revisited. *J. Exp. Algorithmics*, *16*, 1.6:1.1–1.6:1.18.

Régin, J.-C. (2005). MAC algorithms during the search without additional space cost. In *Proc. 11th Principles and Practice of Constraint Programming (CP 2005)*, pp. 520–533.

van Dongen, M. R. C. (2004). Saving support-checks does not always save time. *Artif. Intell. Rev.*, *21*(3-4), 317–334.

Van Gelder, A. (2002). Generalizations of watched literals for backtracking search. In *Seventh Intl Symposium on AI and Mathematics*.

Zhang, H., & Stickel, M. E. (2000). Implementing the Davis-Putnam method. *J. Autom. Reasoning*, *24*(1/2), 277–296.