The Logic of Search Algorithms: Theory and Applications*

Ian P. Gent[1] and Judith L. Underwood[2]

[1] APES Research Group, Department of Computer Science,
University of Strathclyde, Glasgow G1 1XH, United Kingdom. ipg@cs.strath.ac.uk
[2] BeAUTy Research Group, Department of Computing Science,
University of Glasgow, Glasgow G12 8QQ, United Kingdom. jlu@dcs.gla.ac.uk

Abstract. Many search algorithms have been introduced without correctness proofs, or proved only with respect to an informal semantics of the algorithm. We address this problem by taking advantage of the correspondence between programs and proofs. We give a single proof of the correctness of a very general search algorithm, for which we provide Scheme code. It is straightforward to implement service functions to implement algorithms such as Davis-Putnam for satisfiability or forward checking (FC) for constraint satisfaction, and to incorporate conflict-directed backjumping (CBJ) and heuristics for variable and value ordering. By separating the search algorithm from problem features, our work should enable the much speedier implementation of sophisticated search methods such as FC-CBJ in new domains, and we illustrate this by sketching an implementation for the Hamiltonian Circuit problem.

1 Introduction

The constraint satisfaction community has an excellent record of introducing intelligent search procedures for binary constraint satisfaction problems. However, the record is less impressive in formally proving such algorithms correct, and in encouraging the application of the same search methods to other NP-complete problems. These two failings represent problems for the community both theoretically, in the absence of formal correctness proofs, and practically, in that the most appropriate search techniques may be reinvented in several problem classes, or simply not used at all outside the CSP community.

In this paper we show that both the theoretical and practical problems can be addressed by separating out the search algorithm from the details of the problem domain. We give a correctness proof of a very general search algorithm using techniques from theoretical computer science, and give Scheme code implementing this algorithm. Then we sketch implementations of a number of search algorithms in a variety of NP-complete problem domains simply by providing

* Judith Underwood is supported by EPSRC award GR/L/15685. We thank members of APES, particularly Patrick Prosser and Toby Walsh for their code. We especially thank Mr Denis Magnus for his invaluable contributions to our research.

service functions to our code. To get the full benefit of, say, conflict-directed backjumping (CBJ), domain-specialist need not implement it, nor even understand all the nuances of the technique. If suitably extended, our sketches could form the basis of full correctness proofs of algorithms which have not yet been proved correct, for example MAC-CBJ (maintaining arc-consistency with CBJ.)

2 The Logic of Programs

We consider problems which may be described by a finite set of variables, each of which may take a finite number of values, and a decidable predicate P on an assignment of values to these variables. In this section, we prove a theorem which essentially states that for any problem instance, either there is or is not an assignment which satisfies P. In classical logic, this theorem is trivial. However, we treat the theorem *constructively*: in order to show that this theorem is true in a constructive logic, we must have a decision procedure which, given a problem instance, produces a satisfying assignment if one exists. The proof we give here is designed in such a way that it constitutes a correctness proof of a large family of backtracking search algorithms. The connection between the proof and the algorithms it proves correct is best described by constructive type theory. Type theories are widely used in the theorem proving and formal methods communities [4, 14, 9]; we outline here the main ideas and how they will be used in this setting.

In general, a type theory is an expressive logical language together with rules describing properties of types and terms of each type. Types can be interpreted as formulas; this idea is known as the *Curry-Howard correspondence* [12]. For example, $A \to B$ is the type of functions from A to B, but it also expresses the logical implication that if we know $A \to B$ and A, we know B. More precisely, if we have a *proof* of $A \to B$ and a *proof* of A then we can produce a proof of B. A proof of $A \to B$ is simply any function f from A to B; similarly, a proof of A is an element a of A, and the resulting proof of B is produced by applying f to a. Thus, proofs are programs and programs are proofs.

Given a reasonably expressive language of formulas, we may specify the desired behaviour of a function by stating a theorem; a proof of that theorem then corresponds to a function which is guaranteed to have that behaviour. For example, the theorem $\forall x : N.\exists y : N.(y^2 \leq x) \wedge (x < (y+1)^2)$ asserts that for any natural number x, there exists a natural number y which is the integer part of the square root of x. A constructive proof of this theorem would correspond a function which actually computes the integer square root.

Type theoretic proofs generally carry more information than ordinary programs do since, in addition to producing data, they must provide proofs of properties of the data. Also, not all mathematical proofs can be expressed in a way which corresponds to a program; we work in a constructive type theory which restricts the proof techniques to those which do generate programs.

This approach has many advantages. It is easier to check a proof automatically than to check code, since the proof contains logical information which is not necessarily used in the computation. This means that code extracted from a proof is guaranteed correct, given the correctness of the proof development and program execution environments. Another significant advantage is in modularity. The proof is, in general, more abstract than the code. For example, the proof may say "choose the next variable to set". Computationally, this is a requirement that a function `p-choose-var`[3] exists and satisfies certain requirements, namely, that it actually returns a variable from the set of unused variables. The proof requires the existence of such a function but does not specify how it is to be implemented. Thus, a new function may be supplied which implements some variable ordering heuristic, and as long as it satisfies the requirements generated by the proof, the program using this new function is still guaranteed correct.

