Modular Lazy Search for Constraint Satisfaction Problems

Andrew Tolmach
Thomas Nordin

Pacific Software Research Center
Portland State University and Oregon Graduate Institute
Portland, Oregon
Constraint Satisfaction Problems

- Ubiquitous, important, computationally hard
  - Graph coloring and matching
  - Scene labeling for vision
  - Temporal reasoning
  - Resource allocation for planning, scheduling
  - etc., etc.

- Try to simplify constraints first; then must use brute force

- Handle **binary** constraints over **finite** domains

- Assume nothing known about **structure** of constraint graph
  - n-Queens looks just like graph coloring
CSP Algorithm Zoo

- No agreed-upon common framework.
- Many problems benefit from tailor-made combinations of algorithms.
"Re-use" of Imperative Code

```c
int FC_BJ(z)
int z;
{
    int h, i, j, jump, fail;
    if (z > N) {
        solution();
        return(N);
    }
    max_check[z] = 0;
    for (i = 0; i < K; i++) {
        if (domains[z][i])
            continue;
        v[z] = i;
        fail = consistent(z);
        if (fail == 0) {
            max_check[z] = z-1;
            jump = FC_BJ(z + 1);
            if (jump != z)
                return(jump);
            restore(z);
            if (fail)
                for (j = 1; j < z; j++)
                    if (checking[j][fail])
                        max_check[z] =
                        max(max_check[z],j);
            h = max_check[z];
            for (j = 1; j < z; j++)
                if (checking[j][z])
                    h = max(h, j);
            for (i = z; i >= h; i--)
                restore(i);
            return(h);
        }
    }
}
```

```c
int FC_CBJ(z)
int z;
{
    int h, i, j, jump, fail;
    if (z > N) {
        solution();
        return(N);
    }
    empty(conf_set[i]);
    for (i = 0; i < K; i++) {
        if (domains[z][i])
            continue;
        v[z] = i;
        fail = consistent(z);
        if (fail == 0) {
            jump = FC_CBJ(z + 1);
            if (jump != z)
                return(jump);
            restore(z);
            if (fail)
                for (j = 1; j < z; j++)
                    if (checking[j][fail])
                        add(j, conf_set[z]);
        }
    }
}
```

**Key:**
- = identical line
- ■ changed line

[Kondrak94]
Lazy Functional Programming View

- Modularize search into separate generate & test functions...
  ...communicating via explicit, but lazy, intermediate data structure.
- Simple program structure

\[
\begin{align*}
\text{generate} & : : \text{problem} \rightarrow [\text{candidate}] \\
\text{test} & : : \text{candidate} \rightarrow \text{Bool} \\
\text{search} & = (\text{filter test}) \ . \ \text{generate}
\end{align*}
\]
Binary CSPs in Haskell

- Set of variables \{1,\ldots,m\}
  
  \[
  \text{type Var} = \text{Int}
  \]

- Set of possible values \{1,\ldots,n\}, same for each variable
  
  \[
  \text{type Value} = \text{Int}
  \]

- Assignments associate variables to values
  
  \[
  \text{data Assignment} = \text{Var} ::= \text{Value}
  \]

- Set of pairwise constraints on assignments
  
  - Defined by a symmetric \textbf{oracle} function
    
    \[
    \text{type Rel} = \text{Assign} \rightarrow \text{Assign} \rightarrow \text{Bool}
    \]
    
    - If oracle returns true, assignments are \textbf{consistent}
    
    - Each call on this function is a \textbf{constraint check}

- Problem: \textbf{type CSP} = \text{CSP}\{\text{vars::Int},\text{vals::Int},\text{rel::Rel}\}
States and Solutions

- A **state** is a set of assignments
  
  
  type State = [Assignment]

- A state that assigns all variables is **complete**.
  
  
  complete :: CSP -> State -> Bool
  
  complete CSP{vars} as = (length as == vars)

- A state is **consistent** if every pair of assignments is.
  
  consistent :: CSP -> State -> Bool
  
  consistent CSP{rel} [] = True
  
  consistent CSP{rel} (a:as) =
  
  (all (rel a) as) && (consistent as)

- A **solution** is a complete, consistent state.
  
  solution :: CSP -> State -> Bool
  
  solution csp as = (complete csp as)
  
  && (consistent csp as)
n-Queens Problem

- Assume one queen per column.
- Variables model rows; values model columns.

```haskell
queens :: Int -> CSP
queens n = CSP{vars = n, vals = n, rel = safe}
    where safe (c1 := r1) (c2 := r2) =
          (r1 /= r2) && abs (c1-c2) /= abs(r1-r2)
```

- Obtaining all solutions

```haskell
solver :: CSP -> [State]
solver (queens 5)) ->
    [[e:=4,d:=1,c:=3,b:=5,a:=2], ...
```

- Obtaining one solution

```haskell
head (solver (queens 5))
```
Tree Search

data Tree a = T a [Tree a]

mkTree :: CSP -> Tree State
pruneTree :: (State -> Bool) -> Tree State -> Tree State
leaves :: Tree State -> [State]

solver :: CSP -> [State]
solver csp = (filter (complete csp) .
leaves .
pruneTree (not . (consistent csp)) .
mkTree) csp

generate  
prune   
collect
Tree Search Example

- Equivalent to ordinary imperative backtracking algorithm.
- Tree is isomorphic to activation history tree for recursive implementation.
Organizing the Zoo with Conflict Sets

- A **conflict set (CS)** for a state S is:
  - a non-empty subset of the variables in S, such that
  - if S’ is any **solution** state, then there is at least one variable x in CS such that S(x) ≠ S’(x).

  I.e., at least one of the variables in CS “must change its value” to reach a solution.

- A state that can be extended to a solution iff it has no CS.

