CIS 189

Lecture 4: Variable Selection & Engineering a Solver

Jediah Katz jediahk@seas.upenn.edu

Recap: DPLL

```
dpll(φ):
    if φ = ∅: return TRUE
    if ϵ ∈ φ: return FALSE
    if φ contains unit clause {ℓ}:
        return dpll(φ|ℓ)
    let x = pick_variable(φ)
    return dpll(φ|x) OR dpll(φ|x̄)
```

Recap: Iterative DPLL

```
dpll(φ):
    if unit_propagate() = CONFLICT: return UNSAT
    while not all variables have been set:
        let x = pick_variable()
        create new decision level
        set x = T
        while unit_propagate() = CONFLICT:
            if decision_level = 0: return UNSAT
            backtrack()
            set x = F
  return SAT
```

Recap: Iterative DPLL

```
dpll(φ):
    if unit_propagate() = CONFLICT: return UNSAT
    while not all variables have been set:
        let x = pick_variable()
        create new decision level
        set x = T
        while unit_propagate() = CONFLICT:
            if decision_level = 0: return UNSAT
            backtrack()
            set x = F
  return SAT
```

How to implement this?

Decision Heuristics

- Order of assigning variables greatly affects runtime
- Want to find a satisfying assignment quicker and find conflicts (rule out bad assignments) quicker

- **Ex:** $\{1\overline{2}34, \overline{12}3, 12\overline{3}5, 23\overline{5}, 3\overline{45}, \ldots, 67, \overline{6}7, 6\overline{7}, \overline{67}\}$
  - If we assign 6 first, then we can find conflicts right away

Naive Decision Heuristics

- We can consider a decision heuristic as a permutation of $[1..n]$
- **Ascending:** sort variables in increasing order
- **Random:** shuffle variables at random
  - Slightly better if ascending order is adversarial, but still has no knowledge of the structure of the formula

Decision Heuristic Wishlist

What properties do we want in a decision heuristic?

- Fast to compute
- Splitting creates “easy” subproblems
    - Fewer unsatisfied clauses (“smaller” formula)?
    - Shorter unsatisfied clauses (easier to do UP)?
    - Fewer variables among all unsatisfied clauses?

Attempt 1: Most Frequent

- **Intuition:** selecting more common variables will satisfy or shrink more clauses
- Sort variables in decreasing order of frequency in the entire formula
- **Issues:** common variables might appear in long clauses, isolated clauses

Attempt 2: Highest Activity

- **Intuition:** want to prioritize variables that appear in short clauses (so we can do UP)
- Each variable receives an **activity score**: each clause of length L contributes $\frac{1}{2^L}$ to its variables
- Sort variables by decreasing activity score
- **Issues:** might still prioritize variables in short but isolated clauses

Example: Decision Heuristics

$$\varphi = \{\bar{1}23, 1\bar{2}3, 12\bar{3}, \overline{5432}1, \overline{543}2, \overline{54}3, \bar{5}4, 5\bar{4}\}$$

Variable	Frequency	Activity Score
1	4	0.40625
2	5	0.46875
3	6	0.59375
4	5	0.71875
5	5	0.71875

Dynamic Decision Heuristics

- All previous heuristics are **static heuristics**
  - Fixed permutation only based on initial formula
- **Idea:** design **dynamic heuristics** that can modify the variable ordering over time

Most Frequent Revisited

- **Idea:** instead of using the initial frequency ordering, recompute frequencies after each decision
    - That is, pick the variable which appears most frequently across all unsatisfied clauses
- This decision heuristic is called **DLCS**
    - **Dynamic Largest Combined Sum**
- **Issues:** very expensive, poor combo with 2WL
- Can we compute this more efficiently?

Lazy DLCS with 2WL

- **Idea:** Maintain and update frequency ordering instead of recomputing
- Store frequencies in a priority queue, and whenever a clause “becomes satisfied” decrement the frequency of its variables
- More specifically, consider a clause to become satisfied when:
  - It is watching a literal that gets set to True, or
  - It starts watching a literal that was already True

Lazy DLCS with 2WL

- What about backtracking?
  - A clause can only become unsatisfied when its watched literal(s) change from True to unassigned after backtracking
- Idea 1: decrement frequencies after backtracking
  - Issues: complex (high bookkeeping) and possibly expensive
  - Many variables might become unassigned
- To reduce bookkeeping: keep “frequency stack” (like assignment stack)
  - On new decision, push copy of priority queue onto stack
  - When backtracking, pop off old copy of priority queue

Lazy DLCS with 2WL Example

Steps

$$(\overline{1} \vee 2)$$

$$(\overline{3} \vee 4)$$

$$(\overline{5} \vee \overline{6})$$

$$(6 \vee \overline{5} \vee \overline{2})$$

	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		**2**		**3**		**4**		**5**		**6**	

Assignment/Frequency Stack

Note: I described the frequencies as being stored in a priority queue, but for simplicity they are stored in an array here.