To avoid having to present a complex type theory, we present the proof in ordinary mathematical language. The Scheme program corresponding to this proof is given in full in the Appendix. This correspondence is informal – although we have formalized the core of the proof in Lego [14, 18] and extracted a type theoretic program, we have not produced our code by translating that program to Scheme by some verified method. However, the code we present closely follows the structure of the proof, and in theory could be shown to behave in the same way as the extracted program. To reinforce the connection, in the presentation of the proof we will frequently refer to the corresponding sections of the program.

The remainder of this section is devoted to a proof of the following theorem:

Theorem 1. *Given a finite set of variables, Varset, a finite set of values Valset, and a predicate P on assignments of variables to values, then*

$$(\exists A : \mathit{Assign(Varset, Valset)}\ .P(A)) \vee (\forall A : \mathit{Assign(Varset, Valset)}\ .\neg P(A))$$

The notation $A : \mathit{Assign(Varset, Valset)}$ means A is in the type of assignments of values in *Valset* to variables in *Varset*. If an assignment A is only defined on a subset *Vars* of *Varset*, this is denoted by $A : \mathit{Assign(Vars, Valset)}$. Thus, if $A : \mathit{Assign(Vars, Valset)}$, a value has been assigned to every variable in *Vars*. On the types *Varset* and *Valset*, and on the type of assignments, we assume we have the strict ($\subset$) and non-strict ($\subseteq$) subset/subassignment orderings.

We prove Theorem 1 as a corollary to a more general theorem about partial assignments, in which we have introduced conflict sets explicitly. Constructive logic has a *semantics of evidence*: to prove $\forall A : \mathit{Assign(Varset, Valset)}\ .\neg P(A)$, we must show, for any A, some kind of evidence that the assumption $P(A)$ leads to a contradiction. This is given in the form of a conflict set, which is a set CS of variables in the current assignment A and a guarantee that if A' satisfying $P(A')$ exists, the value of some $v \in CS$ in A' is different from its value in A. This ensures that no extension of A satisfies P. The conflict set is only one way of expressing this, and the idea of the proof will succeed for any kind of structure from which we can conclude $\neg(\exists A'.A \subseteq A' \wedge P(A'))$. We will later use our logical description of conflict sets to justify backjumping and forward checking.

[3] Typographically, names in `teletype` font are also the names of functions in the Scheme code in the Appendix, or of service functions required to be implemented in a given domain for the Scheme code to work.

Theorem 2. *Given a finite set of variables, Varset, a finite set of values Valset, and a predicate P on full assignments of values to variables, then*

$$\begin{array}{l}
\forall \mathit{Vars} \subseteq \mathit{Varset}. \\
\quad \forall A : \mathit{Assign(Vars, Valset)}\ . \\
\quad\quad \exists A' : \mathit{Assign(Varset, Valset)}\ .A \subseteq A' \wedge P(A') \\
\quad \vee \\
\quad\quad \exists CS \subseteq \mathit{Vars}.\forall A' : \mathit{Assign(Varset, Valset)}\ . \\
\quad\quad\quad P(A') \rightarrow \exists v_0 \in CS.\, \mathit{Val_of}\,(v_0, A') \neq \mathit{Val_of}\,(v_0, A)
\end{array}$$

Theorem 1 follows from Theorem 2 by taking *Vars* to be empty, and noting that this requires CS to be empty and thus $P(A)$ leads to a contradiction.

The proof of Theorem 2 will be by induction. The proof is designed so that its computational meaning is a function which performs backtracking search. A proof by induction over a well-founded partial order corresponds to a function defined by well-founded recursion; thus the function can be defined recursively but evaluation is guaranteed to terminate because each recursive call must be applied to arguments which are smaller in the partial order. Such a recursive call corresponds to use of the inductive hypothesis in the proof. Type theoretic languages usually make this explicit by using a special operator for defining functions inductively. To make the code more readable, we use ordinary recursive calls but emphasize that the recursion must terminate.

There are actually two inductions involved in the proof of the theorem. The first is on the size of the set of variables yet to be assigned values. Given a variable to set, the second induction is on the size of the set of values of that variable which have not been tested. This leads to two functions, `test` and `enumerate-domain`. The function `test` takes a given partial assignment and tries to extend it by setting a new variable, while `enumerate-domain` takes a partial assignment and a new variable and tries the possible values for that variable.

We describe in some detail the core of the proof and show how it corresponds to a backtracking search procedure. We then discuss various extensions and generalizations, including propagation and backjumping. These extensions fit naturally into the framework arising from the proof.

We use some abbreviations. If $A : \mathit{Assign(Vars, Valset)}$, let *Result(A)* be

$$\begin{array}{l}
(\exists A' : \mathit{Assign(Varset, Valset)}\ .A \subseteq A' \wedge P(A')) \\
\vee\ (\exists CS \subseteq \mathit{Vars}.\forall A' : \mathit{Assign(Varset, Valset)}\ . \\
\quad\quad P(A') \rightarrow \exists v_0 \in CS.\, \mathit{Val_of}\,(v_0, A') \neq \mathit{Val_of}\,(v_0, A))
\end{array}$$

Result(A) denotes both the formula above and the type of the result of applying the search procedure to the partial assignment A – it returns either a full assignment extending A and satisfying P, or it returns a conflict set. We also abbreviate $\exists v_0 \in CS.\, \mathit{Val\_of}\,(v_0, A') \neq \mathit{Val\_of}\,(v_0, A))$ to $\mathit{Conflict}(CS, A', A)$ in order to have a concise notation for the fact that CS describes why the assignment A cannot be extended to a satisfying assignment A'.

