TPLP **21** (2): 125–144, 2021.
doi:10.1017/S1471068420000435 First published online 16 November 2020

Backjumping is Exception Handling

ED ROBBINS and ANDY KING
University of Kent, Canterbury, CT2 7NF, UK
(*e-mails:* edd.robbins@gmail.com, a.m.king@kent.ac.uk)

JACOB M. HOWE
City, University of London, EC1V 0HB, UK
(*e-mail:* j.m.howe@city.ac.uk)

submitted 15 September 2017; accepted 24 August 2020

Abstract

ISO Prolog provides catch and throw to realize the control flow of exception handling. This pearl demonstrates that catch and throw are inconspicuously amenable to the implementation of backjumping. In fact, they have precisely the semantics required: rewinding the search to a specific point and carrying of a preserved term to that point. The utility of these properties is demonstrated through an implementation of graph coloring with backjumping and a backjumping SAT solver that applies conflict-driven clause learning.

KEYWORDS: backjumping, exception handling, conflict-driven clause learning, SAT

1 Introduction

The ISO Prolog reference manual (Deransart *et al*. 1996) explains how catch and throw can pass control from one point of the program to another. The default behavior of `catch(Goal, Catcher, RecoveryGoal)` is to simply invoke `Goal`. However, if during execution, there is a call to `throw(Ball)` then control (and bindings) are unwound to the closest ancestor `catch` in the call stack which matches against the term `Ball`. Specifically, if the `Catcher` argument of the closest `catch` unifies with a copy of `Ball`, then the `RecoveryGoal` meta-call of that `catch` is invoked. Otherwise, control is unwound further until a matching `catch` is found. Since bindings are undone as the call stack is unwound, `Ball` might also be used to communicate information to `RecoveryGoal`, for example, to report the nature of a failure.

The power of this control flow construct is that it can transfer control to a specific point in the call stack using the `Ball` to target a specific `catch`. This is exactly what is required for backjumping. Backjumping (Stallman and Sussman 1977; Gaschnig 1979), in contrast to chronological backtracking, leaps across multiple levels in a search tree directly to the decision that triggered failure, rather than stepping through each decision, one by one. Backjumping has found application in truth maintenance systems (De Kleer 1986), logic programming (Bruynooghe 1980), constraint solving (Dechter 1990), and most recently in SAT to realize (Marques-Silva and Sakallah 1996) conflict-driven clause learning (CDCL). A CDCL solver requires not only search to be unwound to a specific decision (by backjumping), but also a term (a learnt clause) to be preserved and carried to that decision. Fortuitously, this facility is also provided by catch and throw.

The problem of adding control to the logic of a search algorithm sits at the very heart of logic programming (Kowalski 1979). However, how control is added, and the clarity of the control component, can be controversial. The problem of adding control is particularly acute when programming search problems like SAT, where the problem statement can be very simple, but the best algorithms analyze the decisions to focus search (as in CDCL). Moreover, backjumping is at odds with chronological backtracking, and as a consequence certain classes of algorithms that at first glance appear well suited to logic programming are, in fact, almost incompatible with the paradigm, at least in its purest form.

This pearl proposes catch and throw for programming backjumping, work that grew out of the (irritating) problem of how to clearly code CDCL in Prolog. This chimes with Bentley who coined the term programming pearl, and wrote, "Just as natural pearls grow from grains of sand that have irritated oysters, these programming pearls have grown from real problems that have irritated real programmers" (Bentley 1986). In contrast to a previous pearl (Bruynooghe 2004), which used a mutable database to orchestrate all aspects of intelligent backtracking, this paper breaks down the problem of implementing CDCL into its various components which are then matched against the language constructs of Prolog. The net result is clarity. To be precise, CDCL decomposes into three components: (1) rewinding search and bindings, (2) communication of a newly learned clause to its insertion point in the search tree, and (3) retaining learned clauses (across backtracking and backjumping). This paper argues that catch and throw provide (1) and (2), whereas (3) is naturally provided by a mutable database that might be implemented with a dynamic predicate, blackboard (De Bosschere and Jacquet 1993), or non-backtrackable global variables (Wielemaker *et al*. 2012). Applications are not limited to SAT, or even SMT (Robbins *et al*. 2015); to demonstrate versatility, the approach is first illustrated on the classic problem of graph coloring, providing a template for backjumping with `catch` and `throw`, then, second it is applied for SAT with learning.

Catch and throw have been advocated for programming backjumping before as part of a `comp.lang.prolog` discussion (Baljeu 2005), but the authors are not aware of any studies which actually demonstrate the viability of the idea. Deploying catch and throw in backjumping is unconventional since exception handling is intended to support exceptional behavior, whereas in backjumping these constructs are used for the intended control flow, which is in turn exceptional in the context of Prolog's execution model. The discussion of Baljeu (2005) is centered on the use of these non-logical ISO language features in Prolog programming.

The rest of this paper is structured as two case studies on backjumping, the first on graph coloring and the second on SAT. The graph coloring study has been chosen as a minimal example of depth-first search with backjumping, and the code provides a template for other examples. Section 2 explains how `catch` and `throw` can be used to realize backjumping for a graph coloring problem, where the edge constraints are realized as tests which check the color assigned to each vertex as it becomes bound. Section 3 moves onto SAT, building on the template provided by the graph coloring study to

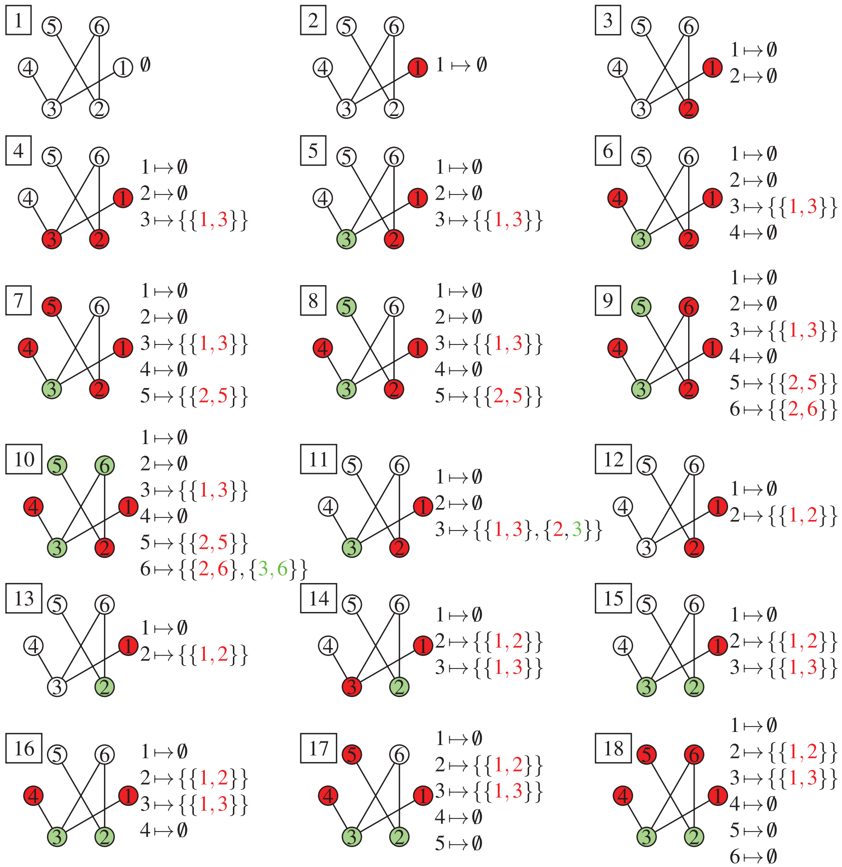

Fig. 1. Graph coloring with backjumping.

illustrate how `catch` and `throw` can be deployed to communicate (learnt) information back to the `catch`, necessary when guiding search using CDCL. Section 4 presents the concluding discussion.