Lazy DLCS with 2WL Example

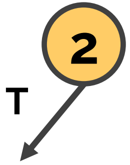
$(\overline{1} \vee 2)$

$(\overline{3} \vee 4)$

$(\overline{5} \vee \overline{6})$

$(6 \vee \overline{5} \vee \overline{2})$

Steps



	0	T	1		1		1		2		2
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		2		3		4		5		6	

Assignment/Frequency Stack

Lazy DLCS with 2WL Example

$(\overline{1} \vee 2)$

$(\overline{3} \vee 4)$

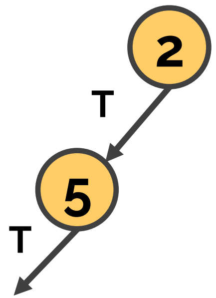
$(\overline{5} \vee \overline{6})$

$(6 \vee \overline{5} \vee \overline{2})$

	0	T	1		1		1	**T**	2		2
	0	T	1		1		1		2		2
	1		2		1		1		2		2
v	***f***	***v***	***f***	***v***	***f***	***v***	***f***	***v***	***f***	***v***	***f***
1		**2**		**3**		**4**		**5**		**6**	

Assignment/Frequency Stack

Steps



Lazy DLCS with 2WL Example

$(\overline{1} \vee 2)$

$(\overline{3} \vee 4)$

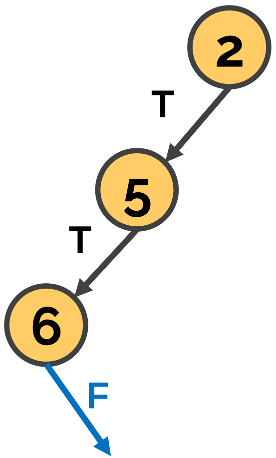
$(\overline{5} \vee \overline{6})$ **Unit!**

$(6 \vee \overline{5} \vee \overline{2})$

	0	T	1		1		1	T	**1**	**F**	**1**
	0	T	1		1		1		2		2
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		**2**		**3**		**4**		**5**		**6**	

Assignment/Frequency Stack

Steps



Lazy DLCS with 2WL Example

$(\overline{1} \vee 2)$

$(\overline{3} \vee 4)$

$(\overline{5} \vee \overline{6})$

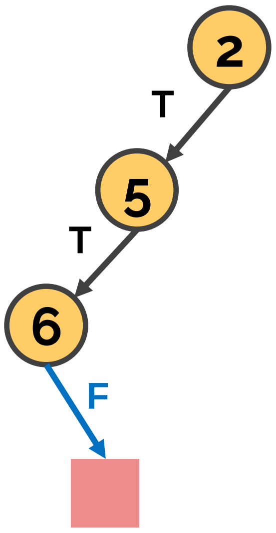
$(6 \vee \overline{5} \vee \overline{2})$

Unit! Conflict!

	0	T	1		1		1	T	1	F	1
	0	T	1		1		1		2		2
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		2		3		4		5		6	

Assignment/Frequency Stack

Steps



Lazy DLCS with 2WL Example

$(\overline{1} \lor 2)$

$(\overline{3} \lor 4)$

$(\overline{5} \lor \overline{6})$

$(6 \lor \overline{5} \lor \overline{2})$

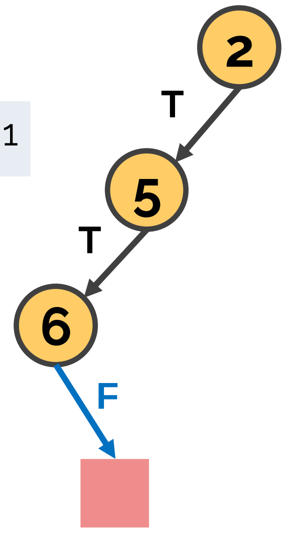
Pop!