Proof of theorem. The proof of the theorem corresponds to the function `test`. Given a partial assignment A, `test` applied to A returns a element of $\mathit{Result(A)}$.

The proof is by induction on the size of the set of unassigned variables, $\mathit{Varset} - \mathit{Vars}$. The base case is when this set is empty. Then the assignment A assigns a value to every variable. We assume that we have a lemma `check-full` which, given a full assignment A, proves $\mathit{Result(A)}$. Computationally, the lemma `check-full` is a function which takes a full assignment A as an argument and returns either A (since A is the only assignment extending A) along with evidence that $P(A)$ is true or returns a conflict set CS, along with a proof that CS really is a conflict set. If the proofs are irrelevant to remaining computation, they need not be returned as objects, but there still remains an obligation on the function `check-full` that if it returns an assignment then P holds for that assignment, and if it returns a conflict set then the set has the specified property. Note that if P does not hold for a full assignment A, then Varset is a valid conflict set.

For the inductive case of this first induction, we assume we have a nonempty set of unset variables, $\mathit{Varsleft}$. We have the following as an inductive hypothesis:

$$IH1 : \forall s \subset \mathit{Varsleft}. \forall A' : \mathit{Assign(Varset} - s, \mathit{Valset)} . \mathit{Result}(A')$$

and we must prove $\forall A : \mathit{Assign(Varset} - \mathit{Varsleft}, \mathit{Valset)} . \mathit{Result(A)}$. In other words, given an assignment $A : \mathit{Assign(Varset} - \mathit{Varsleft}, \mathit{Valset)}$ we need to construct either an assignment extending A or a conflict set for A.

Computationally, the inductive hypothesis is a function from assignments A' to $\mathit{Result}(A')$, where $A' : \mathit{Assign(Varset} - s, \mathit{Valset)}$ and s is a subset of $\mathit{Varsleft}$. Given $A : \mathit{Assign(Varset} - \mathit{Varsleft}, \mathit{Valset)}$, we can apply this function to any assignment extending A. Since a call to the inductive hypothesis corresponds to a recursive call, in the code we simply call the function `test` recursively.

We construct $\mathit{Result(A)}$ by trying the possible extensions of A. We choose a variable in $\mathit{Varsleft}$. Computationally, this is the application of the function `p-choose-var`. We now prove the result for A by using the following lemma:

Lemma 3. *Given* $A : \mathit{Assign(Varset} - \mathit{Varsleft}, \mathit{Valset)}$ *and* $v \in \mathit{Varsleft}$,

$$\begin{array}{l} \forall \mathit{Vals} \subseteq \mathit{Valset} \\ \quad \exists A' : \mathit{Assign(Varset, Valset)} . A \subseteq A' \wedge P(A') \wedge \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \\ \vee\ \exists CS \subseteq \mathit{Varset} - \mathit{Varsleft}. \forall A' : \mathit{Assign(Varset, Valset)} . \\ \qquad \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \rightarrow P(A') \rightarrow \mathit{Conflict}(CS, A`, A) \end{array}$$

This lemma is represented computationally by the function `enumerate-domain`. The lemma is proved by induction on the size of the set Vals. Given the lemma, we can prove $\mathit{Result(A)}$ by applying the lemma with $\mathit{Vals} = \mathit{Valset}$.

The base case of the induction is when Vals is empty. There is no full assignment extending A which gives v a value in $\emptyset$, so we must have a conflict set CS. The property which CS must satisfy is trivial, since $\mathit{Val\_of}\,(v, A') \in \mathit{Vals}$ will always be false, so the empty set is acceptable for CS.

In the inductive case, we have a second inductive hypothesis:

$$\begin{aligned}&IH2(vs_0) : \forall vs \subset \mathit{Vals}.\\ &\quad \exists A' : \mathit{Assign(Varset, Valset)}\ .A \subseteq A' \wedge P(A') \wedge \mathit{Val\_of}\,(v, A') \in vs\\ &\vee \exists CS \subseteq \mathit{Varset} - \mathit{Varsleft}. \forall A' : \mathit{Assign(Varset, Valset)}\ .\\ &\qquad \mathit{Val\_of}\,(v, A') \in vs \rightarrow P(A') \rightarrow \mathit{Conflict(CS, A', A)}\end{aligned}$$

We then wish to prove

$$\begin{aligned}&\exists A' : \mathit{Assign(Varset, Valset)}\ .A \subseteq A' \wedge P(A') \wedge \mathit{Val\_of}\,(v, A') \in \mathit{Vals}\\ &\vee \exists CS \subseteq \mathit{Varset} - \mathit{Varsleft}. \forall A' : \mathit{Assign(Varset, Valset)}\ .\\ &\qquad \mathit{Val\_of}\,(v, A') \in \mathit{Vals} \rightarrow P(A') \rightarrow \mathit{Conflict(CS, A', A)}\end{aligned}$$

Here, *Vals* represents the set of values which have yet to be tried as values of the variable v. Thus, to apply the second inductive hypothesis (corresponding to a recursive call to `enumerate-domain`), we must reduce this set. So choose a value n in *Vals*. Computationally, this is the function `p-val`. Let $A_{v=n}$ be the assignment A extended with v equal to n.

