L7: Recursive Data Types

Excerpt for Spring '13: SAT

Today

- Immutable lists
- Datatypes & functions over datatypes
- Abstract syntax trees
- SAT
Boolean Formulas and Satisfiability

the SAT problem

- given a formula made of boolean variables and operators \((P \lor Q) \land (\neg P \lor R)\)
- find an assignment to the variables that makes it true
- possible assignments, with solutions in green, are:

  - \{P = false, Q = false, R = false\}
  - \{P = false, Q = false, R = true\}
  - \{P = false, Q = true, R = false\}
  - \{P = false, Q = true, R = true\}
  - \{P = true, Q = false, R = false\}
  - \{P = true, Q = false, R = true\}
  - \{P = true, Q = true, R = false\}
  - \{P = true, Q = true, R = true\}
SAT solver

- program that takes a boolean formula in CNF
- returns an assignment, or says none exists

how to build a SAT solver, version one

- just enumerate assignments, and check formula for each
- for k variables, \(2^k\) assignments: surely can do better?

SAT is hard

- in the worst case, no: you can’t do better
- Cook (1973): 3-SAT (3 literals/clause) is “NP-complete”
- the quintessential “hard problem” ever since

how to be a pessimist

- suppose you have a problem P (that is, a class of problems)
- show SAT reducible to P (ie, can translate any SAT-problem to a P-problem)
- then if P weren’t hard, SAT wouldn’t be either; so P is hard too

remarkable discovery

![Graph showing the number of boolean variables SAT solver can handle](image)

#boolean vars SAT solver can handle (from Sharad Malik)

- most SAT problems are easy
- can solve in much less than exponential time

how to be an optimist

- suppose you have a problem P
- reduce it to SAT, and solve with SAT solver

In the 1980s, researchers were publishing papers on how to find hard SAT problems! It turned out that even though in the worst case SAT is really hard, in practice almost all the cases you get if you generate them randomly are easy. And the ones that arise in real problems are often easy too. The story’s actually more complicated than this though; it turns out that there’s what’s called a ‘phase transition’, a point at which problems get really hard, and this phase transition is roughly at the midpoint between the two extremes of the formula being so constrained it’s easy to solve because you can easily determine values for variables early on, and the formula being so underconstrained that it’s easy to solve just by guessing.
applications of SAT

planning
- solve \((\text{initial state} \land \text{goal} \land \text{rules})\) to obtain plan
- eg, ZYpp package manager for Linux
verification
- solve \((\text{code} \land \neg \text{spec})\) to obtain counterexample
- industrial application to hardware; software applications coming
design
- solve \((\text{design rules} \land \text{constraints} \land \text{requirements})\) to obtain design

for more info
- see http://www.satlive.org

why are we teaching you this?

SAT is cool
- good for (geeky) cocktail parties
- many useful applications
- compilation-to-SAT idea is powerful
- builds on your 6.042 knowledge

fundamental techniques
- you’ll learn about datatypes and functions
- same ideas will work for any compiler or interpreter

A Naive SAT Solver

one way to represent boolean formulas

\[
\text{Formula} = \text{Var(name:String)}
+ \text{Not(formula: Formula)}
+ \text{Or(left: Formula,right: Formula)}
+ \text{And(left: Formula,right: Formula)}
\]

\((P \lor Q) \land (\neg P \lor R)\) would be

\[
\text{And( Or(Var("P"), Var("Q")), Or(Not(Var("P")), Var("R")))}
\]

Socrates \(\Rightarrow\) Human \(\land\) Human \(\Rightarrow\) Mortal \(\land\) \(\neg\) (Socrates \(\Rightarrow\) Mortal) would be:
\[
\text{And} (\text{Or} (\text{Not}(\text{Var}("Socrates")), \text{Var}("Human")),
\text{And} (\text{Or} (\text{Not}(\text{Var}("Human")), \text{Var}("Mortal")),
\text{Not} (\text{Or} (\text{Not}(\text{Var}("Socrates")), \text{Var}("Mortal"))))).
\]

Note that a client of this datatype should NOT see the internal classes, Not, Or, and And. They should use abstract operations to build the formula. In this case they would be operators like and, or, and not. We have to make an exception with the Var class, and expose it, or else use a constructor method. So here's what a client might write in Java:

\[
\text{P} \lor \text{Q}
\]
\[
\text{not P} \lor \text{R}
\]

a naive SAT solver

generate and test strategy

- steps
  1. extract set of variables from formula
  2. try all assignments of true/false values to those vars
     a. we'll represent an assignment with an environment Env, which is just a list of variables and their values
  3. evaluate the formula for each environment
  4. return the first environment in which the formula evaluates to true

- functions we'll need
  
  \[
  \text{vars: Formula} \rightarrow \text{Set<Var>}
  \]
  
  \[
  \text{solve: Formula} \rightarrow \text{Env?}
  \]
  
  \[
  \text{eval: Formula, Env} \rightarrow \text{Boolean}
  \]

new datatypes we'll need

\[
\text{Set<T>} = \text{ImList<T>}
\]

\[
\text{Env} = \text{ImList<Var \times Boolean>}
\]

\[
\text{Boolean} = \text{True} + \text{False} + \text{Undefined}
\]

what's wrong with our solver?

consider formula

\[
\text{Socrates} \Rightarrow \text{Human} \land \text{Human} \Rightarrow \text{Mortal} \land \neg (\text{Socrates} \Rightarrow \text{Mortal})
\]

suppose order of trying the variables is Socrates, Human, Mortal and suppose we set Socrates to true then clearly must set Human to true and then must set Mortal to true...