2 Graph coloring

The first of the two worked examples in this paper considers graph coloring, adding backjumping to depth-first search. This example has been chosen as a minimal illustrating example, with the code in Figure 3 providing a template for other search problems, including the motivating example of SAT solving with CDCL.

Figure 1 illustrates depth-first search with backjumping for a coloring problem, where the objective is to assign red or green to each of the six vertices of the graph so that vertices which share an edge are colored differently. The example uses just two colors for simplicity. The vertices of the graph are ordered, as indicated by the numbering. Coloring commences at vertex 1, red is tried before green at each vertex, and white indicates the absence of a color assignment. Each partially colored graph is augmented with a map which associates each vertex with a (possibly empty) set of conflicts. A conflict is a

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

set of vertices with a color assignment that cannot be extended to satisfy all the edge constraints.

The conflict map is initially empty and is extended as each vertex is colored and contracted on backtracking and backjumping. The first conflict occurs in diagram 4 of Figure 1 between vertices 1 and 3 and is recorded in the conflict map for vertex 3; the conflict is always logged on the most recently assigned vertex. For expositional purposes, 1 and 3 are colored red indicating the partial color assignment when the conflict is detected.

Green has yet to be tried for vertex 3; hence, backtracking is applied to undo the assignment at vertex 3 and reassign it green (diagram 5). The conflict $\{1, 3\}$ is retained: the assignment to vertex 1 is still red but vertex 3 is now green. Thus, both vertices of the conflict, with the possible exception of the last, preserve their initial colors, which is a general pattern.

The next conflict arises at vertex 5 (diagram 7), causing the conflict $\{2, 5\}$ to be recorded. Not all colors have been considered at vertex 5 so again backtracking undoes the assignment to vertex 5 and it is reassigned green. Next, vertex 6 is colored red, which conflicts with vertex 2 (diagram 9). Again, not all colors have been tried at vertex 6, thus the vertex is reassigned to green, which then conflicts with vertex 3 (diagram 10). Notice that the first vertices (in the vertex ordering) of $\{2, 5\}$, $\{2, 6\}$, and $\{3, 6\}$ retain their initial color.

2.1 Conflict analysis for graph coloring

Now that all colors have been tried at vertex 6, backjumping is deployed after a form of conflict analysis which infers the target of the backjump. The conflicts for vertex 6 are $\{2, 6\}$ and $\{3, 6\}$, indicating that the conflicts at vertex 6 involve the assignments of vertices 2, 3, and 6, but not vertices 4 and 5. Moreover, one conflict occurs when vertices 2 and 6 are both red, and the other occurs when vertices 3 and 6 are both green. Hence, $\{2, 3\}$ is also a conflict, where 2 is red and 3 is green, since this partial assignment is incompatible with the edge constraints, irrespective of the color assigned to vertex 6. Therefore, a solution cannot be found without either reassigning the color at vertex 2, or vertex 3, or both. The conflict set of vertex 3 is augmented with $\{2, 3\}$ to give $\{\{1, 3\}, \{2, 3\}\}$ (diagram 11). Since vertex 3 was assigned more recently than vertex 2, it is selected as the target of the backjump and search resumes at vertex 3.

It should be noted that although the vertices of each conflict are colored in Figure 1, it is not necessary to introduce additional color assignments, one per conflict, to record these colors. To see this, observe how the colors of 2 and 3 in $\{2, 3\}$ align with the current color assignment because the 2 of $\{2, 6\}$ and the 3 of $\{3, 6\}$ also match with the current color assignment. Thus, the first vertices of both $\{1, 3\}$ and $\{2, 3\}$ match the current color assignment. Hence, the colors of the vertices of a conflict, with the possible exception of the last, match those of the current color assignment, a property which holds inductively.

Vertex 3 has already been assigned to both red and green, so again a form of conflict analysis is applied to infer that the partial assignment on the vertices 1 and 2 cannot be extended to a complete solution, whatever the color of vertex 3. Therefore, $\{1, 2\}$ is also a conflict, which is associated with vertex 2. Note how the colors of $\{1, 2\}$, which are both red, match the current color assignment. This conflict cannot be remedied without reassigning either vertex 1 or vertex 2 or both. The higher of the two, vertex 2, is thus

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

taken as the target of the backjump, which is then colored green (diagram 13). Conflicts on vertex 3 result in reassignment of that vertex to green (diagram 15), before the search proceeds to find a complete satisfying color assignment (diagram 18).

Finally, observe that the map data structure can be simplified by replacing each set of conflicts with a single set which is the union of all its conflicts (Bruynooghe 2004). Thus, the map for diagrams 9, 10, 11, and 12, are replaced by:

$$\begin{array}{l} 1 \mapsto \emptyset \\ 2 \mapsto \emptyset \\ 3 \mapsto \{1,3\} \\ 4 \mapsto \emptyset \\ 5 \mapsto \{2,5\} \\ 6 \mapsto \{2,6\} \end{array} \qquad \begin{array}{l} 1 \mapsto \emptyset \\ 2 \mapsto \emptyset \\ 3 \mapsto \{1,3\} \\ 4 \mapsto \emptyset \\ 5 \mapsto \{2,5\} \\ 6 \mapsto \{2,3,6\} \end{array} \qquad \begin{array}{l} 1 \mapsto \emptyset \\ 2 \mapsto \emptyset \\ 3 \mapsto \{1,2,3\} \end{array} \qquad \begin{array}{l} 1 \mapsto \emptyset \\ 2 \mapsto \{1,2\} \end{array}$$

Observe how entry $3 \mapsto \{1,2,3\}$ of the third map given immediately above can be found by unioning the vertex sets for $3 \mapsto \{1,3\}$ and $6 \mapsto \{2,3,6\}$ of its predecessor and then eliminating vertex 6, which is the vertex whose coloring induced the conflict analysis. In fact, not all the map needs to be accessed simultaneously: only the set of vertices for the highest identifier. This allows the map to be organized as a stack of lists, where only the topmost list of vertices on the stack is modified at any one time. This gives a straightforward data structure for implementing conflict analysis.

