Modular Lazy Search for Constraint Satisfaction Problems*

Thomas Nordin   Andrew Tolmach
Pacific Software Research Center
Oregon Graduate Institute & Portland State University
nordin@cse.ogi.edu apt@cs.pdx.edu

Abstract

We describe a unified, lazy, declarative framework for solving constraint satisfaction problems, an important subclass of combinatorial search problems. These problems are both practically significant and hard. Finding good solutions involves combining good general-purpose search algorithms with problem-specific heuristics. Conventional imperative algorithms are usually implemented and presented monolithically, which makes them hard to understand and reuse, even though new algorithms often are combinations of simpler ones. Lazy functional languages, such as Haskell, encourage modular structuring of search algorithms by separating the generation and testing of potential solutions into distinct functions communicating through an explicit, lazy intermediate data structure. But only relatively simple search algorithms have been treated in this way in the past.

Our framework uses a generic generation and pruning algorithm parameterized by a labeling function that annotates search trees with conflict sets. We show that many advanced imperative search algorithms, including backmarking, conflict-directed backjumping, and minimal forward checking, can be obtained by suitable instantiation of the labelling function. More importantly, arbitrary combinations of these algorithms can be built by simply composing their labelling functions. Our modular algorithms are as efficient as the monolithic imperative algorithms in the sense that they make the same number of consistency checks, and most of our algorithms are within a constant factor of their imperative counterparts in runtime and space usage. We believe our framework is especially well-suited for experimenting to find good combinations of algorithms for specific problems.

1 Introduction

Combinatorial search problems offer a great challenge to the academic researcher: they are of tremendous interest to commercial users, and they are often very computationally intensive to solve. Over the past several decades the AI community has responded to this challenge by producing a steady stream of improvements to generic search algorithms. There have also been numerous attempts to organize the various algorithms into standardized frameworks for comparison (e.g., [8, 6, 16]).

While the speed and cunning of the search algorithms have improved, the new algorithms are more complicated and harder to understand, even though they are often combinations of simpler standard algorithms. The problem is exacerbated by the fact that most algorithms are described by large, monolithic chunks of pseudo-code (or C code). Although it is recognized that most problems benefit from a tailor-made solution, involving a combination of existing generic and domain-specific algorithms, modularity has not been a strong point of much of the recent research. It is difficult to reuse code except via cut-and-paste. Moreover, to prove these algorithms correct we must resort to complex reasoning about their dynamic behavior.

* Work supported, in part, by the US Air Force Materiel Command under contract F19628-26-C-0161.

For example, although most of these search algorithms are conceived as varieties of "tree search," no actual tree data structures appear in their implementations; only virtual trees are present, in the form of recursive routine activation histories. Perhaps for this reason, even widely-used and well-studied algorithms often lack correctness proofs.

In the lazy functional programming world, the idea of implementing a search algorithm using modular techniques is a commonplace. The classic paper of Hughes [9] and text of Bird and Wadler [3] both give examples of search algorithms in which generation and testing of candidate solutions are separated into distinct phases, glued together using an explicit, lazy, intermediate data structure. This "generate-and-test" paradigm makes essential use of laziness to synchronize the two functions (really coroutines) in such a way that we never need to store much of the (exponential-sized) intermediate data structure at any one time. In general, the modular lazy approach can lead to algorithms that are much simpler to read, write, and modify than their imperative counterparts. However, the algorithms described in these sources are fairly elementary.

In this paper we present a lazy declarative framework for solving one important class of combinatorial search problems, namely constraint satisfaction problems (CSPs). For simplicity, we restrict our attention to binary constraint problems, and to search algorithms that use a fixed variable order; neither of these restrictions is fundamental. Our framework is based on explicit, lazy, tree structures, in which each tree node represents a *state* in the search space; a subset of the tree's leaf nodes corresponds to problem solutions. Nodes can be labeled with *conflict sets*, which record constraint violations in the corresponding states; many algorithms use these sets to *prune* subtrees that cannot contribute a solution.

Our framework is written in Haskell. We provide a small library of separate functions for generating, labeling, rearranging, pruning, and collecting solutions from trees. In particular, we describe a generic search algorithm, parameterized by a labeling function, and show that a variety of standard imperative CSP algorithms, including simple backtracking, backmarking, conflict-directed backtracking, and forward checking, can be obtained by making a suitable choice of pruning function. Using an explicit representation of the search tree allows us to think about the intermediate values and gives us new insights into more efficient algorithms. As in recent work on functional data structures[10, 14], we found that recasting imperative algorithms into a declarative lazy setting casts new light on the fundamental algorithmic ideas. In particular, it is easy to see how to combine our algorithms, simply by composing their labeling functions, and to see that the result will be correct.

Since the whole point of improving search algorithms is to be able to solve larger problems faster, we must obviously be concerned with the performance of our lazy algorithms. Our experiments show that lazy, modular Haskell code is several times slower than strict, manually integrated Haskell code; moreover, even the latter can be an order of magnitude slower than highly optimized C code. However, since search times often explode exponentially, even slowdowns of one or two orders of magnitude have little effect on the size of problem we can solve within a fixed time bound. All our algorithms and their combinations are fast enough for experiments that have been interesting to researchers in the past; for example we are able to reproduce parts of the tables in [2, 11]. More importantly, our implementations are fast enough to allow experimentation with different combinations of algorithms on problems of realistic size. For such experiments, CPU time is often not an ideal comparison metric, since it is difficult to compare numbers obtained from different implementations on different systems. A widely used alternative metric is the number of consistency checks performed by the algorithm.

