Satisfiability Modulo Theories

CIS700 — Fall 2023

Kris Micinski

In this lecture, we'll discuss **Satisfiability Modulo Theories**

SMT solving is one of the most successful results in formal methods, from both practical and theoretical perspectives

SMT solvers enrich SAT to encompass **first-order theories**, but the restriction to *satisfiability* excludes true *first order* reasoning. Supported theories include uninterpreted functions, linear arithmetic, arrays, datatypes, etc…

SMT: $x * 2 + 3 \wedge (y < x + 2 \vee p[x] > 3) \Rightarrow p[x+1] > 2$

Not SMT: $\forall x.\ \exists y.\ \forall z.\ x > 0 \Rightarrow (x * 2 + y) * z > (x * 2 + y - 1) * z$

SMT is both a kind of extension (axioms for logical theories) and restriction (only SAT, no proper quantifiers) of FOL, so we will begin with a short discussion of FOL and its use to us as computer scientists hoping to use programs to prove theorems.

In practice, SMT solvers offer you a "library" of theories. The goal is to encode application-specific information in terms of these theories without too much encoding overhead.

Digression: First-Order Logic

We have discussed FOL briefly, but have not formally defined it in slides. First-Order Logic is an extension (syntactic and semantic) to propositional logic.

Syntactically:

- Terms include not simple propositional atoms, but also ***functions*** applied to other terms, and also ***variables***
- There are also **predicates**, which can be applied to some arity of terms (e.g., GreaterThan(x,y), Includes(S,T), …)
- Propositions are extended to include the quantifiers "for all" written ∀, and "there exists," written ∃ and bind terms

Example: Group Theory

For example, the theory of **groups** can be axiomatized via a set of first-order statements as follows:

$\forall a, b \in G, (a * b) \in G$

$\forall a, b, c \in G, (a * b) * c = a * (b * c)$

$\exists e \in G\ \forall a \in G, (e * a = a * e = a)$

$\forall a \in G\ \exists a^{-1} \in G, (a * a^{-1} = a^{-1} * a = e)$

Notice that $*$ and $^{-1}$ are functions, and $=$ is a proposition

Examples of Groups

- The integers, with + for * and - for $^{-1}$
  - E.g., -((a + -b) + b) + -b= -(a + b)
- Non-zero integers with multiplicative identity:
  - * (times) is * and 1/x is $^{-1}$. The idea is that 1/x is a ***symbolic*** representation of the reciprocal; "/" is not computational. (The group theory axioms define / (via $^{-1}$)!)
  - E.g., "(1/a * 1/b) = 1/(a * b)"
- Cyclic groups: integers mod n with addition, $^{-1}$ is - mod n
- Multiplicative groups over the real numbers

The idea is that you can take the axioms of group theory and then some statement you want to test. Add its negation and prove unsat—if you can do so, you have done a ***proof***

"***Prove this (UNSAT):***"

$\forall a, b \in G, (a * b) \in G$

$\wedge\ \forall a, b, c \in G, (a * b) * c = a * (b * c)$

$\wedge\ \exists e \in G\ \forall a \in G, (e * a = a * e = a)$

$\wedge\ \forall a \in G\ \exists a^{-1} \in G, (a * a^{-1} = a^{-1} * a = e)$

$\wedge\ \neg(\forall a, b\ (a * b) * (-b * -a) = e)$ (Statement to prove)

So: how do we prove the big conjunction is UNSAT?

So: how do we prove the big conjunction is UNSAT?

Basic idea: extend a complete **proof system** from propositional logic (resolution, natural deduction, etc…) to first-order logic.

Problem: a proof system gives us the ability to give evidence to check that a certain thing is a valid proof; still doesn't give us an **algorithm** to prove things.

Propositional logic is "easy" (2^n) because the relevant domain is always finite, and thus everything can be (in principle) decided

So: how do we prove the big conjunction is UNSAT?

Most basic idea: ask a first-order theorem prover (e.g., Vampire).

