Search In CSPs, Search SW Engineering

CS410/510CS Lecture 7

Bart Massey <bart@cs.pdx.edu>
PSU CS Department
February 22, 2001

Overview

- Enforcing Consistency
- Pruning
- Variable Ordering
- Engineering Search SW
- Some Sample NPC Reductions

Enforcing Consistency

Search time for CSP is function of

- Product of size of domains
- Number and tightness of constraints
- Search ordering

We can try to improve all of these

- Statically
- Dynamically

k-Satisfiability

A CSP is k-satisfiable if every subset of k variables is satisfiable. A CSP is satisfiable if it is n-satisfiable, where $n = |\textbf{vars}(C)|$

Leads to notions of combining sub-solutions. Book dwells heavily on these: we will search problem as a whole

Idea of succeeding definitions: systematically eliminate domain values that cannot be part of solution

“Node Consistency”

A CSP is node-consistent if all its unary constraints are satisfied by available domain values

Unary constraints are strange: usually result from wanting uniform domains. E.g. “All variables are integers from $1 \ldots 5$, except v_4 cannot be 3”

Strong k-Consistency

A CSP is strongly k-consistent if it is $1 \ldots k$-consistent. A CSP is k-consistent iff for every consistent assignment to a subset of $k-1$ variables, there is a consistent value for any additional variable.

Book gives example of 3-consistent but not 2-consistent graph

2-consistent graph is arc-consistent. Strongly 2-consistent graph is node-consistent and arc-consistent

Directional Arc Consistency

Given static var order, CSP is directional-arc-consistent iff for any available domain value of any var x, there exists a compatible domain value for every var after x

Weaker than AC, sufficient for...

Backtrack-Free Search

Assume binary constraints form tree(!) If DAC is enforced on tree order, can find solution with no backtracks!

Restriction is very strong. If constraints form "k-tree" there is $O(a^k)$ algorithm for search

Any graph can be regarded as a "partial k-tree" and searched in $O(a^k)$ time. Catch? Intractable to find partial k-tree ordering

Arc-Consistency Algorithms

Strategy: enforce AC to reduce search time (remove "obvious" redundant domain values)

- Statically? Dynamically?
- Req. fast algorithm for AC
- Strong k-consistency?

Algorithms "AC-1 — AC-4"

- AC-1: $O(a^3ve)$ time, $O(e + na)$ space
- AC-4: $O(a^2e)$ time and space

Non-Binary Constraints

Most of discussion has been about binary CSPs. Common problems (e.g. SAT) not necessarily binary

- Can lift algorithms discussed thus far to *hypergraph* (hard)
- Can binarize constraints
  - not so hard
  - increases problem size
  - obscures problem

Pruning

After problem reduction, try efficient search!

Dynamic pruning: CSP-specific methods?

- Directly prune unsat subtrees
- Control search order to avoid useless work
- Learn better prunes

Lookahead

Idea: when var is bound, reduce remaining problem

E.g. check that all unbound variables still OK: *Forward Checking*

Use FC-2: one value in domain of each unbound var consistent with all bound vars

Works great for n-queens!

DAC-L

Recall Directional Arc Consistency algorithm: cheap, fairly powerful

DAC-Lookahead (Partial Lookahead): Run DAC dynamically (after every binding)

- Can integrate DAC with search: enforce directional consistency with new binding at each step
- Nice tradeoff between work/node and search

AC-L (Full Lookahead)

Can enforce full AC incrementally as well

- Maintain AC among unbound vars
- Delete inconsistent vals when new binding
- Easier with dynamic var order
- Somewhat more pruning than DAC-L

Dependency-Directed Backtracking

Idea: when binding fails

- May not be enough to undo immediate (DFS) parent
- May be obvious how far back search must go

Several methods allow backing up farther than immediate parent

Backjumping

When no val is available for var

- Some set of constraint must conflict with every possible val
- Identify latest conflicting constraint
- Backtrack at least that far
- Implementation: return "conflict set" up to first conflict
- Easy to implement
- Usually helps a lot with bad var ordering

Graph-Based Backjumping

Instead of jumping back to most recent culprit, jump back to most recent constraining var

- Easier to implement
- Usually about as good

Dynamic Backtracking

Idea: Maybe should undo *just* assignments causing constraint to fail!

- Difficult to do: must catch all of them
- Difficult to keep systematic/complete
- Dynamic Backtracking attempts to allow this
- Should work better than it does
- May occasionally lose completely (c.f. Baker *The Hazards of Fancy Backtracking*)

Backmarking

Running time may be dominated not by nodes searched, but by *constraint checks performed*

Idea: Remove redundant constraint checks; checks whose failure can be anticipated

Mechanism: keep track of vars which have changed, relationship between vars and test var. If a check failed before, and nothing has changed, it will fail again...