The remainder of the paper is organized as follows. Section 2 describes our problem domain and Section 3 gives a Haskell specification for it. Section 4 describes simple tree-based backtracking search. Section 5 introduces conflict sets and our generic search algorithm, and recasts backtracking search in that framework. Section 6 briefly discusses search heuristics. Sections 7, 8, and 9 describe more sophisticated algorithms, and Section 10 discusses their combination. Section 11 summarizes performance results, Section 12 describes related work, and Section 13 concludes.

The reader is assumed to have a working knowledge of functional programming, and some familiarity with laziness. Peculiarities of Haskell syntax will be explained as they arise. All the code examples in this paper are available on the World Wide Web at `http://www.cs.pdx.edu/~apt/CSP.hs`.

2 Binary Constraint Satisfaction Problems

A *binary constraint satisfaction problem* is:

- a set of variables $V = \{v_1, v_2, \ldots v_n\}$;
- for each variable v_i, a finite set D_i of possible values (its *domain*);
- and a set C of pairwise *constraints* between variables that restrict the values that they can take on simultaneously.

Each constraint is a relation on two named variables, i.e., a triple (i, j, R) where $R \subseteq D_i \times D_j$.

An *assignment* v_i:=x_i associates a variable v_i to some value $x_i \in D_i$. A *state* is a collection of assignments for a subset of V. A state $\{\ldots v_i$:=$x_i, \ldots v_j$:=$x_j, \ldots\}$ *satisfies* a constraint (i, j, R) if $(x_i, x_j) \in R$. A state is *consistent* if it satisfies every constraint on its variables, i.e., if for every pair of assignments v_j:=x_j, v_k:=x_k in the state, and every matching constraint (j, k, R) in C, $(x_j, x_k) \in R$. A state is *complete* if it assigns all the variables of V; otherwise it is *partial*. A *solution* to a CSP is any complete consistent state. For some problems we want to calculate all solutions, but for many we only wish to find the first solution as quickly as possible.

In this paper, we fix the variable order $v_1, v_2, \ldots v_n$, i.e., we consider only states such that if v_i is in the state so is v_j for all $j < i$. We define the *level* of a variable v_i to be i and the level of a state to be the maximum of its variables' levels. To simplify the presentation, we further assume that all domains have the same size m and that their values are represented by integers in the set $\{1, 2, \ldots, m\}$.

A naive approach to solving a CSP is to enumerate all possible complete states and then check each in turn for consistency. In a binary CSP, consistency of a state can be determined by performing *consistency checks* on each pair of assignments in the state, until an *inconsistent pair* of variables is detected, or all pairs have been checked. Following the conventions of the search literature, we use the number of consistency checks as a key measure of execution code, although it is not necessarily an accurate measure unless each check can be performed in unit time, which is not the case for all problems.

3 CSPs in Haskell

Figure 1 gives a Haskell framework for describing CSP problems and an implementation of a naive solver. An assignment is constructed using the infix constructor `:=`. A CSP is modeled as a record containing the number of variables, `vars`, the size of their domain, `vals`, and a constraint oracle, `rel`. We represent the oracle as a Haskell function taking two assignments and returning `False` iff the assignments violate some constraint. This function can be implemented by a four-dimensional array of booleans or by a mathematical formula.

We present the solver in the standard "lazy pipeline" style that separates generation of candidate solutions (here the set of all complete states) from consistency testing. States are represented as lists of assignments sorted in decreasing order by variable number. Although this code appears to produce a huge intermediate list data structure `candidates`, lazy evaluation insures that list elements are generated only on demand, and elements that fail the filter in `test` can be immediately garbage collected. Similarly, although `inconsistencies` appears to build a list of *all* inconsistent variable pairs in the state[1], `consistent`

[1]This function uses a Haskell *list comprehension*, which is similar to a familiar set comprehension: this one builds a list of pairs of variable levels such that the corresponding assignments are drawn from the current state and are in conflict according to `rel`.

```
type Var = Int
type Value = Int

data Assign = Var := Value deriving (Eq, Ord, Show)

type Relation = Assign -> Assign -> Bool

data CSP = CSP { vars, vals :: Int, rel :: Relation }

type State = [Assign]

level :: Assign -> Var
level (var := val) = var

value :: Assign -> Value
value (var := val) = val

maxLevel :: State -> Var
maxLevel [] = 0
maxLevel ((var := val):_) = var

complete :: CSP -> State -> Bool
complete CSP{vars} s = maxLevel s == vars

generate :: CSP -> [State]
generate CSP{vals,vars} = g vars
  where g 0 = [[]]
        g var = [ (var := val):st | val <- [1..vals], st <- g (var-1) ]

inconsistencies :: CSP -> State -> [(Var,Var)]
inconsistencies CSP{rel} as =
  [ (level a, level b) | a <- as, b <- reverse as, a > b, not (rel a b) ]

consistent :: CSP -> State -> Bool
consistent csp = null . (inconsistencies csp)

test :: CSP -> [State] -> [State]
test csp = filter (consistent csp)

solver :: CSP -> [State]
solver csp  = test csp candidates
  where candidates = generate csp

queens :: Int -> CSP
queens n = CSP {vars = n, vals = n, rel = safe}
  where safe (i := m) (j := n) = (m /= n) && abs (i - j) /= abs (m - n)
```