In the case of Vampire (and similar tools), you will get back a proof as something akin to a big resolution-based certificate which can be checked

Practical (UN)SAT solvers (used in verification, bounded model checking, etc…) have **not** been based on natural deduction, tableaux, Hilbert systems, etc…

Instead, they have been based on DPLL/CDCL, i.e., the "decide then propagate then backtrack/learn" algorithms to materialize an assignment

Unfortunately, extending these methods to FOL is hard, because:

- First order logic is undecidable (Godel's incompleteness theorem)
- Quantifiers are hard, pose practical difficulty
- Equality tends to blow up, massively
- Decision also blows up…

Space-efficient representation of equality is key

First-order theorem provers use an advanced mix of techniques to balance these issues. Undecidable does not pose serious issues in practice for many theories, but = is generally ubiquitous—handling equality is a serious issue.

Equality may be handled by adding axioms:

$\forall x \forall y.\ x=y \Rightarrow y=x$

$\forall x_0 \forall x_1 \ldots \forall y_0 \forall y_1 \ldots\ x_0 = y_0 \wedge \ldots \Rightarrow f(x_0,x_1,\ldots) = f(y_0,y_1\ldots)$

Unfortunately, this approach can materialize a ton of data if an explicit term / formula representation is used. A key goal of SMT/first-order solvers is in efficiently handling equality

Semantics of First-Order Logic

To rigorously discuss properties of first-order logic, we should also mention a few words about its ***semantics***.

In propositional logic, interpretations are sets of “true” atoms

In predicate (first-order) logic, there is quite a bit more to define. A “first order structure” is a pair $\langle D, I \rangle$ of a “domain of discourse” and an interpretation I which maps non-logical (variables, functions, constants, predicates) to an interpretation as functions over sets

The field of ***model theory*** studies first-order structures and their properties. I will not much model theory here but will describe symbolic proof systems (without their proofs).

From first-order logic to satisfiability modulo theories

A few slides ago, we presented a general idea for proving things in first-order theories: axiomatize them via a set of axioms and then use a reasoning system / prover / etc… to prove UNSAT

Unfortunately, this approach is often inefficient in practice. Theory-specific solvers can exploit application-specific knowledge, representations (data structures), heuristics, etc…

Modern solvers eschew first-order axiomatization and instead use an application specific solver, backed by a SAT solver.

Such solvers are based on **satisfiability modulo theories** (SMT)

These solvers ***don't*** handle arbitrary quantifiers. Instead, free variables are interpreted existentially (with respect to the theory)

Why “modulo theories?”

I.e., satisfiability **with respect to** the theory used in the formula

$f(x*2) = g(x) \wedge e = g(x) \Rightarrow f(x+x) = e$

The above formula is an “SMT formula,” but this is insufficiently specific, we must say the theor(ies) being used to ensure we are rigorous about the interpretation for *, +, and function application.