Notice that it is possible that an assignment leads to several conflicts. Here, the standard approach is taken – one conflict is selected to inform the backjump, and it is possible that one of the other conflicts is then encountered as search continues. An alternative approach would be to record all conflicts and backjump to the shallowest point in the search tree to guarantee that none of these would be encountered again. In the Prolog implementation considered in the next section, if there are several conflicts, the scheduler will determine which conflict is first encountered and used to make the backjump. The correctness of backjumping is addressed by Kondrak and van Beek (1997).

2.2 Backjumping in Prolog

Coloring can be realized by adopting a test-and-generate model in which checks suspend on their variables until they become sufficiently instantiated to apply the test. These checks then coroutine with a generator phase which binds the variables, one by one, that represent the colors of the vertices. SICStus code is provided to achieve this, where Figure 2 sets up the checks which define the coloring problem and Figure 3 applies labeling, with backjumping, to search for a satisfying assignment.

Setting up the checks for coloring. The predicate `colour(Vars, Values, Cs)` solves a coloring problem, where `Vars` is a list that specifies the color which is assigned to each vertex, `Values` defines the range of colors that can be assigned at each vertex, and `Cs` is a list of disequalities specifying the edge constraints. For the problem instance in Figure 1, the initial call would have

```
Vars = [One, Two, Three, Four, Five, Six]
Values = [red, green]
Cs = [One \= Three, Two \= Five, Two \= Six,
                    Three \= Six, Three \= Four]
```

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
colour(Vars, Values, Cs) :-
   problem_setup(Vars, Values, Cs, Pairs, AllIds),
   search(Pairs, [], AllIds).

problem_setup(Vars, Values, Cs, Pairs, AllIds) :-
   pairs(Vars, 1, Values, Pairs, AllIds),
   setup_checks(Cs).

pairs([], _Id, _Values, [], []).
pairs([Var | Vars], Id, Values, [Var-Values | Pairs], [Id | Ids]) :-
   Var = _-Id,
   NextId is Id + 1,
   pairs(Vars, NextId, Values, Pairs, Ids).

setup_checks([]).
setup_checks([C | Cs]) :-
   post(C),
   setup_checks(Cs).

post(X-XId \= Y-YId) :-
   suspend(X, Y, XId, YId).

:- block suspend(-, ?, ?, ?), suspend(?, -, ?, ?).
suspend(X, Y, XId, YId) :-
   (
      inconsistent(X, Y)
   ->
      MaxId is max(XId, YId),
      MinId is min(XId, YId),
      throw(ball(MaxId, [MinId, MaxId]))
   ;
      true
   ).

inconsistent(X, Y) :- X == Y.
```

Fig. 2. Setting up the checks for graph coloring.

The initial call first invokes `problem_setup(Vars, Values, Cs, Pairs, AllIds)` that is responsible for setting up `Vars`, `Pairs`, and `AllIds` and has as its second goal, `search(Pairs, [], AllIds)` for controlling the search. In setting up the problem, the predicate `pairs(Vars, 1, Values, Pairs, AllIds)` instantiates each element of the `Pairs` list to a term `Var-Values` where `Var` is itself instantiated to a pair `Value-Id` such that `Value` is drawn from the list `Values` and `Id` is a (ground) identifier; `AllIds` is instantiated to a list of all identifiers. The identifiers are numeric, indicating the position of `Var-Values` within `Pairs`, which tallies with the order in which variables are (later) assigned. The predicate `setup_checks(Cs)` posts each check given in a list of disequalities `Cs`. Each goal `post(X-XId \= Y-YId)` invokes `suspend(X, Y, XId, YId)` whose block declaration specifies that the `suspend` goal should not be called until both `X` and `Y` are instantiated. If inconsistency is detected on instantiation, then the term `ball(MaxId, CIds)` is thrown where `MaxId` is the largest identifier of the set `CIds` formed from `XId` and `YId` which identify the variables involved in the conflict. Note that corou-

```
search([], _, AllIds) :-
   succeed(AllIds).
search([(Var-Id)-[Value | RestValues] | Pairs], ConflictIds, AllIds) :-
   catch(
      (
         bind(Var,Value),
         search(Pairs, [], AllIds)
      ),
      ball(Id, CIds),
      (
         update_conflict(RestValues, CIds, ConflictIds, Id, NewConflictIds),
         search([(Var-Id)-RestValues | Pairs], NewConflictIds, AllIds)
      )
   ).

bind(Var, Value) :- Var = Value.

update_conflict([], CIds, ConflictIds, Id, NewConflictIds) :-
   merge(CIds, ConflictIds, NewConflictIds),
   delete(NewConflictIds, Id, RestConflictIds),
   max_member(MaxId, RestConflictIds),
   throw(ball(MaxId, RestConflictIds)).
update_conflict([_Value | _RestValues], CIds, ConflictIds, _Id, NewConflictIds) :-
   merge(CIds, ConflictIds, NewConflictIds).