We could immediately apply the first inductive hypothesis, which computationally is a recursive call to `test`. This corresponds to simple backtracking search. However, we wish to allow for early detection and pruning of impossible partial assignments. Thus, we assume we have a function `check` which, when applied to $A_{v=n}$, returns one of two things. If $A_{v=n}$ is consistent, `check` returns some kind of success token. If $A_{v=n}$ is already inconsistent, `check` returns a conflict set CS for $A_{v=n}$: a subset of $\mathit{Varset} - \mathit{Varsleft}$ such that

$$\forall A' : \mathit{Assign(Varset, Valset)}\ .P(A') \rightarrow \mathit{Conflict}(CS, A', A_{v=n})$$

If `check` does not return a conflict set for the partial assignment, we must try partial assignments extending $A_{v=n}$. Since $A \subset A_{v=n}$, we apply the first inductive hypothesis, calling `test` recursively. The result of applying `test` is of type $\mathit{Result}(A_{v=n})$; that is, either an assignment A' extending $A_{v=n}$ such that $P(A')$, or a conflict set for $A_{v=n}$, as described above. If we have a solution, we are done.

If not, then we have a conflict set (call it $CS1$) for $A_{v=n}$, either derived from `check` or from the recursive call to `test`. Now we remove n from *Vals* and apply the second inductive hypothesis (via a recursive call to `enumerate-domain`) with the set $\mathit{Vals} - \{n\}$. If the result is an $A' : \mathit{Assign(Varset, Valset)}$ such that $A \subseteq A'$ and $P(A')$, then we are done. Otherwise, we have a second conflict set $CS2 \subseteq \mathit{Varset} - \mathit{Varsleft}$ satisfying

$$\begin{aligned}&\forall A' : \mathit{Assign(Varset, Valset)}\ .(\mathit{Val\_of}\,(v, A') \in \mathit{Vals} - \{n\}) \rightarrow\\ &\qquad P(A') \rightarrow \mathit{Conflict(CS, A', A)}\end{aligned}$$

Now let $CS = CS1 \cup CS2$. Then $CS \subseteq \mathit{Varset} - \mathit{Varsleft}$, and it is easy to check that CS satisfies

$$\begin{aligned}&\forall A' : \mathit{Assign(Varset, Valset)}\ .(\mathit{Val\_of}\,(v, A') \in \mathit{Vals}) \rightarrow\\ &\qquad P(A') \rightarrow \mathit{Conflict(CS, A', A)}\end{aligned}$$

This finishes the inductive case of the lemma, and thus the whole proof.

Modifications to the proof

As presented, this proof corresponds to a fairly simple backtracking search procedure. It has the potential for pruning (via the `check` function), variable ordering heuristics (via `p-choose-var`) and value ordering heuristics (via `p-val`). These functions will, in general, be problem-specific, so the proof cannot describe them in detail. However, the proof does describe minimum requirements for these functions which ensure correctness of the resulting code.

The proof can be extended to describe techniques like conflict-directed backjumping and propagation. The latter requires very little modification to the proof. Note that the only properties we have assumed about assignments are that we can order them (by prefix or subset) so we can say $A' \subseteq A$, and that we can look up the value of a variable v in an assignment A using $\mathit{Val\_of}\,(v, A)$. The actual type of an assignment may be much more complicated – it may, for instance, include information about eliminated values of future variables. Such information can be computed by the `check` function, and returned to the main function by having `check` take an assignment structure A and return a (possibly modified) assignment structure A'. The proof only requires that the values of the variables in *Vars* be the same in A and A'.

To introduce the information which may have been computed in this way, we add a step at the beginning of the second induction. Instead of beginning with the whole set of values *Valset*, we assume we have a lemma which, given a partial assignment A and a variable v returns a set *Vals* of values to be tried together with conflict set CS satisfying[4]

$$\forall A' : \mathit{Assign(Varset, Valset)}\ .(\mathit{Val\_of}\,(v, A') \in \mathit{Valset} - \mathit{Vals}) \rightarrow \\ P(A') \rightarrow \mathit{Conflict}(CS, A', A)$$

We cannot eliminate values for no reason; we must still be able to produce the evidence, in the form of a conflict set, that these values are impossible.

If a technique such as forward checking reduces the domain of a variable to a singleton set, the value can be committed to. This is often called 'propagation', and can be the key to the success of search algorithms. For example in Davis-Putnam (DP), this is 'unit propagation'. In CSP's, propagation happens implicitly if forward checking is used with the FF (smallest domain first) heuristic. Our proof allows for propagation without change, because `p-choose-var` is at liberty to pick a variable with domain size 1 if it exists, as long as `p-var-cs` returns an appropriate conflict set. Our code, however, contains a special function `propagate` which is called if commitment is possible: `propagate` is simply a special case of `enumerate-domain` when the domain is known to be of size 1. We have included it for pedagogical purposes to clarify the distinction between propagation and heuristic choice. The extension of our proof for this changed situation would be straightforward.