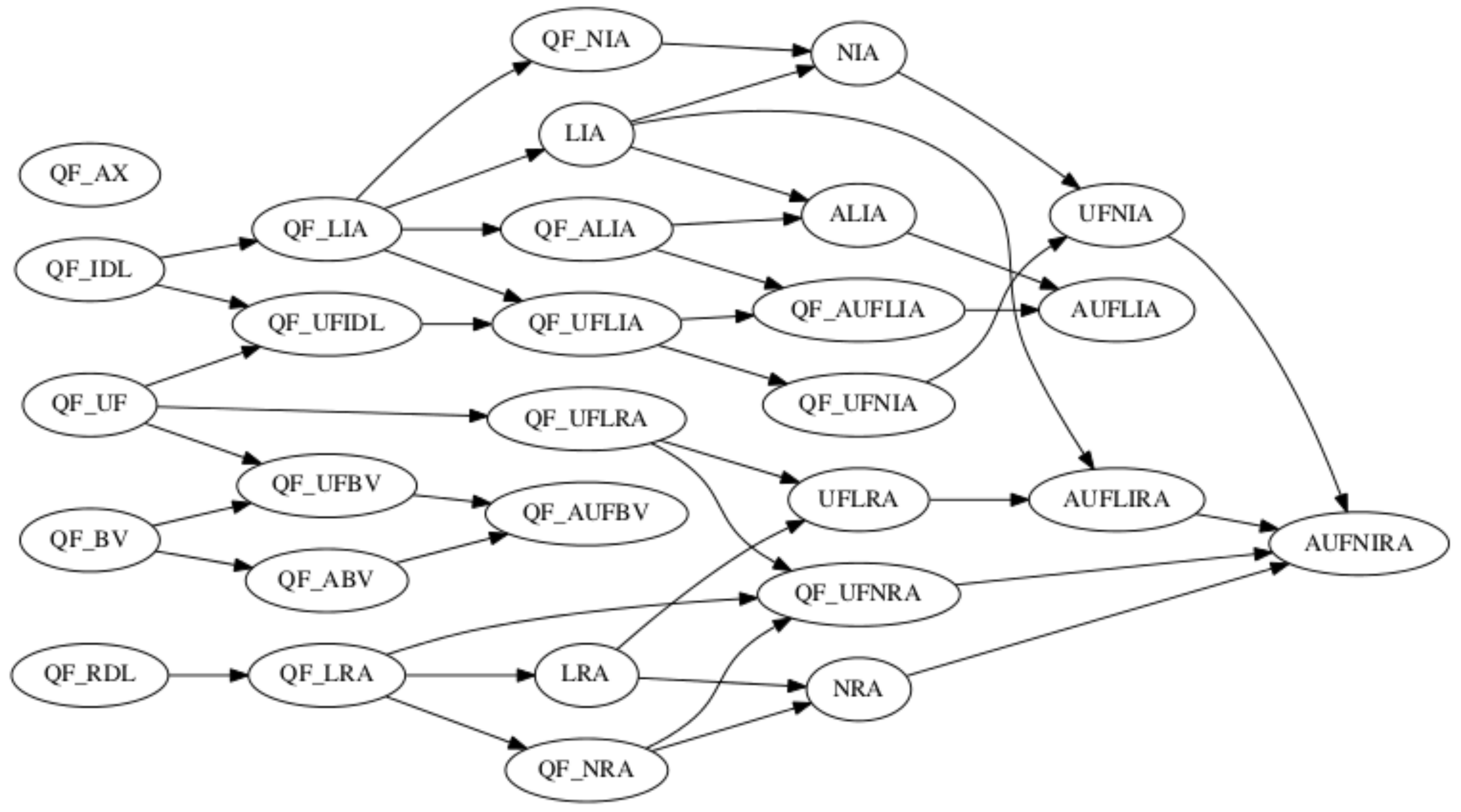
E.g., we may construe this as both:
- Linear integer arithmetic (LIA), for * and +
- Uninterpreted functions (UF), for reasoning over g and f while reasoning over them purely symbolically (“uninterpreted”)

Also, we assume all theories demand at least equality

A wide array of first-order theories, and their combinations

We can see even now that it is practically useful to mix multiple theories. There is a fairly systematic approach to combining SAT and a variety of theory-specific solvers via APIs. This is the DPLL(T) approach, and we will cover it later in the class.

The SMTLIB standard defines a variety of theories, along with a dependency graph of how they may be used.



Which theories do I choose!?

The key idea in using an SMT solver is to be able to leverage efficient solvers for a combination of theories to model properties of interest relevant to your domain.

The precise choice of theory, and whether to use an SMT solver at all, will be heavily application-dependent. It is important to ensure that there is minimal “encoding overhead.”

Example: the symbolic executor KLEE uses bit vectors (BV) along with array logic (A) and uninterpreted functions (UF)

In subsequent classes, we will discuss theory-specific solvers for uninterpreted functions (via congruence closure), bit vectors (bit blasting), and several other theories.