succeed(_).
succeed(AllIds) :-
   max_member(MaxId, AllIds),
   throw(ball(MaxId, AllIds)).
```

Fig. 3. Backjumping search for graph coloring.

tining is described here using SICStus's block declarations (Carlsson and Mildner 2012). Although coroutining is not part of the ISO standard, other mainstream Prolog systems provide similar control constructs for delaying goals, such as `when` or `freeze`.

Backjumping for coloring. The predicate `search(Pairs, ConflictIds, and AllIds)` assigns the variables of `Pairs` to satisfy the constraints, failing if there is no solution. The `ConflictIds` argument maintains a conflict set for the variable of the first pair of `Pairs`.

Consider first the second clause of `search` which is responsible for orchestrating backjumping. Each meta-call `catch(Goal, Catcher, and RecoveryGoal)` has `Goal` that is concerned with assigning one variable, identified by `Id`, to the color `Value`. Binding `Var` to `Value` might wake up blocked calls to `suspend`, which will lead to consistency checks. If binding `Var` to `Value` does not lead to inconsistency, then search proceeds to search for an assignment to the remaining `Pairs`. If inconsistency is discovered, `ball(MaxId, CIds)` is thrown by a check in `suspend` and the call stack is unwound to the first enclosing `catch` for which `Catcher` unifies with `ball(MaxId, CIds)`. The `Catcher` term of each meta-call is `ball(Id, CIds)` where `Id` is ground and `CIds` is a variable, thus any `catch` which intercepts `ball(MaxId, CIds)` must possess an identifier `Id` which matches `MaxId`. In fact, this realizes backtracking; if a call to `bind` assigns `Var` (with identifier `Id`), wakes

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

`suspend` in Figure 2 and inconsistency is discovered, then both variables are instantiated; hence, `MaxId` must be `Id` and the `catch` directly enclosing the call to `bind` handles the exception. This then allows the conflict information in `CIds` to be passed back to the point where search resumes. An earlier ancestor will only intercept a ball thrown in `update_conflict` which is realizing backjumping. Note that `bind(Var, Value)` can unblock several `suspend` goals. Yet when the first goal resumes it will throw its ball, undoing the binding, so that the other `suspend` goals become blocked again.

`RecoveryGoal` has two calls, the first to `update_conflict` maintains conflict information for backjumping and the second continues search. If there are further colors to be tried, the second clause of `update_conflict` merges the conflict information for the current failure with that for previous failures, without duplication. The call to `search` will then assign the next color from `RestValues`. If there are no further colors to be tried, that is `RestValues` is empty, backjumping should occur. The first clause of `update_conflict` enables this. Conflict information is merged, the current assignment identifier is removed from the conflict list, and then the highest identifier remaining is the backjump level; hence, this and the conflict information are thrown as `ball(MaxId, RestConflictIds)`. This achieves backjumping by unwinding the call stack to where the variable with identifier `MaxId` is bound, while communicating the new conflict `RestConflictIds` to that part of the search. At this point, either a color remains to be assigned or backjumping is again applied, and so search continues. Observe that undoing and then reassigning a variable to another color, which is the essence of backtracking, is realized entirely using `catch` and `throw`.

If the first clause of `search` is matched, then all variables will have been assigned a color. This clause of `search` invokes the goal `succeed(AllIds)`, which will immediately succeed thereby returning the solution to the coloring problem. If another answer is requested, then a throw is used to reactivate search, the `MaxId` selected from `AllIds` which is list of all the variable identifiers. This results in search backtracking into the nonconflicting solution. The call to `succeed(AllIds)` can be omitted if it is sufficient to compute a single answer. It should be noted that all control is provided by catch and throw: it is not necessary to resort to a mutable database to maintain the conflicts and direct search (Bruynooghe 2004).

Search as presented in Figures 2 and 3 provides a template for implementing backjumping. Predicate `search` wraps `catch(Goal, Catcher, RecoveryGoal)` where the role of `Goal` is to bind variables to values, but if conflicts arise they are described by the term `ball(Id, CIds)` and caught by `Catcher`, and then the role of `RecoveryGoal` is to update conflict information and either continue search, or backjump, as appropriate.

3 SAT

Figure 4 lists the code for a Prolog SAT solver, adapted from Howe and King (2010; 2012) that uses watched literals to realize unit propagation. Given a propositional formula f in CNF over a set of variables X, and a partial (truth) function $\theta : X \to \{true, false\}$, unit propagation examines each clause of f to deduce another partial function $\theta' : X \to \{true, false\}$ that extends θ and that, if θ can be extended to satisfy f, necessarily holds. For example, suppose $X = \{x, y, z, u, v, w\}$, $f = (\neg x \vee z \vee \neg y) \wedge (\neg z \vee \neg u) \wedge (u \vee w \vee \neg v) \wedge (\neg w \vee v)$ and $\theta = \{x \mapsto true, y \mapsto true\}$. In this instance, the clause $(\neg x \vee z \vee \neg y)$

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
sat(Clauses, Vars) :-
   watch_clauses(Clauses),
   search(Vars).

search([]).
search([Var | Vars]) :-
   (Var = true; Var = false),
   search(Vars).

watch_clauses([]).
watch_clauses([Clause | Clauses]) :-
   watch_clause(Clause),
   watch_clauses(Clauses).

watch_clause([Pol-Var | Pairs]) :- set_watch(Pairs, Var, Pol).

set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1) :-
   watch(Var1, Pol1, Var2, Pol2, Pairs).

:- block watch(-, ?, -, ?, ?).
watch(Var1, Pol1, Var2, Pol2, Pairs) :-
   (
      nonvar(Var1)
   ->
      update_watch(Var1, Pol1, Var2, Pol2, Pairs)
   ;
      update_watch(Var2, Pol2, Var1, Pol1, Pairs)
   ).

update_watch(Var1, Pol1, Var2, Pol2, Pairs) :-
   (
      Var1 == Pol1
   ->
      true
   ;
      set_watch(Pairs, Var2, Pol2)
   ).
```

Fig. 4. A (vanilla) SAT solver using watched literals Howe and King (2010; 2012).

is unit, because it has only one unbound variable, z. Therefore, it can be deduced that, given θ, for the clause to be satisfied $z \mapsto \mathit{true}$. Moreover, for $(\neg z \vee \neg u)$ to be satisfied, it follows that $u \mapsto \mathit{false}$. The satisfaction of the remaining two clauses depends on two unknowns, v and w; hence, no further information can be deduced from them. Therefore, $\theta' = \theta \cup \{z \mapsto \mathit{true}, u \mapsto \mathit{false}\}$.

Searching for a satisfying assignment of f proceeds as follows: starting from an empty truth function $\theta = \emptyset$, unit propagation is applied to θ until either no further propagation is possible or a contradiction is established. In the first case, if all clauses are satisfied then f is satisfied, else an unassigned variable occurring in f, for instance x, is selected and the assignment $x \mapsto \mathit{true}$ is added to θ. In the second case, search backtracks to a

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

previous assignment, $y \mapsto \mathit{true}$ say, then adds $y \mapsto \mathit{false}$ to θ and continues with unit propagation.

The watched literals technique is founded on the simple observation that a particular clause is unit if it does not contain two unassigned variables (Moskewicz *et al*. 2001). Therefore, for each clause of a problem, two unassigned variables are watched; propagation may occur once either is assigned. It is not enough to watch just one variable because this is, in general, not sufficient for detecting if a clause becomes unit: the watch might be on the other, unassigned variable. The SAT solver in Figure 4 takes a problem in CNF, specified as a list of clauses, and a list of variables. Each clause is itself a list of pairs `Pol-Var`, where `Var` is a propositional variable, and `Pol` indicates whether the variable has positive or negative polarity by being either `true` or `false`, respectively. For the introductory example above, the initial call to `sat/2` would have

```
Clauses = [[false-X, true-Z, false-Y], [false-Z, false-U],
           [true-U, true-W, false-V], [false-W, true-V]]
Vars = [X, Y, Z, U, V, W]
```

For each clause, the `watch_clause` predicate invokes `set_watch` that, in turn, selects the first two variables in the clause to be watched. In the above, for the first clause. this leads to the invocation of `watch(X, false, Z, true, [false-Y])`; with neither `X` nor `Z` instantiated, this goal suspends via a delay declaration (specified using `block` in SICStus syntax). When a variable is instantiated, `watch` resumes and executes `update_watch`. If the instantiated variable matches its polarity, the clause is satisfied, and `update_watch` exits successfully, otherwise another variable is selected for watching. For the clause being considered, if `X` is bound to `true`, then `set_watch([false-Y], Z, true)` will be called which in turn will lead to the suspended `watch(Z, true, Y, false, [])`. If only one unbound variable remains, `set_watch` realizes unit propagation and assigns that variable so that the clause is satisfied. So if `Y` is bound to `true`, after waking `watch`, `set_watch` instantiates `Z` to be `true` by unit propagation; further, the second clause leads to `U` being instantiated to `false`. If there are no unassigned variables, or assigned and satisfying variables, then the clause is unsatisfiable and the `set_watch` goal will fail, and search will backtrack. If the example being considered is further extended by binding `V` to `true`, then the third clause will infer by unit propagation that `W` is instantiated to `true`. The fourth clause, which was suspended as `watch(W, false, V, true, [])` will wake and lead to the call `set_watch([], true, false)`, which fails. The search is realized using the `search` predicate to assign variables to `true` or `false`. The search for a satisfying assignment then proceeds as previously described, simply through Prolog backtracking.