Figure 1: A formulation of CSPs in Haskell.

```
data Tree a = Node a [Tree a]

label :: Tree a -> a
label (Node lab _) = lab

type Transform a b = Tree a -> Tree b

mapTree  :: (a -> b) -> Transform a b
mapTree f (Node a cs) = Node (f a) (map (mapTree f) cs)

foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree f (Node a cs) = f a (map (foldTree f) cs)

filterTree :: (a -> Bool) -> Transform a a
filterTree p = foldTree f
  where f a cs = Node a (filter (p . label) cs)

prune :: (a -> Bool) -> Transform a a
prune p = filterTree (not . p)

leaves :: Tree a -> [a]
leaves (Node leaf []) = [leaf]
leaves (Node _ cs) = concat (map leaves cs)

initTree :: (a -> [a]) -> a -> Tree a
initTree f a = Node a (map (initTree f) (f a))
```

Figure 2: Trees in Haskell.

actually demands only the head of the list (to check whether the list is `null`). Thus the solver actually calculates only the *earliest inconsistent pair* of variables for each state. Finally, although the solver returns a list of all solutions if demanded, it can be used to obtain just the first solution (and do no further computation) by asking for just the head of the result. Although the code thus uses much less space than a strict reading would suggest, this solver is still extremely inefficient because it duplicates work, but it is useful to illustrate lazy coding style and as a specification for the more sophisticated solvers we introduce below.

A simple problem useful for illustrating different search strategies is the n-queens problem, that is, trying to put n queens on a $n \times n$ chess board such that no queen is threatening another. using the standard optimization that we only try to place one queen in each column [13]. Given the definition of `queens`, we can apply the general-purpose CSP machinery to solve it; for example, the expression `solver (queens 5)` generates a list of solutions to the 5-queens problem.

4 Backtracking and Tree Search

The most obvious defect of the naive solver is that it can duplicate a tremendous amount of work by repeatedly checking the consistency of assignments that are common to many complete states. We say state S' *extends* state S if it contains all the assignments of S together with zero or more additional assignments. A fundamental fact about CSP's is that no extension to an inconsistent state can ever be consistent, so there is no point in searching such an extension for a solution. This observation immediately suggests a better solver algorithm. A *backtracking* solver searches for solutions by constructing and checking *partial* states, beginning with the empty state and extending with one assignment at a time. Whenever the solver discovers an inconsistent state, it immediately *backtracks* to try a different assignment, thus avoiding the fruitless exploration of that state's extensions. Moreover, consistency of each new state can be tested just by comparing

```
mkTree :: CSP -> Tree State
mkTree CSP{vars,vals} = initTree next []
  where next ss = [ ((maxLevel ss + 1) := j):ss | maxLevel ss < vars, j <- [1..vals] ]

data Maybe a = Just a | Nothing deriving Eq

earliestInconsistency :: CSP -> State -> Maybe (Var,Var)
earliestInconsistency CSP{rel} [] = Nothing
earliestInconsistency CSP{rel} (a:as) =
  case filter (not . rel a) (reverse as) of
    [] -> Nothing
    (b:_) -> Just (level a, level b)

labelInconsistencies :: CSP -> Transform State (State,Maybe (Var,Var))
labelInconsistencies csp = mapTree f
    where f s = (s,earliestInconsistency csp s)

btsolver0 :: CSP -> [State]
btsolver0 csp =
  (filter (complete csp) . leaves . (mapTree fst) . prune ((/= Nothing) . snd)
                                          . (labelInconsistencies csp) .  mkTree) csp
```

Figure 3: Simple backtracking solver for CSPs.

the newly added assignment to all previous assignments in the state, since any inconsistency involving *only* the previous assignments would already have been discovered earlier. If the solver manages to reach a complete state without encountering an inconsistency, it records a solution; if multiple solutions are wanted, it backtracks to find the others.

Backtracking solvers can be viewed very naturally as searching a *tree*, in which each node corresponds to a state and the descendents of a node correspond to extensions of its state. In conventional imperative implementations of backtracking, the tree is not explicit in the program; if a recursive implementation is used, the tree is isomorphic to the dynamic activation history tree of the program, but usually the tree is little more than a metaphor for helping the programmer reason informally about the algorithm. In the lazy functional paradigm it is natural to treat search trees as *explicit* data structures, i.e., programs are constructed as pipelines of operations that build, search, label, manipulate, and prune actual trees. As before, we rely on laziness to avoid actually building the entire tree.

Figure 2 gives Haskell definitions for a tree datatype and associated utility functions. A `Tree` is a node containing a label and a list of children, themselves `Trees`. `mapTree`, `foldTree`, and `filterTree` are the analogues of the familiar functions on lists. `leaves` extracts the labels of the leaves of a tree into a list in left-to-right order. `initTree` generates a tree from a function that computes the children of a node [9].

The code in Figure 3 uses these trees to implement a backtracking solver `btsolver0` using a lazy pipeline. All the algorithms discussed in this paper expect the tree to be generated and maintained in fixed variable order, so that nodes at level i of the tree (counting the root as level 0) always extend their parent by an assignment to v_i. Thus, the generator, `mkTree`, works by providing a `next` function to `initTree` that generates one extension for each possible value of the next variable. Each node describes an entire (partial) state, but (in any reasonable Haskell implementation) it actually stores only a single assignment, together with a pointer to the remainder of the state embedded in its parent node.