The extension of the proof to include backjumping is somewhat more subtle. In the code below, we implement backjumping using the Scheme opera-

[4] In our code the conflict set is returned by `p-var-cs` and the remaining values enumerated by `p-domain-rest`.

tor `call/cc`, or `call-with-current-continuation`. When `(call/cc (lambda (k) ...))` is evaluated, `k` becomes bound to the current continuation; in other words, `k` represents the rest of the computation, apart from that remaining in the body of the `call/cc`. When `k` is applied to an argument, the computation returns immediately to the context which existed when `k` was created, and the argument passed to `k` is used in the place of the `call/cc (lambda (k) ...)` term. Thus, `call/cc` is essentially a functional goto; it allows control to jump immediately to another part of the program.

In this program, we use `call/cc` to create continuations which represent points to which the search might backjump. Backjumping occurs when a conflict set is found which eliminates more of the search tree than its local situation requires. A continuation is created whenever a variable is set to create a partial assignment. Should we discover, deep in the search tree, that this partial assignment is inconsistent, we return immediately to this point by applying the continuation to the evidence of inconsistency, in the form of a conflict set.

To get this computational behaviour from the proof, we use the fact that `call/cc` can be given the type $((\alpha \rightarrow \perp) \rightarrow \alpha) \rightarrow \alpha$ for any type α [11, 17, 16, 23]. This corresponds to a form a proof by contradiction; if, from the assumption that α is false, we can prove α, then we have a contradiction so α must be true. This form of reasoning is not strictly constructive, but in this case we still have a computational meaning for it. Although a constructive formal system like Lego does not permit classical reasoning, we can add it by adding an assumption to the theorem that `call/cc` has type $((\alpha \rightarrow \perp) \rightarrow \alpha) \rightarrow \alpha$. In the proof of the theorem, we add an extra assumption of the form $\forall A_0 \subset A. \neg \mathit{Result}(A_0)$. These assumptions are satisfied by the creation of continuations with `call/cc`. When we produce a more general conflict set than is required and wish to backjump, we use the appropriate continuation[5] to return immediately to the right stage in the computation. Logically, this step is an unnecessarily roundabout proof of $\mathit{Result}(A)$. If the conflict set CS is really a valid result for some previous partial assignment A_0, then we use the assumption $\neg \mathit{Result}(A_0)$ to get a contradiction and hence to conclude anything, and in particular $\mathit{Result}(A)$. However, when the continuation corresponding to the assumption $\neg \mathit{Result}(A_0)$ is applied, the computation returns to the point where A_0 is being tried by `enumerate-domain`, the conflict set CS is now treated as a conflict set for A_0, and computation continues from that point.

This logical treatment of the control ensures that backjumping is sound; we can only backjump when we have evidence that there is no solution in the part of the search tree we are pruning. The proof corresponding to a backjumping algorithm is more complex than the proof corresponding to a simple backtracking algorithm; since the program is more complex as well this should not be surprising. It is perhaps surprising that the modifications necessary are not even more complex. Apart from the assumptions mentioned above, we only add a

[5] I.e. the continuation associated with the most recently assigned variable in the conflict set. In our code we assume this is returned by `cs-deepest`. We have not proved correct this and a number of other functions implementing abstract data types.

function/lemma `backjump` which takes a conflict set and applies the continuation corresponding to the deepest conflict, and a data structure which stores the continuations as they are created.

3 Applications: Existing and New Algorithms

It is clear that the above results can be applied to many problem domains. However, the generality of our approach extends not only to search for different problems, but to different search algorithms. Provided code is supplied for the auxiliary functions which meets the obligations needed for our proof, a correct search algorithm will result. However, different search methods can be implemented, depending on exactly how the obligations are met. For example, `check` can perform more or less complicated calculations at each stage. Different amounts of checking will result in different amounts of pruning and propagation. Similar comments apply to other auxiliary functions, allowing for example for variable and value ordering heuristics.

To illustrate how a variety of algorithms can be developed, we give sketches of how to implement algorithms for SAT, CSP, and the Hamiltonian Circuit (HC) problem. While the sketches are not proofs, they could be expanded to give full proofs of the correctness of the relevant algorithms: in some cases this remains important future work. Because of our framework, full proofs could be given just by proving the relevant auxiliary functions correctly implemented. Such proofs should not in general be difficult, yet the result would be correctness proofs of algorithms never formally proved correct (DP with CBJ for SAT or MAC-CBJ for the CSP) or never even previously described (FC-CBJ for HC.)

In describing implementations, we have not focussed on efficiency issues. Reasonably efficient implementations can be based on the following sketches, because our Scheme code allows the auxiliary function to manipulate and update a problem data structure. This enables implementations to cache computations in this data structure to maintain, for example, a data structure for the heuristic values of unassigned variables, rather than recomputing these after every instantiation.

Application: Satisfiability

We consider SAT problems in clausal form. A literal is a negated (negative) or unnegated (positive) variable. A clause is a disjunction of literals, and the whole problem a conjunction of the clauses in the problem. Variables take the value true or false. An assignment satisfies the problem if every clause contains a literal satisfied by the assignment, a positive literal being satisfied by the value true and a negative literal by the value false.

The standard algorithm for SAT is the Davis-Putnam (DP) algorithm [6, 5] though we describe it here (as is often done) without pure literal deletion. However we do present it with both a variable- and a value- ordering heuristic.