3.1 Backjumping in a Prolog SAT solver

The goal of this paper is to augment the Prolog SAT solver in Figure 4 with CDCL. This is achieved by following the backjumping template from the previous section, where alongside the backjump, learnt clauses are added to the problem description. This section describes how backjumping for SAT is built, including the subtleties arising from accommodating clause learning. Figures 5, 6, 7, 8, and 10 extend the SAT solver of Figure 4 with infrastructure for backjumping and conflict analysis.

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
sat_cdcl(Clauses, Vars) :-
   put_learnt([]),
   ids(Vars, Ids, IdMap),
   vars(Vars, Ids),
   watch_clauses(Clauses),
   search_setup(Vars, IdMap, 1).

ids([Var | Vars], Ids, IdMap) :-
   length(Vars, Length),
   numlist(0, Length, Ids),
   maplist(zip, Ids, [Var | Vars], IdsVars),
   list_to_assoc(IdsVars, IdMap).

zip(X, Y, X-Y).

vars([], []).
vars([_-imp(_Level, Id, _Pol, _SubWhys) | Vars], [Id | Ids]) :-
   vars(Vars, Ids).
```

Fig. 5. Initial call and setup.

Variables and implication graphs. In Figure 5, the initial call to the CDCL solver is given, along with the setup of the variables. An identifier map, `IdMap`, which associates a ground identifier with the variable it represents, is created during setup to later reconstruct clauses during learning. This mapping is constructed up-front prior to invoking `watch_clauses`. An implication graph (Marques-Silva *et al.* 2009) for a variable `Var` is conceptually a DAG in which each node is represented as a term `imp(Level, Id, Value, Whys)` where `Level` records the decision level at which `Var` was assigned; `Id` is a (ground) identifier that is unique to the variable (used solely for deriving a ground representation of a learnt clause); `Value` is the truth value bound to the variable; and `Whys` is itself a list of implication graphs, interpreted as subtrees. Each propositional variable of the vanilla solver is replaced with a compound term `Var-Why` that pairs the variable `Var` with an implication graph `Why` that explains its instantiation. At the setup stage, the `Why` term of each pair `Var-Why` is unified with `imp(_Level, Id, _Value, _Whys)` where `Id` is the identifier for `Var` that ensures that each implication graph always carries its identifier.

Recall that in coloring, the decision level at which a variable is instantiated matches the position of the variable in the list `Vars` which also corresponds to its identifier. A consequence of unit propagation, however, is that the instantiation order does not necessarily follow the ordering of `Vars`; hence, the decision level does not necessarily match the identifier. Therefore, `Level` and `Id` are separately recorded in the `imp` structure. Furthermore, a learning solver reasons about the order in which variables are instantiated. Since, even at the same decision level, the instantiation of one variable can trigger the instantiation of another, the first element of `imp` is actually a pair `Level-SubLevel` where the integer `SubLevel` records the order in which a variable is instantiated within a given `Level`.

Setting up propagators for SAT. Figure 6 presents an enhanced version of `watch_clause` that supports conflict analysis. Unit propagation is extended to record the reason why a propositional variable is bound to a particular truth value. Since the `Var` argument of

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
watch_clause([Pol-VarWhy | Pairs]) :- set_watch(Pairs, VarWhy, Pol, []).

set_watch([], Var-Why, Pol, Whys) :-
   (
      (nonvar(Var), Var \== Pol)
   ->
      conflict([Why | Whys])
   ;
      unit(Var-Why, Pol, Whys)
   ).
set_watch([Pol2-(Var2-Why2) | Pairs], Var1-Why1, Pol1, Whys) :-
   watch(Var1, Why1, Pol1, Var2, Why2, Pol2, Pairs, Whys).

:- block watch(-, ?, ?, -, ?, ?, ?, ?).
watch(Var1, Why1, Pol1, Var2, Why2, Pol2, Pairs, Whys) :-
   (
      nonvar(Var1)
   ->
      update_watch(Var1, Why1, Pol1, Var2, Why2, Pol2, Pairs, Whys)
   ;
      update_watch(Var2, Why2, Pol2, Var1, Why1, Pol1, Pairs, Whys)
   ).

update_watch(Var1, Why1, Pol1, Var2, Why2, Pol2, Pairs, Whys) :-
   (
      Var1 == Pol1
   ->
      true
   ;
      set_watch(Pairs, Var2-Why2, Pol2, [Why1 | Whys])
   ).

unit(Var-Why, Pol, Whys) :-
   (
      nonvar(Var)
   ->
      true
   ;
      max_member(imp(MaxLevel, _, _, _), [imp(0-0, _, _, _) | Whys]),
      increment_sublevel(MaxLevel, NextLevel),
      Why = imp(NextLevel, _Id, Pol, Whys),
      Var = Pol
   ).

increment_sublevel(Level-SubLvl, Level-SubLvl1) :- SubLvl1 is SubLvl + 1.
```

Fig. 6. Setting up the propagators for SAT.

`watch_clause` is replaced by a pair `Var-Why`, the two `Var1` and `Var2` arguments of `watch` are accompanied with two additional arguments `Why1` and `Why2`. Then, the goal `conflict` uses the `Why` for `Var` and a list of implication graphs, `Whys`, for the other variables of the clause to diagnose the cause of the conflict, which is summarized as a clause. The goal ultimately terminates by throwing a ball which includes the learnt clause or triggers failure when an inconsistency is found with the clauses posted during setup. Notice how `Whys` is accumulated as each clause is traversed in `update_watch`.

The `unit(Var-Why, Pol, Whys)` goal calculates the decision level for a variable as part of the construction of its implication graph `Why`. This predicate is only invoked from `set_watch` when `Var` is either uninstantiated or bound to `Pol`. The latter case is vacuous, but the former case of `unit` binds `Var` to the truth value `Pol` and creates `Why` which records the reason for the binding. The `max_member` predicate harvests the maximum of the levels of implication graphs `Whys` of the other variables collected as the clause is traversed. The additional `imp(0-0, _, _, _)` term ensures that `MaxLevel` is well defined even when inconsistency is detected prior to any assignment by `search`. The predicate `increment_sublevel` merely increments the sublevel. The unification `Var = Pol` follows the bind to `Why` to ensure that every instantiated propositional variable is associated with a complete implication graph.