The application `(labelInconsistencies` *csp*`)` returns a *tree transformer*: it adds an annotation to each node recording its earliest inconsistent pair (if any), as returned by `earliestInconsistency`. The standard tree function `prune` *p* removes nodes for which predicate *p* is true; in this instance it prunes all inconsistent nodes. The annotations are then removed by `(mapTree fst)`. Any nodes representing

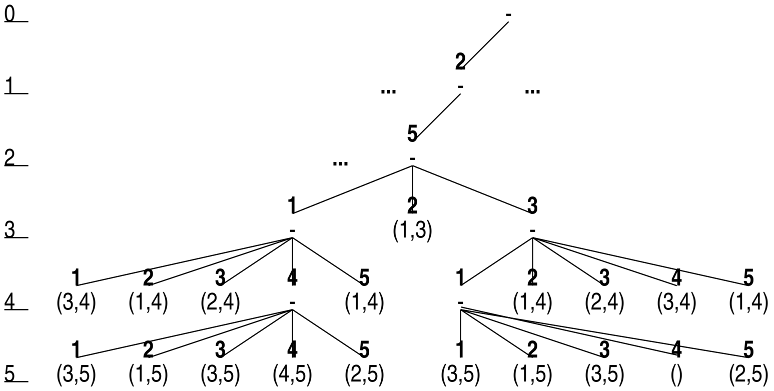

Figure 4: Portion of search tree for `queens 5`. Nodes at level i are annotated with their assigned value x_i (in **bold**), and with their earliest inconsistent pair, if any.

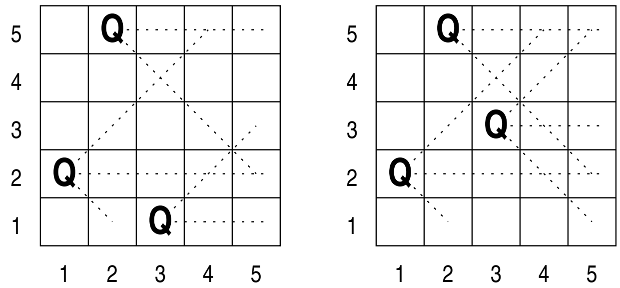

Figure 5: Two positions from the `queens 5` search tree in Figure 4. The left and right diagrams correspond to the left-most and right-most subtrees of level 3, respectively.

complete states that are still left in the tree must be solutions; the remaining pipeline stages extract these using the standard tree function `leaves` and the standard list function `filter`. Figure 4 illustrates the labels produced by `btsolver0` on part of the tree for `queens 5`; the corresponding board positions are shown in Figure 5. Note that the children of inconsistent nodes have been pruned.

It is essential to note that this pipeline is demand driven: each stage executes only when demanded by the following stage. In particular, inconsistency calculations will *not* be performed on nodes of the tree excised by `prune`, because the values of these nodes will never be demanded. Thus we get the desired effect of backtracking without any explicit manipulation of control flow. Also, as before, only a small part of each intermediate tree is ever "live" (non-garbage data) at any one time, namely the spine of the tree from root to current node, i.e., essentially what would be stored in activation records for a recursive imperative implementation. So our lazy algorithms pay at worst a constant factor more space than their imperative counterparts. We do, however, pay some overhead for building, storing, and garbage collecting each tree node, and, unless our Haskell implementation performs effective deforestation [7], this cost will be repeated for each intermediate tree in the pipeline. For these reasons, the lazy implementation of backtracking is about four times slower than a monolithic, strict Haskell implementation (see Section 11).

```
data ConflictSet = Known [Var] | Unknown deriving Eq

knownConflict :: ConflictSet -> Bool
knownConflict (Known (a:as)) = True
knownConflict _              = False

knownSolution :: ConflictSet -> Bool
knownSolution (Known []) = True
knownSolution _          = False

checkComplete :: CSP -> State -> ConflictSet
checkComplete csp s = if complete csp s then Known [] else Unknown

type Labeler = CSP -> Transform State (State, ConflictSet)

search :: Labeler -> CSP -> [State]
search labeler csp =
  (map fst . filter (knownSolution . snd) . leaves .
                             prune (knownConflict . snd) . labeler csp . mkTree) csp

bt :: Labeler
bt csp = mapTree f
  where f s = (s,
               case earliestInconsistency csp s of
                 Nothing    -> checkComplete csp s
                 Just (a,b) -> Known [a,b])

btsolver :: CSP -> [State]
btsolver = search bt
```

Figure 6: Conflict-directed solving of CSPs.

5 Conflict Sets and Generic Search

The utility of the backtracking solver is based on its ability to prune subtrees rooted at inconsistent nodes; it does nothing with consistent nodes. Of course, just because a state is consistent doesn't mean it can be extended to a solution; the assignments already made may be inconsistent with any possible choices for future variables. Figure 4 shows an example for `queens 5`: the assignment to value 1 at level 3 of the left-hand tree is consistent, but cannot be extended to a solution.

If a solver could identify such *conflicted* states, it could prune their subtrees too. To make precise the exact conditions under which such pruning is possible, we use the following definition. A *conflict set* for a state is a subset of (the indices of) the variables assigned by the state such that *any* solution must assign a *different* value to at least one member of the subset. More formally, given a state $S = \{v_1\!:=\!x_1, v_2\!:=\!x_2, \ldots, v_k\!:=\!x_k\}$, a conflict set CS for S is a subset of $\{1, 2, \ldots, k\}$ such that, if $\{v_1\!:=\!y_1, v_2\!:=\!y_2, \ldots, v_n\!:=\!y_n\}$ is a solution, then $(\exists i \in CS) x_i \neq y_i$. (Thinking imperatively, we might say a conflict set contains variables at least one of which "must be changed" to reach a solution.) Note that conflict sets are not, in general, uniquely defined. In particular, if a state at level k has a non-empty conflict set CS, then every subset of $\{1, \ldots, k\}$ containing CS is also a conflict set. If a state has a non-empty conflict set then no extension of that state can be a solution; conversely, if it has an empty conflict set, then it must have at least one extension that is a solution. This is a very strong characterization of states: for example, if we could compute a conflict set for the root of the tree (the empty state), we could test whether it were empty and thereby determine whether the problem has a solution at all! We will therefore often operate in an environment where many conflict sets are unknown. It is obviously not possible to identify a conflicted, but consistent, state without exploring *some* of its extensions;

```
hrandom :: Int -> Transform a a
hrandom seed (Node a cs) = Node a (randomList seed' (zipWith hrandom (randoms seed') cs))
  where seed' = random seed

btr :: Int -> Labeler
btr seed csp = bt csp . hrandom seed
```

Figure 7: A randomization heuristic

the trick is to avoid exploring all of them, and save effort by pruning the remainder. We address algorithms with this property beginning in Section 7.