Worked Example: Bounded Model Checking

To demonstrate the power of SMT, we will see how we can use SMT to model the correctness of a small algorithm:

```
// input x, a bit vector of length N
i = 0;
r = 0;
while(i < N) {
  if (x & 1<<i) {
    r += 1<<i;
  }
  i++;
}
assert( r == x );
```

Worked Example: Bounded Model Checking

We will employ **bounded model checking** to check the correctness of the program. Basically, we will let inputs be symbolic, and we will unroll loops up to some depth:

```
// input x, a bit vector of length N
i_0 = 0;
r = 0;
if (i_0 < N) {
  if (x & 1<<i_0) {
    r += 1<<i_0;
  }
  i++;
}
i_1 = i_0 + 1;
if (i_1 < N) {
  if (x & 1<<i_1) {
    r += 1<<i_1;
  }
  i++;
}
i_2 = i_1 + 1
…
assert( r == x );
```

Worked Example: Bounded Model Checking

The idea is to unroll loops and convert to static single assignment (SSA) which involves “phi” nodes ϕ which are roughly “if then else”

```
(assert (= N_0 (_ bv4 4)))
(assert (= i_0 (_ bv0 4)))
(assert (= r_0 (_ bv0 4)))
(assert (= r_1 (bvadd r_0 (bvshl #x1 i_0))))
(assert
 (=
  r_2
  (ite
   (and (bvult i_0 N_0) (not (= (_ bv0 4) (bvand x (bvshl #x1 i_0)))))
   r_1
   r_0)))
(assert (= i_1 (bvadd i_0 (_ bv1 4))))
(assert (= i_2 (ite (bvult i_0 N_0) i_1 i_0)))
…
```

Full example…

https://gist.github.com/kmicinski/f583dd4ed99d71dd04184f5ce806348b

More Info

On BMC, in this case discussing a SAT-based encoding…

https://courses.cs.washington.edu/courses/cse507/21au/doc/L05.pdf

Z3's Python API

Z3's Python-based API enables us to use it without manually writing code that generates SMTLIB, it is much friendlier to use and enables high-level programming constructs like list comprehensions.

For example, I will show an implementation of sudoku taken from Leonardo de Moura's Python Z3 tutorial.

In sudoku, every number on the board has to be distinct.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]

# each cell contains a value in {1, ..., 9}
cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
             for i in range(9) for j in range(9) ]

# each row contains a digit at most once
rows_c   = [ Distinct(X[i]) for i in range(9) ]

# each column contains a digit at most once
cols_c   = [ Distinct([ X[i][j] for i in range(9) ])
             for j in range(9) ]

# each 3x3 square contains a digit at most once
sq_c     = [ Distinct([ X[3*i0 + i][3*j0 + j]
                        for i in range(3) for j in range(3) ])
             for i0 in range(3) for j0 in range(3) ]

sudoku_c = cells_c + rows_c + cols_c + sq_c
```

```
# sudoku instance, we use '0' for empty cells
instance = ((0,0,0,0,9,4,0,3,0),
            (0,0,0,5,1,0,0,0,7),
            (0,8,9,0,0,0,0,4,0),
            (0,0,0,0,0,0,2,0,8),
            (0,6,0,2,0,1,0,5,0),
            (1,0,2,0,0,0,0,0,0),
            (0,7,0,0,0,0,5,2,0),
            (9,0,0,0,6,5,0,0,0),
            (0,4,0,9,7,0,0,0,0))

instance_c = [ If(instance[i][j] == 0,
                  True,
                  X[i][j] == instance[i][j])
               for i in range(9) for j in range(9) ]