Search with backjumping for SAT. Figure 7 gives code which realizes backjumping search for SAT. The main search predicate is `search(VarWhys, IdMap, Level)` where, like before, `VarWhys` is a list of pairs, but here each pair is a variable conjoined with an implication graph which explains its binding. Search is controlled by overlaying backjumping with learning, in which the reason for a conflict is summarized by a conjunction of propositional literals whose negation gives a clause that is implied by the SAT instance. The clause (often referred to as a blocking clause) is then added to the problem to steer search away from the conflict. The map `IdMap` associates a ground identifier with the variable it represents, needed to reconstruct a (non-ground) clause from a conflict. `Level` is the decision level which a variable adopts if it is assigned by search rather than propagation. The first decision level is taken to be 1 (though, as discussed later, decision level 0 can also occur).

Consider first the second clause of `search` which is responsible for backjumping, learning, and labeling. Each `catch` meta-call is concerned with assigning one variable if it is unassigned. The predicate `bind` realizes labeling and instantiates `Var` to a truth value, while recording the value and level in a (leaf) node of an implication graph. Note that `bind` is never backtracked into, and assignment to `false` is not explicitly made. Since this code is realizing CDCL, failure of a binding will lead to a blocking clause being learnt and added to the problem description which will guide search away from the assignment that has failed, directing search to the `false` branch. If a conflict is discovered during search, the `set_watch` predicate in Figure 6 will call `conflict(Whys)` in Figure 7 and a ball is created with `BackjumpLevel` instantiated to the decision level of the backjump. The throw will thus only match the unique `Catcher` term `ball(Level, Clause)` for which `Level = BackjumpLevel`. The ground `BackjumpLevel` and `Clause` terms are formed by the predicate `analyse_conflict(Whys, BackjumpLevel, Clause)` using a conflicting list of implication graphs `Whys` discovered when watching a clause (see Section 3.2 and Figure 10). On recovery, the bindings made by `search` and propagation are unwound back to the backjump level as required (though the bindings on the ball are retained). The `RecoveryGoal` then uses `update_conflict` in Figure 8 to add `Clause` to a database of learnt clauses (the learning algorithm explained in Section 3.2) which grows as search proceeds. The ground `Clause` is translated into its non-ground representation at the stage of the `RecoveryGoal` where the decision level reverts to that of `BackjumpLevel`:

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
search_setup(Vars, IdMap, Level) :-
   catch(
      search(Vars, IdMap, Level),
      ball(0, Clause),
      (
         update_conflict(1, Clause, 0, IdMap),
         search_setup(Vars, IdMap, Level)
      )
   ).

search([], _, _).
search([Var-Why | VarWhys], IdMap, Level) :-
   NextLevel is Level + 1,
   catch(
      (
         bind(Var-Why, Level),
         search(VarWhys, IdMap, NextLevel)
      ),
      ball(Level, Clause),
      (
         update_conflict(NextLevel, Clause, Level, IdMap),
         search([Var-Why | VarWhys], IdMap, NextLevel)
      )
   ).

bind(Var-Why, Level) :-
   (
      var(Var)
   ->
      Var-Why = true-imp(Level-0, _, true, [])
   ;
      true
   ).

conflict(Whys) :-
   analyse_conflict(Whys, BackjumpLevel, Clause),
   throw(ball(BackjumpLevel, Clause)).
```

Fig. 7. Backjumping search for SAT.

the variables of the clause can have bindings at later decision levels which do not hold at the `BackjumpLevel`. Other elements of the database of learnt clause may also need to be restored in a similar way.

The predicates `get_learnt(Learnt)` and `put_learnt(Learnt)` are merely wrappers to built-ins that read and write a list of (learnt) clauses to a non-backtrackable database: they can be realized with `bb_get/bb_put` as illustrated in Figure 8, or `nb_getval/nb_setval` (Wielemaker *et al.* 2012), or using the assert/retract family of built-ins.

When considering graph coloring in Section 2, colors are always explicitly assigned in a search phase, and a conflict can only occur during search, after a color is assigned. However, SAT is different: a variable might be assigned a value before search at the initial setup. The decision level of such an assignment is taken to be 0. Conflicts which arise

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
update_conflict(NextLevel, Clause, Level, IdMap) :-
   get_learnt(Learnt),
   Learnt1 = [NextLevel-Clause | Learnt],
   add_learnt_clauses(Learnt1, Level, IdMap),
   put_learnt(Learnt1).

add_learnt_clauses([], _, _).
add_learnt_clauses([Level-Clause | Clauses], DecisionLevel, IdMap) :-
   (
      Level < DecisionLevel
   ->
      true
   ;
      unground(Clause, IdMap, NewClause),
      watch_clause(NewClause)
   ),
   add_learnt_clauses(Clauses, DecisionLevel, IdMap).

unground([], _, []).
unground([Pol-Id | Labels], IdMap, [Pol-VarImp | Literals]) :-
   get_assoc(Id, IdMap, VarImp),
   unground(Labels, IdMap, Literals).

get_learnt(Learnt) :- bb_get(learnt, Learnt).
put_learnt(Learnt) :- bb_put(learnt, Learnt).
```

Fig. 8. Conflict analysis.

at this level are handled by the singleton case in `analyse_conflict`, which immediately fails without reaching a `throw`, since if the problem specification is unsatisfiable then no further search is necessary. Furthermore, with assignment at decision level 0 possible, conflict analysis might determine that level 0 is the appropriate decision level to jump to. This is prior to the call to `search`, hence the addition of `search_setup` to handle these backjumps; this predicate mirrors `search`, but without the call to `bind`. In addition, it is possible that the backjump caught by `ball(0, Clause)` describes a unit `Clause` and this itself leads to assignments at decision level 0.

3.2 Conflict analysis for SAT

When a conflict is encountered, the implication graph leading to it is examined to learn a clause, which is added to the formula to steer search away from the conflict. To illustrate, consider the formula $f = (\neg x_1 \vee x_8 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_4) \wedge (\neg x_4 \vee \neg x_5) \wedge (x_5 \vee x_6) \wedge (x_7 \vee \neg x_4 \vee \neg x_6)$ and the partial assignment $\theta = \{x_7 \mapsto \mathit{false}, x_8 \mapsto \mathit{false}\}$ where the variables x_7 and x_8 are assigned at decision levels 1 and 2, respectively, by search. Observe that no further bindings are inferred by unit propagation. However, if search subsequently adds $x_1 \mapsto \mathit{true}$ at decision level 3, then a series of unit propagations ensue that ultimately lead to a conflict, owing to unsatisfiability of the clause $(x_7 \vee \neg x_4 \vee \neg x_6)$.