For the moment, note that any inconsistent state has a non-empty conflict set. In particular, if a state has an earliest inconsistent pair (i, j) then it has $\{i, j\}$ as a conflict set, which we call the *earliest conflict set*. So we can subsume backtracking search in a more general algorithm we call conflict-directed search, shown in Figure 6. We define a generic routine `search`, parameterized by a `labeler` function, which annotates nodes with conflict sets. More precisely, if the labeler can determine a legal conflict set s for the node, it annotates the node with `Known` s; otherwise, it annotates it with `Unknown`. (In general, we also permit the labeler to rearrange or prune its input tree, so long as its output tree is properly labeled and still contains all solution states.) The output of the labeling stage is fed to a pruner, which removes subtrees rooted at nodes labeled with known non-empty conflict sets. Again, demand-driven execution guarantees that the excised subtrees never need to be labeled. Because of this arrangement, the labeler is allowed to assume that if it labels a node with a non-empty conflict set, it will never be called on a descendent of that node, so it need not annotate such descendents properly; this allows simpler labeler code. After pruning, the solution nodes are just the leaves of the tree annotated with known empty conflict sets; the remainder of the pipeline simply filters these out.

The framework of Figure 6 is sufficiently general-purpose to accommodate all the search algorithms discussed in the remainder of the paper. By instantiating `search` with the labeler function `bt` we obtain a simple backtracking solver `btsolver` that behaves just like `btsolver0`. The more sophisticated algorithms discussed below are all obtained by using fancier labeler functions, leaving `search` itself unchanged.

6 Heuristics and Search Order

As with the naive solver, if we are interested in only the first solution rather than all solutions, we can still use `search` unchanged; we merely demand just the head of the solution list. Since solutions are always extracted in left-to-right order, this implies that the time required to find the first solution will be very sensitive to the order in which values are tried for each variable. The use of *value-ordering* heuristics is well-established in the imperative search literature. Such heuristics can be implemented using specialized generator functions that produce the initial tree in the desired order. A more modular approach, however, is to view these heuristics as as *rearrangements* of a canonically-ordered initial tree; this keeps the initial generator simple and allows multiple heuristics to be readily composed.

Such rearrangement heuristics can be easily expressed in our framework by incorporating them into the `labeler` function. For example, `queens` search can be speeded up by considering values in random order. The following function `hrandom` in Figure 7 transforms a canonical tree by randomizing its children (using a random number generator not shown here). The application (`btr` *seed*) returns a labeler that combines randomization with standard backtracking search. We have implemented a number of other such heuristics, both generic and problem-specific, but we omit details from this paper for lack of space.

```
type Table = [Row]       -- indexed by Var
type Row = [ConflictSet] -- indexed by Value

bm :: Labeler
bm csp = mapTree fst . lookupCache csp . cacheChecks csp (emptyTable csp)

emptyTable :: CSP -> Table
emptyTable CSP{vars,vals} = []:[[Unknown | m <- [1..vals]] | n <- [1..vars]]

cacheChecks :: CSP -> Table -> Transform State (State, Table)
cacheChecks csp tbl (Node s cs) =
  Node (s, tbl) (map (cacheChecks csp (fillTable s csp (tail tbl))) cs)

fillTable :: State -> CSP -> Table -> Table
fillTable [] csp tbl = tbl
fillTable ((var' := val'):as) CSP{vars,vals,rel} tbl =
  zipWith (zipWith f) tbl [[(var,val) | val <- [1..vals]] | var <- [var'+1..vars]]
    where f cs (var,val) =
            if cs == Unknown && not (rel (var' := val') (var := val))
            then Known [var',var]
            else cs

lookupCache :: CSP -> Transform (State, Table) ((State, ConflictSet), Table)
lookupCache csp t = mapTree f t
  where f ([], tbl)      = (([], Unknown), tbl)
        f (s@(a:_), tbl) = ((s, cs), tbl)
          where cs = if tableEntry == Unknown then checkComplete csp s else tableEntry
                tableEntry = (head tbl)!!(value a-1)
```

Figure 8: Backmarking

7 Backmarking

Given the formulation of backtracking search as a pipelined algorithm with separate labeling and pruning phases, using a tree annotated with conflict sets as intermediate data structure, it makes sense to ask if there are other ways to perform the labeling phase. `bt` works by checking each assignment against all previous assignments in its state. Although this approach checks the overall consistency of each partial state only once, it can still perform many duplicate pairwise consistency checks because all the children of a given node are isomorphic. Consider a node s at level l, and consider any descendent of s. In checking the consistency of the descendent, pairwise checks will be made between its assignment and all the assignments in s at levels less than l. These checks will be duplicated for the corresponding descendents of *every* sibling of s (unless, of course, they had an inconsistent ancestor and have been pruned away). For an example, compare the leftmost nodes of the left-most and right-most subtrees on level 5 of Figure 4: to generate these conflict sets, `bt` makes the same three comparisons in each case.