Learning Nogoods

Note that backjumping knows more than how far back to jump: complex reason for failure

Remember ("learn") this information as *nogood*: do not go there again

Learning Nogood Compound Labels algorithm remembers "minimal cover" of conflicting vars, uses this to prune search

Death To N-Queens!

Most examples in Tsang are around N-Queens problem

- Great for textbook examples
- Often used as performance benchmark
- For this, it is lousy
  - very regular constraint structure
  - polytime solvable without search
  - instances become progressively less constrained

Variable Ordering

Want principles for CSP var ordering

- Static
- Dynamic

Want to give good definition for most-constrained-first, or find better plan

Static Total Orderings

First look at static orderings

- Can use hefty computation to optimize
- Allow some additional algorithms
- Well understood

Max Degree Ordering

Obvious: order vars from largest to smallest degree in constraint (hyper)graph

- Does not properly account for overlapping constraints
- Does not suggest how to break ties
- Does not properly account for easy/hard constraints
- Easy to understand, implement

Graph Width

The *width* of a node v in graph with ordered nodes is number of earlier nodes adjacent to v

The *width* of a graph under an ordering is the maximum width of any node

The *width* of an unordered graph is the minimum width of any ordering

Minimum Width Ordering

Taking variables in minimum-width order reduces backtracking

- Most constrained variables are labelled first
- Least constrained variables should be OK at end
- If minimum width is k, and graph is strongly k-consistent, search in MWO is backtrack-free

Graph Bandwidth and Minimum Bandwidth Ordering

- Bandwidth is measure of closeness between adjacent nodes in ordered graph
- Searching with minimum bandwidth ordering should reduce backtracking distance on failure
- Not such a big win; use DDB to make long backtracks cheaper

Dynamic Ordering

All of these orderings are static. We know that dynamic ordering can be “better”:

- Early mistakes
- Takes character of constraints into account, not just topology

The Fail-First Principle

Idea: pick variable which has current smallest domain

- Dynamic technique
- Cheap
- Should get cutoffs early rather than late
- Matches intuitions about "most constrained first"

Recommended

Engineering Search SW

Why is search SW special?

- Algorithms, data structures tend to be intricate
- Cannot always exhaustively test
- Instrumentation is difficult
  - almost always interferes with performance
  - hard to cope with volume

...

The Software Lifecycle

- Requirements
- Design
  - Architecture

    . . .

  - Detailed
- Implementation
- Testing
- Maintenance

Prototyping and Desk Design

For search software, important to

- Build prototypes
- Do pencil-and-paper design work

Remember, prototypes should be thrown away, leaving only ideas behind!

Get heuristics right, choose right search, *before* committing to quality implementation

Final design should not be too general!

The Importance Of Pseudocode

Search SW requires pseudocode

- Too hard to keep track of big picture in details
- Programming languages rarely fit well with ideas

Try it!

The Choice Of Programming Language

PL properties that help with search SW

- Strong static and dynamic typechecking
- Dynamic memory management (GC)—but watch out for memory consumption
- No efficiency surprises
- Support for discrete structures

Implementing Recursive Algorithms

CS101: a recursive algorithm has

- Base case
- Recursive case

In procedural language, put base case test + exit first:

```
int foo(n, x) {
   if (n == 0)
     return x;
   int y = f(x);
   return foo(n - 1, y);
}
```

The Importance Of Unit Testing

Search code is

- Hard to diagnose
- Composed of small, simple procedures

This makes unit test important and easy

Hint: replace recursive calls with instrumentation before unit test; makes sure reduction is correct

How To System Test (A Bit)

First principle: system testing search code is hopeless; space is too big

Advice

- Try small examples
- Check examples carefully with pencil and paper
- Be sure the bug is in the code (not in your mind or in the test case)

The Unimportance Of Code Tuning

Remember: saving 20or breadth

It may also make your program wrong

When possible, change the algorithm or data structures, not the implementation

Search is cool, because there are usually huge speedups available without much work

From 3-SAT To CLIQUE

- Idea: encode formula as graph
  - 1 node/literal (i.e 3/clause)
  - edges connect all literals in different clauses except negated
  - k clauses satisfiable iff k-clique
    * if: true literals form clique
    * only if: literals in clique all true

From CLIQUE To VERTEX-COVER

- Find vertices s.t. all edges hit
- Take complement G' of G
- G' has k-clique iff G has $|V| - k$ -cover

From 3-SAT To HAM-CYCLE

Hamiltonian Cycle: path through graph hitting each node exactly once

Take 3-SAT to HAM-CYCLE? Build fancy graphs with structures corresponding to clauses, literals, vars, etc!

Classic scary reduction

Gadget A

Gadget B

The Construction

Use A for var/clause constraint and B for clause sat constraint

From HAM-CYCLE To TSP

Easy final redn: TSP is shortest weighted cycle. Can reduce HAM-CYCLE to TSP with weights 0, 1 for edges/non-edges of graph. TSP has 0 cycle iff graph has Ham cycle