Unique Implication Points. Figure 9 illustrates an implication graph rooted at a special conflict node, κ, that details how the conflict follows from binding x_4, x_6, and x_7. Each

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

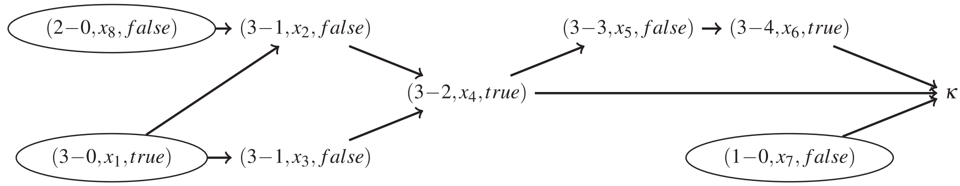

Fig. 9. Implication graph from f and the partial assignment $\theta = \{x_7 \mapsto \mathit{false}, x_8 \mapsto \mathit{false}, x_1 \mapsto \mathit{true}\}$.

triple in the figure gives the level and sublevel, the variable assigned, and its truth value. The implication graph can be inspected to learn a clause. The leaves of the implication graph, which are circled in Figure 9, are the three bindings $x_7 \mapsto \mathit{false}$, $x_8 \mapsto \mathit{false}$, and $x_1 \mapsto \mathit{true}$, that together with clauses of f, prohibit the satisfiability of $(x_7 \vee \neg x_4 \vee \neg x_6)$. This combination of bindings can be avoided by adding $(x_7 \vee x_8 \vee \neg x_1)$ to f which is therefore a candidate learnt clause. However, another choice is possible, notably one with fewer literals. Observe that any path that starts with the binding of x_1 at decision level 3, the current decision level, and ends at κ passes through the intermediate node where x_4 is assigned. The single binding $x_4 \mapsto \mathit{true}$ therefore summarizes the net effect of the two bindings $x_1 \mapsto \mathit{true}$ and $x_8 \mapsto \mathit{false}$. Thus, an alternative learnt clause is $(\neg x_4 \vee x_7)$. In general, any node between the current (most recent) decision variable and the conflict κ that strictly dominates (Cooper *et al.* 2006) κ can be used to construct a learnt clause. Such nodes are termed Unique Implication Points (UIPs). The UIP nearest κ is the first UIP (the node where x_4 is assigned). The last UIP is furthest from κ and is where the current decision variable is bound (the node where x_1 is assigned).

Figure 10 presents code for creating a learnt clause based on the last UIP. The predicate `construct_clause` performs a depth-first traversal of the implication graph, starting from κ, and identifies decision variables by their empty implication graphs (`SubWhys`). The constructed clause is made up of literals where variables are identified by the `Id` ground term, and polarities are the negation of those that caused the conflict.

The `construct_clause` predicate also gathers the decision levels at which literals in the new clause were assigned. These levels are used to find the backjump level, which is chosen to be such that the learnt clause becomes unit, directing search away from the conflict. For the last UIP learnt clause $x_7 \vee x_8 \vee \neg x_1$ of Figure 9, `construct_clause` will derive the clause `[false-x1, true-x8, true-x7]` with decision levels `[3, 2, 1]`. Search should resume by backjumping to level 2. Therefore, the call to `conflict(Whys)` leads to `throw(ball(2, [false-x1, true-x8, true-x7]))`. More generally, the backjump level is the largest level strictly less than the maximum (which is actually the current decision level); search would not immediately benefit from the new clause if resumed at an earlier decision level. Continuing at the backjump level ensures that the learnt clause becomes unit almost immediately.

The act of backjumping removes bindings, in particular those induced by recently added clauses. However, these clauses are retained in the non-backtrackable database using a ground representation. Figure 8 lists the code for reinstating learnt clauses after backjumping using the predicate `add_learnt_clauses`. Learnt clauses are saved together with the level at which they were learned. Upon backjumping, the calls to `watch_clause` for clauses learnt at the backjump decision level or later are lost, and these calls need to

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

```
analyse_conflict(Whys, BackjumpLevel, Clause) :-
   construct_clause(Whys, Lits, Levels),
   remove_dups(Lits, Clause),
   sort(Levels, SortedLevels),
   (
      SortedLevels = [BackjumpLevel]
   ->
      BackjumpLevel \== 0
   ;
      reverse(SortedLevels, [_, BackjumpLevel | _])
   ).

construct_clause([], [], []).
construct_clause([imp(Level, Id, Pol, SubWhys) | Whys], Lits, Levels) :-
   (
      SubWhys == []
   ->
      negate(Pol, NegPol),
      Lits = [NegPol-Id | MoreLits],
      Levels = [Level | MoreLevels],
      construct_clause(Whys, MoreLits, MoreLevels)
   ;
      append(SubWhys, Whys, AllWhys),
      construct_clause(AllWhys, Lits, Levels)
   ).

negate(true, false).
negate(false, true).
```

Fig. 10. Inferring and instantiating learnt clauses.

be made again. This necessitates rebuilding clauses after backjumping from the ground representation of the database. The `unground` predicate achieves this using association list `IdMap` to map a ground identifier to its propositional variable. This list is built during problem setup and is passed through `search` as shown in Figure 7.

Reinstatement of learned clauses can be performed selectively to realize k-learning. In k-learning (Marques-Silva and Sakallah 1996), learned clauses are only added to the constraint store permanently if they have less than k variables. It decreases the cost of learning by lessening pressure on memory and reducing the number of updates to the store of clauses. Clauses with fewer variables have most influence, but learning will have little impact if k is set too low. The presented approach can be also modified to allow first UIP clause learning. This typically produces smaller clauses more tightly focused on the cause of the conflict (Marques-Silva *et al*. 2009). First UIP can be realized in Prolog by running a frontier over the implication graph, starting at κ, repeatedly expanding the node with the highest level and sublevel. The first UIP is found when the frontier reduces to contain a single node at the highest level (possibly augmented with nodes of lower level). First UIP can thus be found in a single pass over the conflict graph without introducing additional data structures.