An alternative approach is to *cache* the results of such consistency checks so they can be reused for each sibling; this should reduce the total number of consistency checks at the cost of the space needed for the cache. Figure 8 shows a Haskell algorithm incorporating this idea. We annotate each node with a cache to store information about inconsistencies between that node's state and the assignments made in its descendents. Each cache is organized as a table of earliest conflict sets for *all* descendents, indexed by level (greater than or equal to the node's own level) and value; the table is represented as a list of lists. The root has a table in which every entry contains `Unknown`. `fillTable` computes the table contents for a node based on the node's assignment and the node's parent's table by considering each possible future assignment in turn. If the parent's table already records a known conflict pair for the future assignment, that conflict

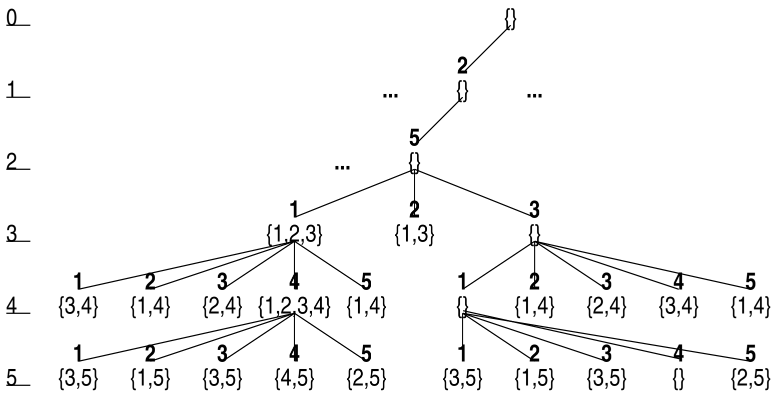

Figure 9: Same portion of search tree for `queens 5`, annotated with conflict sets as computed by `bj`.

pair is copied into the current table; otherwise a conflict check is performed and the result (a known conflict pair or `Unknown`) is recorded. Note that each node's table contains a refinement of the information in its parent's table, with a table at level l containing complete consistency information about assignments at level l. Once the tree has been annotated with cache tables, `lookupCache` is mapped over each node to extract the conflict pair for the node's own assignment from the node's table; if the node has no recorded conflicts and represents a complete state, it is a solution and is therefore given an empty conflict set. The ultimate annotated tree is identical to that produced by `bt`.

As usual, we rely on lazy evaluation to avoid building the tables or their contents unless they are needed. So most of the tables remain unbuilt, and the actual order in which consistency checks is performed is similar to `bt`. The important point is that, because many of a node's table entries are inherited from its parent's table, all duplicate consistency checks are avoided.

As before, we obtain a complete solver by using `bm` as the `labeler` parameter to `search`. Somewhat surprisingly, this algorithm turns out to be equivalent (in terms of consistency checks made) to a standard imperative algorithm called *backmarking*[1].

8 Conflict-Directed Backjumping

The `bt` and `bm` algorithms annotate inconsistent nodes with known conflict sets, but most internal nodes remain marked `Unknown`. If we could somehow compute non-empty conflict sets for internal nodes closer to the root of the tree, we could prune larger subtrees and so speed up search. In fact, many such nodes *do* have non-empty conflict sets; for example, see the leftmost node at level 3 in Figure 9.

One approach to computing internal node conflict sets is to construct them bottom-up from the conflict sets of a subset of their children. To do this, we make use of two key facts about conflict sets:

- (i) If a node s at level l has a child (at level $l+1$) with a known conflict set CS that does not contain $l+1$, then CS is also a conflict set for s. (In particular, if s has a child with an empty conflict set, then s also has an empty conflict set.)
- (ii) If all the children of node s at level l have non-empty conflict sets $CS_1, CS_2, \ldots, CS_n$, then $(CS_1 \cup CS_2 \cup \ldots \cup CS_n) \cap \{1, \ldots, l\}$ is a conflict set for s.

```
bjbt :: Labeler
bjbt csp = bj csp . bt csp

bj :: CSP -> Transform (State, ConflictSet) (State, ConflictSet)
bj csp = foldTree f
  where f (a, Known cs) chs = Node (a,Known cs) chs
        f (a, Unknown)  chs = Node (a,Known cs') chs
          where cs' = combine (map label chs) []

combine :: [(State, ConflictSet)] -> [Var] -> [Var]
combine []                 acc = acc
combine ((s, Known cs):css) acc =
  if maxLevel s `notElem` cs then cs else combine css (cs `union` acc)

bj' :: CSP -> Transform (State, ConflictSet) (State, ConflictSet)
bj' csp = foldTree f
  where f (a, Known cs) chs = Node (a,Known cs) chs
        f (a, Unknown) chs =
             if knownConflict cs' then Node (a,cs') [] else Node (a,cs') chs
                  where cs' = Known (combine (map label chs) [])
```

Figure 10: Conflict-directed backjumping.

These facts are easy to prove from the definition of conflict set. Intuitively, fact (i) says that if *any* child of s has conflicts that don't involve v_{l+1}, then *all* children of s have (at least) the same conflicts, and hence so does s itself. (The special case just says that if a child of s can be extended to a solution, then so can s.) Fact (ii) says that if no child of s can be extended to a solution, then neither can s, and any solution must differ from s in the value of at least one of the offending variables of one of the children. Fact (i) is the crucial one for optimizing search, since it permits the parent's conflict set to be computed from a strict *subset* of the children's conflict sets.

We can now define a lazy bottom-up algorithm for computing internal node conflict sets from a tree that has been (lazily) "seeded" with at least one conflict set along every path from root to leaf. Function `bj` in Figure 10 is a Haskell version of this labeling algorithm. At each parent node that doesn't already have a conflict set, `bj` calls `combine` to build one. `combine` inspects the conflict sets of the children in turn. If it finds a child to which fact (i) can be applied, it immediately returns this as the conflict set for the parent; if no such child is found, it applies fact (ii).[2] Under lazy evaluation, the subtrees corresponding to the remaining children are never explored.