- If we know a CS for a state, we can safely prune its sub-tree.

- Many interesting algorithms can be phrased as conflict-set computations, allowing them to be classified and combined.
Conflict Set Labeling Example

- Earliest Conflict

- Union Rule
Generic Solver in Haskell

• Parameterized by conflict set labeling mechanism
  
  ```haskell
  type ConflictSet = [Var]
  type Labeler = CSP -> Tree State ->
                  Tree (State, ConflictSet)
  ```

• Labeling just adds extra stage to solver’s “lazy pipeline”
  
  ```haskell
  search :: Labeler -> CSP -> [State]
  search labeler csp =
    (filter complete . map fst . leaves .
     prune (not.null snd) . labeler csp . mkTree) csp
  ```

• Example: simple backtracking uses a trivial labeler
  
  ```haskell
  bt :: Labeler
  bt csp = mapTree f
    where f s = (s, inconsistency csp s)
  btsolver = search bt
  ```
Conflict-directed Backjumping

- Complicated algorithm, usually phrased as “jumping back” to a state further up the recursion stack; hard to show correct.
- We can give a purely local, declarative description.
- Use union rule plus one other fact:
  - If a node A has a known conflict set CS that does not contain the variable assigned at A, then CS is also a conflict set for A’s parent.
- View CBJ as way to improve an existing CS labeling

\[
\begin{align*}
  \text{cbj} & : \text{CSP} \rightarrow \text{Tree (State,ConflictSet)} \\
               & \rightarrow \text{Tree (State,ConflictSet)} \\
  \text{cbjsolver} = \text{search cbjbt} \\
  \text{where cbjbt} & = \text{cbj csp . bt csp}
\end{align*}
\]
Backjumping Example

a
b
c
d
e
f

\{a,b,c\} \quad \{a,e\} \quad \{c,e\} \quad \{a,b,c,e\} \\
\{b,e\} \quad \{a,e\} \\
\{b,f\} \quad \{e,f\} \\
\{b,f\} \quad \{c,f\}
Some Other Algorithms

- **Forward checking, backmarking** and related algorithms compute CSs for all **future** assignments at each node.

  \[
  \text{storeConflicts} :: \text{CSP} \rightarrow \text{Tree} \text{State} \rightarrow \text{Tree} (\text{State}, \text{Cache ConflictSet})
  \]
  \[
  \text{bm csp} = \text{extractConflicts} \cdot \text{storeConflicts} \text{ csp}
  \]

- **Value-ordering heuristics** change the order of branches to put more promising branches on the left.

  \[
  \text{hrandom} :: \text{Seed} \rightarrow \text{Tree} \text{a} \rightarrow \text{Tree} \text{a}
  \]
  \[
  \text{btr} :: \text{Seed} \rightarrow \text{Labeler}
  \]
  \[
  \text{btr seed csp} = \text{bt csp} \cdot \text{hrandom} \text{ seed}
  \]

- **Fail first dynamic variable ordering** requires just slightly richer framework.

- Trivial to **mix and match** by composing labelers.
Runtime Comparison

Graph coloring

Relative runtime:
- All 12-queens
- First 16-queens
- Anna
- Miles250
- Miles500
- Miles1000

Time values:
- (79.1s)
- (0.03s)
- (0.47s)
- (0.37s)
- (0.71s)
- (2.15s)
Performance of Modular Lazy CSP

- Compared to imperative algorithms:
  - Same number of consistency checks
  - Roughly same space (polynomial in problem size) after plugging “space leaks”
  - Roughly 30X slower than optimized C (on kernel)
- Compared to manually fused Haskell code
  - Roughly 4X slower (on kernel)
- But fast enough to allow experimentation with different combinations of algorithms and heuristics.
  - Can then recode in imperative style if desired
  - Constant factors don’t matter much anyhow.
Fusion by Rewrite Rules

- Search pipeline generates **lots** and **lots** of tree nodes.
  
  \[ \text{search} \approx \text{leaves} \cdot \text{prune} \cdot \text{label} \cdot \text{mkTree} \]

- Can reimplement Tree ADT in terms of highly regular **producer** and **consumer** functions:
  
  \[
  \begin{align*}
  \text{data Tree } a & \equiv \text{T } a \ [\text{Tree } a] \\
  \text{foldTree} & :: (a \to [b] \to b) \to \text{Tree } a \to b \\
  \text{buildTree} & :: (\forall b.(a\to [b]\to b)\to b) \to \text{Tree } a \\
  \text{buildTree } g & = g \ T
  \end{align*}
  \]

- Simple rewrite **rule** describes fusion
  
  \[ \forall k, g. \text{foldTree } k \ (\text{buildTree } g) = g \ k \]
  
  to avoid building intermediate nodes

- Glasgow Haskell Compiler (GHC) has prototype mechanism to specify and apply rules.

- Improves speed of kernel by >3X, almost to hand-fused Haskell, **without** changing search application code at all.
Space Leaks

- Space behavior of lazy programs is not compositional.
- Tiny changes in the way a tree producer is used can easily change program’s space from linear to exponential.

- Our (ignorant) development cycle:
  - Code (hoping for the best)
  - Profile (awkward in practice, but tools can be improve)
  - Ponder for awhile (or ask a guru – not too useful)
  - Fiddle with the code and try again

- Improving this story is a major research challenge.
  - More important than shaving constant factors with better optimizing compilers.
Conclusions & Future Work

- Using modular lazy framework can clarify algorithms and their key invariants.
- New combinations of algorithms for particular problems can be easily expressed -- often with just one line of code.
- Useful experiments can be conducted, despite the overheads due to laziness.
- Future work:
  - More sophisticated algorithms
  - Tools/ideas for space behavior and selective laziness
  - Selling to constraints community (without functional programming?)