Solving SAT instances using Evolutionary Computation

Kristopher Micinski, Computer Engineering, or Computer Science

Overview

- What is SAT?
- Motivation for Evolutionary Computation techniques
- Traditional solutions
  - Hill climbing
  - Other ideas
- Our representation
  - Another possible representation
- Fitness function
- Combining GA with Hill climbing
- Implementing adaptive penalties
- Results
- Conclusions and closing notes
  - Why aren’t we winning the world cup anytime soon???

What is SAT?

- The Boolean Satisfiability Problem (SAT) is as follows:
  - Given a formula in propositional logic, find a satisfying assignment (a model) for that formula.
  - Note that we can always find a solution to these problems:
    - Decidable, just enumerate the solns?
    - How can you say “for all?”

Why do we care

- Why do we care about SAT?
  - Very simple problem to state, not very expressive logic…
  - Turns out you can reduce a whole bunch of problems to SAT:
    - K-colorability
    - And you can solve it fast!
- Also…
  - First problem proven to be NP-complete
- Why really care?
  - Package management, resolve dependencies
  - $$$: Prove a circuit implements its required function
  - We can do bounded model checking with SAT.
What are genetic algorithms?

Step 1: Encode your problem as a genome, a simplified representation (in the simplest case using 1s and 0s, but could be more complex) that you can modify.

Step 2: Generate a number of these possible genomes (usually randomly).

Step 3: Evaluate each of these individuals: take the encoded solution and assign it a score based on its relative fitness to solving the problem.

How well does this bridge solve our problem?
• Costs a lot to build?
• Strong?
• Natural looking?

Good bridge
Best bridge

Bad bridge!
What are genetic algorithms?

Step 4: Generate a new population of new individuals from taking the current solutions and combining their solutions to find a better solution!

Cross over!

Why might GAs be a good solution

- **State space:**
  - Very large. First thing implemented in the GalSat solver was a proof of concept enumerative solver…
  - We’ll let you know when we finish deciding even a near trivial formula using it…
  - And state space is growing every year, as tools from industry (model checkers, optimizing compilers, and competition generators) become more and more aggressive

- **Natural encoding:**
  - Simply variable assignments, though we try to improve upon this

- **Parallelizable:**
  - SAT solutions can be checked in parallel, and we can do the evaluation easily.

So how do we solve this problem?

**Enumerative Techniques**
- Exploration and backtracking
- Seem like they’d take a long time
- There are some very good tricks that we’ll discuss…
  - You can learn from your mistakes and prune the search space

**Stochastic Search**
- Random search?
  - Really bad… SAT answers are usually fairly hard to find
- Local search?
  - Seems like you’d get stuck in local max, but works surprisingly well
- EC techniques?
  - Let’s find out!

First I’ve got to mention…

DPLL:
The best we’ve got right now (and possibly ever will have). Guess at a variable assignment, find out what other facts are forced on you, then backtrack if you have to. (Also, learn what you did wrong!)

X1
Satisfied!

X2
Satisfied!

X3
Satisfied!

CONFLICT!

Forced, from X1 and X2 and clause 2
But now! We use the simple rule: Any clause with a single unassigned literal, assign it the value in the clause

Still working on these…
Animation got too complex…

Now we simplify clause 2 and 3!

Satisfied!

We are forced to go back and say that X3
Now we simplify clause 2 and 3!

Can't be correct. We try -X3
Now we simplify clause 2 and 3!

We find that conflicts clause 3,
so we go back to X2 and switch it to -X2

Now we use the same rule (It's called Boolean Constraint Propagation, by the way) and we can see that we could choose X3 to be True!

Now we satisfy clause 2 and 3!

Satisfied!

Hill climbing the solution (GSAT)

This assignment results in only 1 conflicting assignment… but what do we flip now?

T T T

Two observations: We can make sideways flips that keep us at the same fitness, or we can flip a random bit (I do this occasionally)

Now: Find out what variable we can flip to satisfy the most clauses

First: start with a random assignment…

One last note! You might think that satisfying the most clauses is not what we want, but this intuition is very wrong, doing it based on delta (num satisfied - num made conflicting) may take up to an order of magnitude more!

Hill climbing the solution (GSAT)

This assignment results in only 1 conflicting assignment… but what do we flip now?

T T T

Two observations: We can make sideways flips that keep us at the same fitness, or we can flip a random bit (I do this occasionally)

Now: Find out what variable we can flip to satisfy the most clauses

First: start with a random assignment…

One last note! You might think that satisfying the most clauses is not what we want, but this intuition is very wrong, doing it based on delta (num satisfied - num made conflicting) may take up to an order of magnitude more!

First attempt at a genetic algorithm.

Our first attempt at a genetic algorithm is fairly obvious.

Just use a simple binary string where 1 represents variable on, and 0 represents variable off.

The genome length varies across program runs, but is fixed throughout the execution.

Evaluation function: # of satisfied clauses

We also set a low mutation rate

One variable at most changes its value.

Improving on this initial algorithm

Implemented this, didn’t work very well:

Got close to the solution quickly but didn’t converge easily.

Solution?

Hill climb a little each time!

So in our evaluation function we first calculate the fitness, then move around a little and climb a bounded number of moves

Then throw the new individual back in the population
Another representation

- What happens if we use partial assignments?
- How do we encode this?
  - Simple, just extend representation to use an on/off bit.
  - In the evaluation function we can account for this by finding out how many are assigned
  - Intuition: let’s us climb the hill a little easier

New fitness:

\[ F(x) \times G(x) \]

- \( F \) – Satisfaction function
- \( G \) – Variable assignment penalty

How do we hill climb this?