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

Table 1. *Performance of learning versus no learning*

	No learning		8-learning			
Benchmark	Time	Assign	Time	Assign	Throws	Jumps
CBS_k3_n100_m435_b90_127	2987	179,427	2206	56,329	24,650	31,667
CBS_k3_n100_m435_b90_139	3204	193,247	874	26,827	10,163	16,647
flat175-17	22,791	748,377	10,225	107,066	43,546	63,477
flat175-28	15,754	471,521	15,132	154,985	70,895	84,059
flat200-20	16,345	519,868	3252	39,674	16,107	23,524
flat200-39	>30,000		13,326	125,053	54,516	70,502
uf100-0126	4097	248,581	1984	53,320	23,009	30,293
uf100-015	3499	210,330	857	24,676	10,101	14,559
uuf100-0119	9651	634,568	3644	95,553	40,073	55,497
uuf100-0120	5731	350,866	2578	67,868	28,378	39,498

Backjumping is designed to accelerate search which begs the question of whether backjumping, when implemented with `catch` and `throw`, can ever improve on the default search mode of Prolog. Table 1 presents timings for a biased sample of classic SAT benchmarks; biased because they were chosen to be nontrivial in that the execution time exceeds 2 s for the vanilla solver. Times are in milliseconds, and benchmarking was carried out with SICStus 4.5.1 on a 2.5-GHz Macbook Pro with 16GB RAM. The first two benchmarks are random 3-SAT instances with controlled backbone size, the next four originate from flat graph coloring problems, and the final four include two unsatisfiable and two satisfiable random 3-SAT instances. The table gives data for the vanilla solver with no learning, and for the backjumping solver given in this paper, augmented with first UIP CDCL and using k-learning with $k = 8$. For both solvers, the execution time in milliseconds is given (time), alongside the number of variable assignments made (assign). In addition, for the backjumping solver, the number of throws made by the solver (throws), and the number of assignments jumped over (jumps) are given. The balls thrown in backjumping are the size of the analyzed conflict which is typically less than 20 literals for these benchmarks, though this is problem-dependent. The timings suggest that `catch` and `throw` are not only a useful code structuring device but can enable significant improvement in performance when used to realize backjumping. As is conventional in SAT solving, the timings are for finding a single solution. To enumerate all solutions, the template from graph coloring using the `succeed` predicate can be adapted to add a blocking clause for the solution and reactivating search reusing the learnt clauses. Code for the work presented in this paper is available at `https://www.cs.kent.ac.uk/~amk/backjump.zip`, including some variations not discussed here, such as pruning the set of learnt clauses by removing those which are entailed by a newly learnt clause.

4 Concluding Discussion

The idea of using catch and throw to realize backjumping dates back at least to a discussion (Baljeu 2005) on `comp.lang.prolog` but the authors are not aware of any programming examples that serve to illustrate the technique. The message of this discussion is exemplified with two case studies which demonstrate that `catch` and `throw` are more

https://doi.org/10.1017/S1471068420000435 Published online by Cambridge University Press

versatile than one would expect, providing exactly what is required for programming backjumping. In fact, random solver restarts (Howe and King 2012) can also be accommodated by adding an outer `catch` meta-call which intercepts restart exceptions thrown when the number of backjumps exceeds a threshold.

Intelligent backtracking (Bruynooghe 2004) can be combined with learning by restarting the search from scratch and then fast-forwarding to a particular decision point (Howe and King 2012). However, `catch` and `throw` provide a more general solution, not requiring search to be completely restarted, and there seems no reason why this strategy cannot extend to SMT solving (Robbins *et al*. 2015) by another instantiation of the template. It has been recently shown (Drabent 2018) that the vanilla SAT solver of Figure 4 can be understood as a logic program with added control; however, reasoning about the correctness of learnt clauses is more challenging still, a research problem that is not exclusive to logic programming.

Realizing backjumping with an ISO feature is undoubtedly attractive, and since ISO encourages the use of catch and throw, one might hope that increasingly efficient implementations will emerge over time. Even though SWI-Prolog does not comply with the ISO specification (since it does not copy the `Ball` before unifying it with `Catcher`), the two case studies are fully portable because the `Goal` and `Catcher` goals do not share variables.

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068420000435.

Acknowledgments

The authors thank the editor and anonymous referees for their thought-provoking comments. This is a better paper for these suggestions. This work was supported, in part, by EPSRC grants EP/K031929/1 and EP/N020243/1.

References

Baljeu, A. 2005. Using Exceptions to Implement Backjumping? comp.lang.prolog newsgroup discussion. URL: `http://coding.derkeiler.com/Archive/Prolog/comp.lang.prolog/2005-09/msg00048.html`.

Bentley, J. 1986. *Programming Pearls*. Addison-Wesley: New Jersey.

Bruynooghe, M. 1980. Analysis of dependencies to improve the behaviour of logic programs. In *International Conference on Automated Deduction*. Lecture Notes in Artificial Intelligence, vol. 87. Springer, 293–305.

Bruynooghe, M. 2004. Enhancing a search algorithm to perform intelligent backtracking. *Theory Practice of Logic Programming 4,* 3, 371–380.

Carlsson, M. and Mildner, P. 2012. SICStus Prolog – the first 25 years. *Theory and Practice of Logic Programming 12,* 1–2, 35–66.

Cooper, K. D., Harvey, T. J. and Kennendy, K. 2006. A Simple, Fast Dominance Algorithm. Technical Report TR-06-33870, Rice University, Computer Science. URL: `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`.

De Bosschere, K. and Jacquet, J. 1993. Multi-Prolog: Definition, operational semantics and implementation. In *International Conference on Logic Programming*. MIT Press, 299–313.

De Kleer, J. 1986. An Assumption-based TMS. *Artificial Intelligence 28,* 2, 127–162.

Dechter, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence 41*, 273–312.

Deransart, P., Ed-Dbali, A. and Cervoni, L. 1996. *Prolog: The Standard.* Springer.

Drabent, W. 2018. Logic + Control: On program construction and verification. *Theory Practice of Logic Programming 18,* 1, 1–29.

Gaschnig, J. 1979. *Performance Measurement and Analysis of Certain Search Algorithms*. Ph.D. thesis, Carnegie Mellon University, Department of Computer Science, Pittsburgh, Pennsylvania 15213. CMU-CS-79-124.

Howe, J. M. and King, A. 2010. A pearl on SAT solving in Prolog. In *Functional and Logic Programming.* Lecture Notes in Computer Science, vol. 6009. Springer, 165–174.

Howe, J. M. and King, A. 2012. A pearl on SAT and SMT solving in Prolog. *Theoretical Computer Science 435*, 43–55.

Kondrak, G. and van Beek, P. 1997. A theoretical evaluation of selected backtracking algorithms. *Artifical Intelligence 89*, 365–387.

Kowalski, R. A. 1979. Algorithm = Logic + Control. *Communication of the ACM 22,* 7, 424–436.

Marques-Silva, J. P., Lynce, I. and Malik, S. 2009. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. Van Maaren and T. Walsh, Eds. IOS Press, 131–153.

Marques-Silva, J. P. and Sakallah, K. A. 1996. GRASP – A new search algorithm for satisfiability. In *Computer Aided Design.* IEEE, 220–227.

Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference.* ACM Press, 530–535.

Robbins, E., Howe, J. M. and King, A. 2015. Theory propagation and reification. *Science of Computer Programming 111,* 1, 3–22.

Stallman, R. and Sussman, G. J. 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence 9*, 135–196.

Wielemaker, J., Schrijvers, T., Triska, M. and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1–2, 67–96.