DP: Variables are, as would be expected, the variables in the problem. If all clauses have been satisfied by a partial assignment we can stop searching (`p-domain-end?`)[6] and the problem has been solved (so `check-full` need do

[6] To ease reference to our code we mention in passing relevant Scheme functions when

nothing.) A partial assignment is unsatisfiable if it makes every literal false in some clause: in this situation a valid conflict set is every variable in the partial assignment (`check`). One reasonable variable-ordering heuristic is to consider only the clauses not yet satisfied and with fewest unassigned literals, and then pick the first variable occurring in the first such clause in the problem (`p-choose-var`). A value-ordering heuristic is to first give the variable the value true or false according to whether it was in a positive or negative literal (`p-val`). We can commit (`p-commit?`) if there is a unit clause under the current partial assignment, i.e. any clause with exactly one unassigned literal. The variable to set (`p-commit-var`) is the remaining variable to the value it has in the unit clause (`p-commit-val`). In this situation, the conflict set in evidence of the commitment is again all variables in the partial assignment (`p-commit-var-cs`).

We implemented the auxiliary functions as described above in Scheme, and tested our code against special-purpose DP code with the same heuristics written independently in Common Lisp.[7] We tested 20 random 3-SAT problems (described for example in [1]) with from 100 to 500 clauses in steps of 1 clause. In each of these 8020 tests both implementations found identical solutions in identical numbers of nodes searched. Such experiments provide confidence that we have implemented the service functions correctly.

DP-CBJ: To implement conflict-directed backjumping in our framework is now easy. We need only change the way conflict sets are returned. When we find an unsatisfied clause, a valid conflict set is just the set of variables occurring in that clause (`check`). When we find a unit clause, a valid conflict set is just the set of assigned variables occurring in the clause (`p-var-cs`).

The algorithm DP-CBJ has been reported by Bayardo and Schrag [1]. They have shown that it can outperform the best implementations of DP without CBJ and compete with the best local search methods for SAT [2]. However we are not aware of a formal correctness proof of the algorithm: extending our sketch above into one should be straightforward.

Application: Binary Constraint Satisfaction

Conflict-directed backjumping was first described in the context of binary CSPs [19] so it is natural to apply our framework to that domain. A problem consists of a number of variables each of which can take a value from a finite domain. Each constraint acts on two variables, and rules out a subset of the possible pairs of values from the two domains. An assignment of variables to values is a solution if there are no conflicts with any of the constraints.

CBJ: If all variables have been set we can stop searching (`p-domain-end?`), and the problem has been solved if all constraints are satisfied (`check-full`). A partial assignment is unsatisfiable if the pair of values of the current variable and any other variable is ruled out by some constraint: in this situation a valid conflict set is the current variable and the other variable in the conflict (`check`). Following Prosser [19] we return the conflict with the shallowest variable in the search tree,

describing our implementations.

[7] We thank Toby Walsh for supplying this code. Our code runs considerably slower than Walsh's through using lists for all data structures.

if there is more than one conflict. We pick the smallest unassigned variable in the lexicographic order (`p-choose-var`). Similarly a trivial value-ordering heuristic is to return each value in the variable's domain in lexicographic order (`p-val`), and we consider all values so do not need a conflict set for any ruled out values (`p-var-cs`). We perform no forward checking (`p-commit?` always returns false.)

We implemented the auxiliary functions as described above in Scheme, and tested our code against special-purpose CBJ code written independently, also in Scheme.[8] When run on problem generated randomly in a standard way (see for example [21]) with 10 variables, 10 values, 23 constraints, and constraint tightness varying from 0.01 to 0.99 in steps of 0.1, the two implementations produced identical results in the solutions found, number of nodes searched, number of checks performed and conflicts found, on 100 problems at each point.

As well as backjumping techniques such as CBJ, propagation techniques such as forward checking are often used, and indeed it would not be difficult to implement FC-CBJ in our framework. A more sophisticated propagation technique is 'maintaining arc consistency' (MAC) [22]. MAC has been combined with CBJ to yield MAC-CBJ [20] and this algorithm has been extensively studied empirically [10]. However we are not aware of a correctness proof of MAC-CBJ. It is therefore particularly interesting to sketch how MAC-CBJ can be implemented in our framework, and therefore how a proof of its correctness might be given.

Forward checking removes values from the domains of future variables which are inconsistent with the assigned value of the current variable. MAC is based on the following observation. If some given value in the domain of one variable conflicts with every value in the current domain of a second variable, we can remove the given value from the domain of the first variable: the given value is said to be 'unsupported' by the second variable.

The key to FC and MAC is the removal of values from variables' domains. Our framework requires evidence in the form of a conflict set for any such removal. But, especially in the case of MAC, construction and maintenance of such conflict sets requires considerable care. Fortunately, this care can be exercised by our search code, leaving only the initial construction of conflict sets to special purpose code. To do this, it is convenient to introduce additional variables representing the removal of a given value from a given variable's domain: such variables can be true or false.[9] When the auxiliary functions detect a value removal, they can report this to the search algorithm by committing the value of an additional variable to true, together with a conflict set representing the reason why the value false is impossible. Later, when the relevant natural variable is chosen, `p-var-cs` can return the reduced domain together with a conflict set consisting of the additional variables representing all the value removals. This idea makes it comparatively straightforward to implement MAC-CBJ.