- Hill climbing this takes a bit more effort, we simply consider a ternary sequence of possibilities:
  - Unassigned
  - False
  - True
- Then we look at what flip would be best.
- We still introduce some random noise
  - About 10% seems to work well.

But we’re climbing a lot.

- Where most of our time is spent.
- We really need to optimize this, luckily we have some tricks, but it’s still hard

Modern SAT and SMT (Satisfiability Modulo Theories) solvers try to optimize for cache locality and other architecture related issues that really boost performance.

A note about penalizing

- We are trying to really capture how two factors interact in this analysis.
- This is really multi objective EC, but we didn’t have time to explore that yet.
- We usually multiply to combine, and use linear penalty scaled by some constant
- Somewhat dangerous to push too hard against one piece of the formula.
- Have to adjust it as a linear combination, etc…

Even more improvement to develop: GalSAT

- The goal here was twofold:
  - Use a GA to get close to the solution.
  - Use a hill climber to get the rest of the way there.
- But using a hill climber really pulls us up to local maximum, and so we get stuck there.
- We can try to do this stochastically, but you’ve got to be very careful.

Instead why don’t we do something like Tabu search?

- Whenever you find yourself being pulled up to a local maximum, avoid going there again (or at least… for a while…)
- That’s expensive to do in SAT!

Checking against a large number of solutions hard.

- Use a bloom filter if you really want to, but there’s an even better way with SAT!
Adaptive Clause Penalty

- We introduce what we call Adaptive Clause Penalties.
- The goal is: if many solutions start violating the same clauses, punish a candidate more for making those clauses conflicting than other clauses.

Let's say we've got a bunch of potential candidates, if they're converging, they're violating some of the same clauses more often than other clauses!

So introduce a penalty for violating more popular clauses into our fitness function!

Implementation details

- How do we actually do this?
- Keep track of a vector of the normalized clause weights, \( \varphi \), that assigns to each clause a penalty in the range \([0, 1]\)
- When we run the evaluation function, we simply penalize more highly if the clause violates a more commonly violated clause:
  
  \[
  f(C) = \frac{1}{\text{population size}} \sum \varphi(C)
  \]

It’s a generational thing:

- After we finishing processing one generation, go process the next!
- This has some nice floating point aspects too, the points “roll off.”

Evaluation function:

Implementation: GalSat

- Constructed a fairly substantial (~4kloc) SAT package to experiment with different search techniques.
- Implemented a number of the different search techniques outlined here

- GSAT
  - De-facto hill climber for SAT
- GSAT-Partial
  - Uses partial assignments in representation developed here
- SimpleGA
  - Uses GA / hill climber combo
- AdaptivePenaltyGA
- PartialGA
Results

- We ran GalSat on a number of different sample files from SATLIB.
- Solving formulas with <100 variables seems to be easy for most of the solvers we implemented.
- Except for the partial solvers, they were able to complete the formulas, but not without a lot of work...
- It became harder as we increased the variable count to around 300, but after that it became impossible for all of our stuff.
- Not great compared to state of the art solvers but they're somewhat naïve implementations wrt. Memory and cache.

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Number of Variables</th>
<th>GalSat</th>
<th>GalSat -Partial</th>
<th>SimpleGA</th>
<th>PartialGA</th>
<th>AdaptivePenaltyGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>UF150</td>
<td>150</td>
<td>6.572</td>
<td>810.228</td>
<td>26.837</td>
<td>X</td>
<td>96.734</td>
</tr>
<tr>
<td>BMF100</td>
<td>100</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MEDIUM116</td>
<td>116</td>
<td>0.032</td>
<td>430.02</td>
<td>0.013</td>
<td>21.322</td>
<td>0.682</td>
</tr>
</tbody>
</table>
| X indicates that the solver ran out of time!

So why did partial assignments do badly?

- Unfortunately, our partial assignment strategies did not do very well.
- The key observation is that if you're going to do a partial assignment, you might as well do a full assignment.
- You shouldn't worry about making a few clauses conflicting to get something satisfied.
- We even see a trend like this in GSAT: If we try to account for clauses we never get any exploration!
- The bottom line is: We just plain move slower using a partial assignment!

Notes about the exploitive nature of incremental assignment

- Discovered that there is one inherent disadvantage in using EC techniques to solve SAT.
- You have to pay the cost of the evaluation function.
- This is pretty hefty, especially when you've got 100,000 facts to check!
- They use watched literals:
  - When we change one literal, we don't have to look at all of the clauses to see which ones were satisfied, etc… just look at the clauses that include that variable.

Q: If we flip X2 which clauses do we have to check?

[X1, ~X2]
[X0, X1, X2]
[~X2, X0]
[~X0, X1]
Conclusion

- Using EC for SAT solving works
- But doesn't nearly compare to state of the art enumerative techniques:
  - They're working smart. GAs are working hard.
- The fundamental idea of using a GA is reduce the state space, but the pruning mechanisms in modern SAT solvers prune the tree and restart without losing any knowledge.

- The bottom line:
  - There's no way to beat the evaluation function.
- However,
  - Adaptive penalty function was able to beat all the other techniques we implemented on large state spaces!
  - We might be able to take this concept and apply it elsewhere
  - Tabu search from indirect consequence
- Also, try using EC for fields such as SMT, and higher order logic?!?
  - Maybe?
- Onward to MAX-SAT!

Thanks a lot!
Have a good break!