This algorithm works correctly for *any* initial seeding of conflict sets, but it is most effective when the conflict sets are small and contain low-numbered variables, because this increases the number of levels for which fact (i) can be applied. This is why we use *earliest* inconsistent pairs to represent consistency conflicts. The combination of `bj` with `bt` is commonly referred to as *conflict-directed backjumping (CBJ)* (or just backjumping) in the literature and it is the cornerstone of many newly-developed algorithms [6]. In its usual imperative formulation this algorithm is notoriously difficult to understand or prove correct. While we have relied on the analysis of Caldwell, et al. [4] for our understanding of conflict sets, we are unaware of any description of the algorithm as a form of labeling.

While `search bjbt` behaves just like imperative CBJ in the sense that it performs the same number of consistency checks, it has an unfortunate space leak. The problem is that the pruning phase cannot remove the children of a node until that node's conflict set has been computed, but that computation may generate a substantial part of the children's subtrees into memory. We can plug the space leak effectively, if not too

[2] To simplify the implementation, we don't bother performing the intersection step in fact (ii), since it is harmless for a node's (non-empty) conflict set to include indices of its descendents.

```
fc :: Labeler
fc csp = domainWipeOut csp . lookupCache csp . cacheChecks csp (emptyTable csp)

collect :: [ConflictSet] -> [Var]
collect [] = []
collect (Known cs:css) = cs `union` (collect css)

domainWipeOut :: CSP -> Transform ((State, ConflictSet), Table) (State, ConflictSet)
domainWipeOut CSPvars t = mapTree f t
  where f ((as, cs), tbl) = (as, cs')
          where wipedDomains = ([vs | vs <- tbl, all (knownConflict) vs])
                cs' = if null wipedDomains then cs else Known(collect(head wipedDomains))
```

Figure 11: Forward checking.

neatly, by adding additional pruning into the labeler itself, as illustrated by `bj'`.

9 Forward Checking

Another way of assigning conflict sets to consistent internal nodes can be developed on the basis of the the cache tables introduced for backmarking (Section 7). Recall that these tables record, for each node, the earliest conflict sets for all descendent nodes; table entries for consistent nodes will remain marked `Unknown`. Suppose, however, that the table for some node n at level i contains a row, corresponding to a domain level $j > i$, in which every entry contains a non-empty conflict set. Then it is evident that the node can never be extended to a solution, because the assignments in n rules out all possible values for variable j. (As an example, consider the the left diagram in Figure 5; if we add a queen at position (4,4), then we can immediately see that no row placement will work for column 5.) Therefore, there must exist a non-empty conflict set for n. By labelling n with such a set, we can avoid further search in the subtree rooted at n. This technique has been called *domain wipeout* [1]. The combination of domain wipeout with backmarking corresponds to the well-known imperative algorithm called *forward checking*. Because our cache table construction is lazy, we have actually rediscovered ("for free") *minimal (*or *lazy) forward checking*, itself a recent discovery in the imperative literature [5].

Figure 11 shows code for implementing domain wipeout. To gather a list of `wipedDomains` and test whether it is non-empty is straightforward. The interesting question is what conflict set to assign to the node n if domain wipeout has occurred. Since it is always valid to throw additional variables into a non-empty conflict set, we could just use the set $\{1, \ldots, i\}$. But it is better to use the smallest available conflict sets based on the available information, because this can increase their utility for other algorithms (e.g., CBJ). In this case, the cache table row for a wiped-out domain records which existing assignment rules out each possible value for that domain. The union of the variables in these assignments (restricted to $\{1, \ldots, i\}$)[3] is a valid conflict set for n, since any solution must assign a different value to at least one of them. If there is more than one wiped out domain, we could compute a conflict set from any one of them; for simplicity and to limit computation, `domainWipeOut` just chooses the first.

10 Mixing and Matching

A major advantage of our declarative approach is that we can trivially combine algorithms using function composition, so long as they take a consistent view of conflict set annotations. The combination of forward

[3] Again, we simplify the implementation by omitting the restriction step, which is harmless.

Queens	8	9	10	11	12	13
CSPlib BT	0.01	0.05	0.27	1.50	8.91	57.34
ghc monolithic BT	0.14	0.60	3.20	18.03	108.34	686.92
ghc `btsolver0`	0.56	2.84	14.18	76.29	440.72	2686.13

Table 1: Runtime in seconds for different versions of simple backtracking search for the n-queens problem.

checking and backjumping

```
bjfc csp = bj csp .  fc csp
```

is well known, although to our knowledge it has not previously been achieved for lazy forward checking. Imperative forward checking is traditionally described as filtering out all the conflicting values from the domains of future variables; this makes it hard to explain how it can be profitably combined with backjumping, since the latter would seem to have no information on which to base backjumping decisions. Our viewpoint is that forward checking is just a more (time-)efficient way of generating conflict sets, which makes the combination perfectly reasonable.

Similarly, the combination of backmarking and backjumping

```
bjbm csp = bj csp .  bm csp
```

is tricky to implement correctly in an imperative setting [11], but is simple for us, and turns out to do do fewer consistency checks on `queens` than any of our other algorithms.

Once problem-specific value ordering heuristics are introduced, many more possibilities for new algorithm design open up. Since the best combination of algorithm features tends to depend on the particular problem at hand, it is important to be able to experiment with different combinations; our framework makes this extremely easy.