...but our solver ignores all this!
A better SAT Solver

Conjunctive Normal Form (CNF)

conjunctive normal form (CNF) or “product of sums”

\((P \lor Q) \land (\neg P \lor R)\) is in conjunctive normal form.

- set of clauses, each containing a set of literals \(\{\{P, Q\}, \{\neg P, R\}\}\)
- literal is just a variable, maybe negated

Note that CNF is just a format for a boolean formula -- but one that turns out to be very helpful, making it easier to write solvers. The notion of a literal is important, since it means you can only negate variables, and not clauses.

Datatype definition:

```
Formula = ImList<Clause>       // a list of clauses ANDed together
Clause = ImList<Literal>       // a list of literals ORed together
Literal = Positive(v: Var) + Negative(v:Var) // either a variable P or its negation \(\neg P\)
Var = String
```

Note that as long as the concrete classes we were using in the old Formula (And, Or, Not) were hidden from the client, and the client was limited to using abstract operations to combine formulas (and, or, not), then we can freely make this change to the rep without changing any client code. e.g.,

\(\neg P \lor R\) ( new Var(“P”).not() ).or (new Var(“R”))

now constructs a data structure that looks like:

```
[ [ Negative(Var(“P”)), Positive(Var(“R”)) ] ]
```

That is, a list containing a single clause, which in turn contains two literals, one negative and one positive.

**basic backtracking algorithm using CNF**

- CNF is a product of sums: we need every clause true, and at least one literal in each clause
- backtracking search: pick a literal, try false then true
- if clause set is empty, success
- if clause set contains empty clause, failure

example

- want to prove Socrates\(\Rightarrow\)Mortal from Socrates\(\Rightarrow\)Human \(\land\) Human\(\Rightarrow\)Mortal
- so give solver: Socrates\(\Rightarrow\)Human \(\land\) Human\(\Rightarrow\)Mortal \(\land\) \(\neg\) (Socrates\(\Rightarrow\)Mortal)
- in CNF: \((\neg\text{Socrates} \lor \text{Human}) \land (\neg\text{Human} \lor \text{Mortal}) \land \text{Socrates} \land \neg\text{Mortal}\)
- in clausal form: \(\{\neg\text{Socrates,Human}\},\{\neg\text{Human,Mortal}\},\{\text{Socrates}\},\{\neg\text{Mortal}\}\}
- in shorthand: \(\{\text{SH}\}\{\text{HM}\}\{\text{S}\}\{\text{M}\}\) (underlines mean negation)
backtracking execution

- stop when node contains {} (failure) or is empty (success)
- in this case, all paths fail, so theorem is valid
- in worst case, number of leaves is $2^\#\text{liters}$

**DPLL: the classic SAT algorithm**

- Davis-Putnam-Logemann-Loveland, 1962

key idea: **unit propagation** on top of backtracking search

- if a clause contains one literal, set that literal to true

example

```
{SH}{HM}{S}{M}

unit S

{H}{HM}{M}

unit H

{M}{M}

unit M

{}
```

- in this case, no splitting needed
- propagate S, then H, then M
- performance is often much better, but worst case still exponential
Backtracking Search with Immutability

A final comment about what we’ve seen in this lecture. We started out with immutable lists, which are a representation that permits a lot of sharing between different list instances. Sharing of a particular kind, though: only the ends of lists can actually be shared. If two lists are identical at the beginning but then diverge from each other, they have to be stored separately. (Why?)

It turns out that backtracking search is a great application for these lists, and here’s why. A search through a space (like the space of assignments to a set of boolean variables) generally proceeds by making one choice after another, and when a choice leads to a deadend, you backtrack.

Mutable data structures are typically not a good approach for backtracking. If you use a mutable Map, say, to keep track of the current variable bindings you’re trying, then you have to undo those bindings every time you backtrack. That’s error-prone and painful compared to what you do with immutable maps – when you backtrack, you just throw the map away!

But immutable data structures with no sharing aren’t a great idea either, because the space you need to keep track of where you are (in the case of SAT, the environment) will grow quadratically if you have to make a complete copy every time you take a new step. You need to hold on to all the previous environments on your path, in case you need to back up.

Immutable lists have the nice property that each step taken on the path can share all the information from the previous steps, just by adding to the front of the list. When you have to backtrack, you stop using the current step’s state – but you still have references to the previous step’s state.

Perhaps best of all, a search that uses immutable data structures is immediately ready to be parallelized. You can delegate multiple processors to search multiple paths at once, without having to deal with the problem that they’ll step on each other in a shared mutable data structure. We’ll talk about this more when we get to concurrency.

Summary

big ideas

‣ datatype definitions: a powerful way to think about abstract data types, particularly recursive ones
‣ backtracking search: easy with immutable types
‣ SAT: an important problem, theoretically & practically