MAC-CBJ: If all natural variables have been set (`p-domain-end?`) we can

---

[8] We thank Patrick Prosser for supplying the independent code. The two implementations are of comparable speed. Our code happens to run slightly faster than Prosser's.

[9] The apparent increase in the number of variables is not a representational problem, as we can name them via some convention, for example `(removal var8 red)`.

stop searching, and the problem has been solved if all constraints are satisfied (`check-full`). A partial assignment is unsatisfiable if the domain of some unassigned variable is empty, given the value removals made so far. In this situation a valid conflict set is simply the set of additional variables representing the value removals for the variable with empty domain (`check`), since to find a solution we must restore at least one value to the domain, i.e. change one of the additional variables from true to false. A well-known variable-ordering heuristic is to pick the variable with smallest domain, i.e. least unremoved values (`p-choose-var`) and we continue to consider remaining values in lexicographic order (`p-val`). The conflict set for removed values is simply the set of additional variables representing any value removals (`p-var-cs`). We can make a commitment if some value of an unassigned variable conflicts with the chosen value of the current natural variable (`p-commit?`), committing the relevant additional variable representing the removal (`p-commit-var`) which we set to true (`p-commit-val`), the conflict being with the current natural variable (`p-var-cs`). We can also make a commitment whenever a given value of a variable is unsupported by a second variable (`p-commit?`). We can commit to removing the given value of the first variable (`p-commit-var`, `p-commit-val`). The removal depends on the current domain of the second variable. Thus a valid conflict set is the set of additional variables representing all value removals so far from the domain of the second variable (`p-commit-var-cs`). An additional point is that we must look for value removals as a preprocessing step, to establish arc consistency. The first value removal has an empty conflict set: in other words, if this additional variable is ever backjumped to, the problem is insoluble.

Application: Hamiltonian Circuit

The Hamiltonian Circuit problem is to visit all nodes in a graph exactly once and returning the starting point, while only traversing edges that appear in the graph. The problem is NP-complete and a phase transition in solubility has been observed [3, 7]. Algorithms analogous to forward checking have been given, for example by Martello [15], but we are not aware of CBJ having been described for this problem. In our framework it is more natural to implement CBJ rather than chronological backtracking. We consider the problem for directed graphs. We sketch an implementation of the analog of FC-CBJ for this problem.

To apply our technique to this problem, we need to specify the variables and values and describe when and how conflict sets are produced for partial assignments. Variables are nodes in the graph, and possible values for each node n are the nodes accessible by out-arcs from n. When n_1 is assigned n_2, n_2 can be removed from the domains of all future variables; the conflict set which justifies this removal is $\{n_1\}$. Thus we have forward checking for Hamiltonian circuits. A commitment can be made if there is a node with only one out-arc (or only one in-arc) remaining (`p-commit-var`, `p-commit-val`). The conflict set justifying this is the set of variables which caused any other out-arcs (or in-arcs) to be removed (`p-commit-var-cs`). Similarly, when choosing a node heuristically (`p-choose-var`), we must construct the conflict set of variables which caused any out-arcs from this node to be removed (`p-var-cs`).

There are three cases in which a partial assignment is inconsistent, and a conflict set is produced by `check`: there is a node with no in-arcs remaining, there is a node with no out-arcs remaining, or there is a cycle which is not a circuit. If some node n has no in-arcs, a conflict set is the set of nodes in the original graph which had out-arcs to n, since all of these nodes must have been assigned other values. If some node n has no out-arcs, a conflict set is the set of nodes in the original graph which had out-arcs to nodes to which n also had an out-arc. Finally, if there is a cycle which is not a circuit, the conflict set is the set of nodes in that cycle.

For efficiency in an implementation, when an arc from n_1 to n_2 is added to the circuit, the nodes n_1 and n_2 can be collapsed, and the irrelevant arcs deleted, following the description given for example by Martello. However, this is invisible to the search functions – the interface can be written so that `test` and `enumerate-domain` see only values in the original problem.

4 Related Work

Following Prosser's introduction of conflict-directed backjumping (CBJ) [19], Ginsberg [8] and Kondrak & van Beek [13] have given proofs of the correctness of CBJ and also related the numbers of nodes searched by different algorithms. The significant advance of our work is in its underlying basis in formal semantics and in its generality. Ginsberg gave proofs of pseudo-code written in English, and Kondrak & van Beek of Prosser's Pascal-like pseudo-code: thus neither proof applies to code for which formal semantics exists. Our results are very general because they apply to a wide variety of search algorithms, and a wide variety of problem classes, all obtainable from the Scheme code we have presented by implementing suitable service functions.

5 Conclusions

We have shown the way that correct implementations of important search algorithms can be achieved in great generality. We have shown this by sketching implementations within our framework of algorithms for diverse problems such as constraint satisfaction, satisfiability, and Hamiltonian circuit. Sophisticated algorithms such as conflict-directed backjumping (CBJ) for each can be implemented without detailed knowledge of the working of the backjumping process.