11 Experimental Results

To estimate the cost of modularity and laziness we wrote an integrated, strict version of simple backtracking search for the n-queens problem in Haskell and compared the runtime with that of `btsolver0`. Table 1 reports the results; they indicate an overhead factor of about four times. The measurements were taken using `ghc` (the Glasgow Haskell compiler) version 3.02 with optimization turned on, running on a lightly loaded Sun Ultra 1 under Solaris 2.5.1. We also show the runtime of an optimized C library for solving CSPs [17] compiled with `egcs` version 2.93.06 using `-O4` on the same platform; it runs an order of magnitude faster, partly because it performs consistency checks via lookup into a precomputed table. Table 2 gives the number of consistency checks made by the different algorithms for the n-queens problem.

12 Related Work

Hughes [9] gives a lazy development of minimax tree search. Bird and Wadler [3] treat the n-queens problem using generate-and-test and lazy lists. Laziness (not in the context of lazy languages) has been used for improving the efficiency of existing CSP algorithms [15, 5], but as far as we know laziness has not been previously been used to modularize any of the CSP algorithms presented here.

Queens	5	6	7	8	9	10	11	12	13
`bjbm`	276	909	3158	11928	49369	210210	975198	4938324	26709008
`bjfc`	279	916	3182	12229	51314	218907	1026826	5231284	28387767
`bm`	276	944	3236	12308	50866	220052	1026576	5224512	28405086
`fc`	279	920	3189	12276	51642	220745	1038129	5297651	28817439
`bjbt`	405	1828	8230	41128	214510	1099796	6129447	36890689	233851850
`bt`	405	2016	9297	46752	243009	1297558	7416541	45396914	292182579
Solutions	10	4	40	92	352	724	2680	14200	73712

Table 2: Number of consistency checks performed by various algorithms on the n-queens problem. Algorithms are identified by their labeler function name.

Many reformulations of standard algorithms into a framework exist in the literature [8, 6, 16, 2], but the frameworks typically aren't modular; in the best case the differences between two algorithms are highlighted by showing which lines of pseudo-code have changed [11]. Algorithms have been classified according to the amount of arc consistency (AC) they do [12] or the number of nodes visited [11]. These classifications have shown that the backmarking and forward checking algorithms, which were previously thought of as being fundamentally different, actually share the same foundation [1], as we independently rediscovered (Section 9). There often remains confusion, even among experts in the field, about which algorithm a given description really implements.

Considering how long the standard algorithms have existed and how much they are used, there have been surprisingly few proofs of correctness. A correctness criterion for search algorithms based on soundness and completeness was presented in Kondrak [11] and an automatic theorem prover was used to derive the algorithms in Caldwell, et al. [4].

The term "conflict set" is very common in the literature, but a precise definition is difficult to achieve; we base ours on that of Caldwell, et al. [4].

13 Conclusion

Expressing algorithms in a lazy functional language often clarifies what an algorithm does and what invariants it depends on. With a little bit of care we can modularize code that traditionally has been expressed in monolithic imperative form. Experimentation is also very easy. New combinations of algorithms, such as forward checking plus conflict-directed backjumping, can be expressed in a single line of code; the equivalent algorithm in the imperative literature requires many lines of (mysterious) C or pseudocode. Despite the overheads introduced by laziness and use of Haskell, large experiments can be conducted. For example, combining `hrandom` with `bjbt` allowed us to find solutions for the queens problem with well over 100 queens, even using the Haskell interpreter Hugs.

The major problem of working with lazy code is difficulty in predicting runtime behavior, particularly for space. Very minor code changes can often lead to asymptotic differences in space requirements, and the available tools for investigating such problems in Haskell are inadequate.

For future work, we plan to work on formal proofs of algorithmic correctness, which should be relatively easy in our framework, and to investigate variable-reordering heuristics, which are at the core of current work in the AI search literature.

References

[1] F. Bacchus and A. Grove. On the forward checking algorithm. In *Principles and Practice of Constraint Programming*, pages 293–309, Cassis, France, September 1995.

[2] F. Bacchus and P. van Run. Dynamic variable ordering in CSPs. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming*, pages 258–275, Cassis, France, September 1995.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[4] J. L. Caldwell, I. P. Gent, and J. Underwood. Search algorithms in type theory. In *Submitted to: Theoretical Computer Science: Special Issue on Proof Search in Type-theoretic Languages*, September 1997.

[5] M. J. Dent and R. Mercer. Minimal forward checking. In *Prec. of the Int'l Conference on Tools with Artificial Intelligence*, pages 432–438, New Orleans, Louisiana, 1994. IEEE Computer Society.

[6] D. H. Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems*. PhD thesis, University of California Irvine, 1997.

[7] A. Gill, J. Launchbury, and S. Peyton Jones. A short-cut to deforestation. In *Proc. ACM FPCA*, 1993.

[8] M. L. Ginsberg. Dynamic backtracking. *Journal of Artifical Intelligence Research*, 1:25–46, 1993.

[9] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[10] D. King and J. Launchbury. Structuring depth first search algorithms in Haskell. In *Proc. ACM Principles of Programming Languages*, 1995.

[11] G. Kondrak. A theoretical evaluation of selected backtracking algorithms. Master's thesis, University of Alberta, 1994.

[12] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[13] B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5(3):16–23, June 1990.

[14] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[15] T. Schiex, J. C. Regin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proc. of AAAI*, pages 216–221, Portland, Oregon, USA, 1996.

[16] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 1993.

[17] P. van Beek. A 'C' library of constraint satisfaction techniques., 1999. Available from `ftp://ftp.cs.ualberta.ca/pub/vanbeek/software/`.