s = Solver()
s.add(sudoku_c + instance_c)
if s.check() == sat:
    m = s.model()
    r = [ [ m.evaluate(X[i][j]) for j in range(9) ]
          for i in range(9) ]
    print_matrix(r)
else:
    print ("failed to solve")
```

N-Queens (also from PyZ3 tutorial)

Place N queens on an N*N chessboard such that no two queens can attack each other. No two queens can share a row, column, or diagonal

N-Queens code (from PyZ3 tutorial)

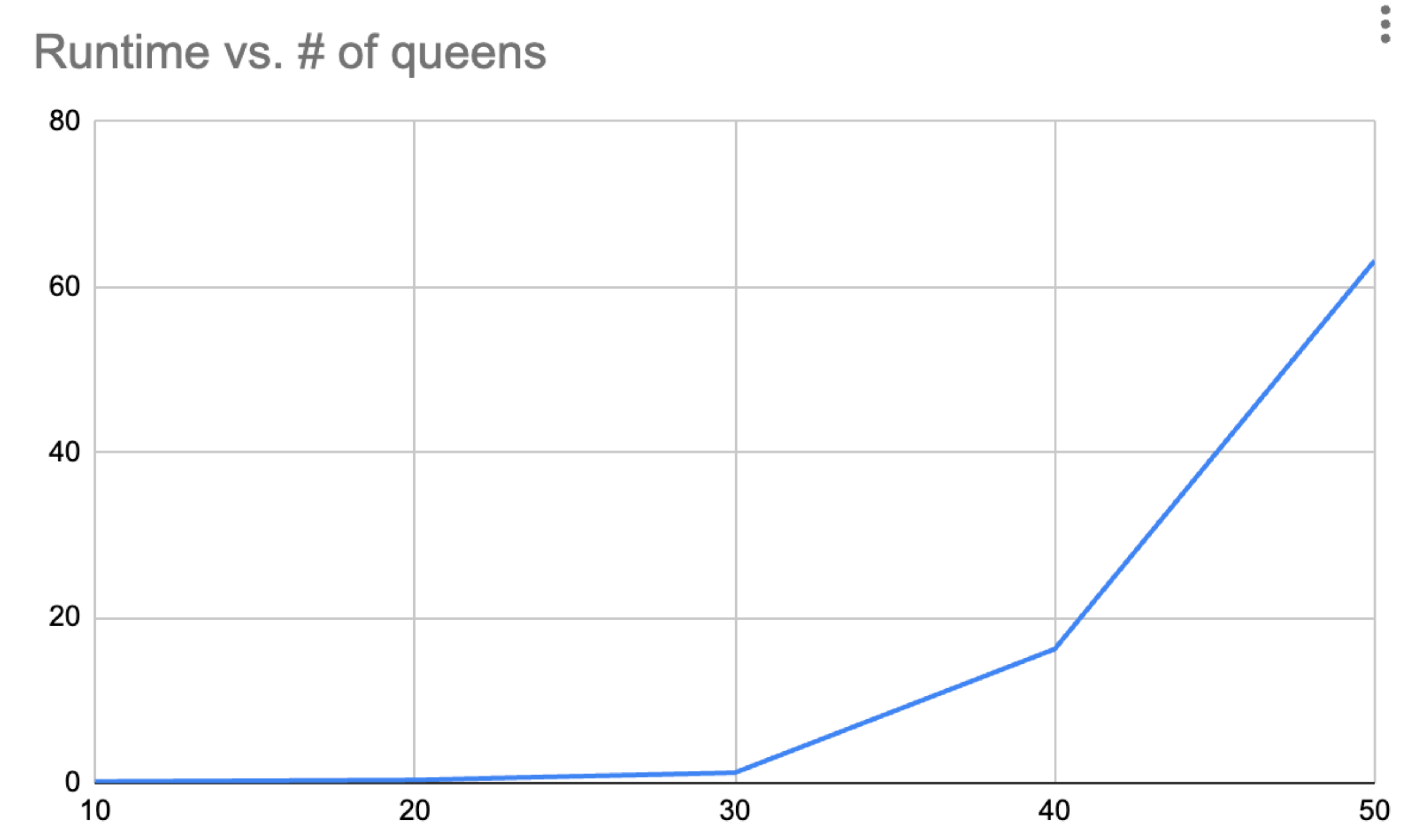
```
# We know each queen must be in a different row.
# So, we represent each queen by a single integer: the column position
Q = [ Int('Q_%i' % (i + 1)) for i in range(8) ]

# Each queen is in a column {1, ... 8 }
val_c = [ And(1 <= Q[i], Q[i] <= 8) for i in range(8) ]

# At most one queen per column
col_c = [ Distinct(Q) ]

# Diagonal constraint
diag_c = [ If(i == j,
              True,
              And(Q[i] - Q[j] != i - j, Q[i] - Q[j] != j - i))
           for i in range(8) for j in range(i) ]

solve(val_c + col_c + diag_c)
```



Using the Python-based API provides an intuitive programmatic interface and provides many high-level niceties and abstracts around some of the ugliness of manually calling Z3 and parsing its output, generating SMTLIB, etc…

In project three of the course, you'll implement a symbolic executor. In that project, I suggest you use Python and Z3's API