The importance of our work lies in its generality and its ability to deliver correctness proofs of algorithms. In the future, we hope to use this to extend our sketches to full correctness proofs, most especially for important algorithms that have not to our knowledge been formally proved correct, for example MAC-CBJ for the constraint satisfaction problem and Davis-Putnam with CBJ for satisfiability. While our framework already implicitly allows for techniques such as forward checking, we also hope to expand our general proof and code to incorporate these explicitly, and therefore to further ease the speedy development of good search algorithms in new domains.

References

1. R.J. Bayardo and R.C. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *CP-96*, pages 46–60. Springer, 1996.
2. R.J. Bayardo and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings, AAAI-97*, 1997.
3. P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th IJCAI*, pages 331–337, 1991.
4. R. Constable et al. *Implementing Mathematics with The Nuprl Development System.* Prentice-Hall, New Jersey, 1986.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comms. ACM*, 5:394–397, 1962.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Association for Computing Machinery*, 7:201–215, 1960.
7. J. Frank and C. Martel. Phase transitions in random graphs. In *Proceedings, Workshop on Studying and Solving Really Hard Problems, CP-95*, 1995.
8. M.L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1:25–46, 1993.
9. J. Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.
10. S.A. Grant and B.M. Smith. The phase transition behaviour of maintaining arc consistency. In *Proceedings of ECAI-96*, pages 175–179, 1996.
11. T. Griffin. A formulas-as-types notion of control. In *Proc. of the Seventeeth Annual Symp. on Principles of Programming Languages*, pages 47–58, 1990.
12. W. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.
13. G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 89:365–387, 1997.
14. Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science.* Oxford University Press, 1994.
15. S. Martello. An enumerative algorithm for finding Hamiltonian circuits in a directed graph. *ACM Transactions on Mathematical Software*, 9:131–138, 1983.
16. C. Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Dept. of Computer Science, 1990. (TR 89-1151).
17. C. Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1991.
18. R. Pollack. *The Theory of Lego.* PhD thesis, University of Edinburgh, 1995. Available as report ECS-LFCS-95-323.
19. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
20. P. Prosser. Maintaining arc-consistency with conflict-directed backjumping. Res. rep. 95-177, Dept. of Computer Science, University of Strathclyde, UK, 1995.
21. P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:127–154, 1996.
22. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of ECAI-94*, pages 125–129, 1994.
23. J. Underwood. *Aspects of the Computational Content of Proofs.* PhD thesis, Cornell University, 1994.

Appendix: Scheme Code

```
;;; Written by Judith Underwood and Ian Gent, 1997
;;; This code may be copied and used freely, with due credit, but we provide no warranties or guarantees of any kind.

(define (search check check-full)
  (let ((res (call-with-current-continuation
              (lambda (terminate)
                      (test (make-data) (make-problem) 0 check check-full terminate)))))
        res))

(define (test data problem depth check check-full terminate)
  (cond ((p-end? problem)
         (check-full data problem))
        ((p-commit? problem)
         (let* ((new-problem (p-commit-var problem))
                (var (p-var new-problem))
                (result (propagate
                         data
                         (make-result-fail (p-commit-var-cs data new-problem))
                         var new-problem (+ 1 depth)
                         check check-full terminate)))
               (if (result-succeed? result)
                   (solved data result terminate)
                   result)))
        (t
         (let* ((new-problem (p-choose-var problem))
                (var (p-var new-problem))
                (result (enumerate-domain
                        data
                        (make-result-fail (p-var-cs data new-problem))
                        var new-problem (+ 1 depth)
                        check check-full terminate))  )
               (if (result-succeed? result)
                   (solved data result terminate)
                   result)))))

(define (propagate data result-so-far var problem depth check check-full terminate)
     (let* ((new-problem (p-domain-commit problem))
            (assign (make-assign var (p-commit-val new-problem)))
            (this-result
             (call-with-current-continuation
              (lambda (k)
                     (let* ((new-data (data-add (make-datum assign k depth result-so-far) data))
                            (checkres (check new-data problem))
                            (result (if (result-succeed? checkres)
                                        (test new-data new-problem depth check check-full terminate)
                                        checkres)))
                           (if (result-succeed? result)
                               (solved new-data result terminate)
                               result))))))
           (backjump data
                     (result-cleanup (result-merge result-so-far this-result) depth)
                     terminate)))

(define (enumerate-domain data result-so-far var problem depth check check-full terminate)
 (if (p-domain-end? problem)
     (backjump data (result-cleanup result-so-far depth) terminate)
     (let* ((new-problem (p-domain-choose problem))
            (assign (make-assign var (p-val new-problem)))
            (this-result
             (call-with-current-continuation
              (lambda (k)
                      (let* ((new-data (data-add
                                         (make-datum assign k depth result-so-far)
                                       data))
                             (checkres (check new-data problem))
                             (result (if (result-succeed? checkres)
                                         (test new-data new-problem depth check check-full terminate)
                                         checkres)))
                             (if (result-succeed? result)
                                 (solved new-data result terminate)
                                 result))))))
          (enumerate-domain data
                            (result-merge result-so-far this-result)
                            var
                            (p-domain-rest problem)
                            depth
                            check check-full terminate))))

(define (backjump data result terminate)
        (if (null? (result-cs result))
            (terminate result)
            (let ((back (data-depth data (cs-deepest (result-cs result)))))
                 ((datum-continuation back)
                  (result-cleanup result (datum-depth back))))))

(define (solved data result terminate)
          (terminate (make-result-success (data-solution data))))
```