	0	T	1		1		1	T	1	F	1

	0	T	1		1		1		2		2
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		2		3		4		5		6	

Assignment/Frequency Stack

Steps



Lazy DLCS with 2WL Example

$$(\overline{1} \vee 2)$$

$$(\overline{3} \vee 4)$$

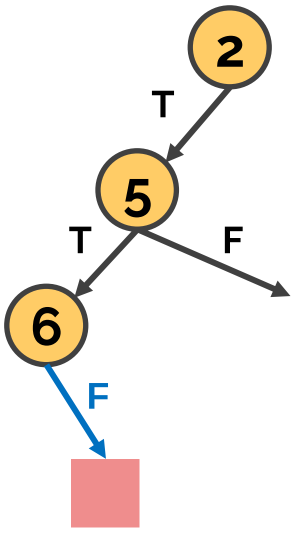
$$(\overline{5} \vee \overline{6})$$

$$(6 \vee \overline{5} \vee \overline{2})$$

	0	T	**0**		1		1	**F**	**0**		**0**
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		**2**		**3**		**4**		**5**		**6**	

Assignment/Frequency Stack

Steps



Lazy DLCS with 2WL Example

$$(\overline{1} \vee 2)$$

$$(\overline{3} \vee 4)$$

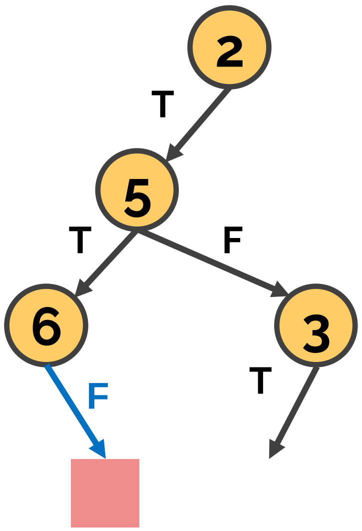
$$(\overline{5} \vee \overline{6})$$

$$(6 \vee \overline{5} \vee \overline{2})$$

	0	T	0	**T**	1		1	F	0		0
	0	T	0		1		1	F	0		0
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		**2**		**3**		**4**		**5**		**6**	

Assignment/Frequency Stack

Steps



Lazy DLCS with 2WL Example

$(\overline{1} \vee 2)$

$(\overline{3} \vee 4)$ **Unit!**

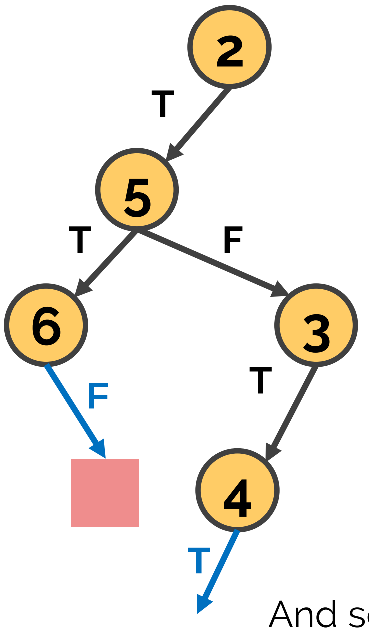
$(\overline{5} \vee \overline{6})$

$(6 \vee \overline{5} \vee \overline{2})$

	0	T	0	T	**0**	**T**	**0**	F	0		0
	0	T	0		1		1	F	0		0
	1		2		1		1		2		2
v	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*	*v*	*f*
1		2		3		4		5		6	

Assignment/Frequency Stack

Steps



And so on…

Lookahead Heuristics

- General goal of decision heuristics: pick variable causing cascade of unit propagation
- Why not calculate directly?
- **Lookahead heuristics:** for each unassigned variable x, count how many variables are set by unit propagation from $x = T$ and $x = F$
  - Pick variable that maximizes product of the numbers
- **Issue:** extremely expensive
  - In reality, "lookahead" refers to an entire family of similar techniques

Phase Selection Heuristics

- We have been always trying $x = T$ before $x = F$ for our decisions, but this is arbitrary
  - Might sometimes be faster to try one than the other
- **Phase selection heuristics:** strategies for deciding which polarity (phase) to try first
- Won't go into much depth, but many decision heuristics can be made **one-sided**
  - One-sided: pick a variable and polarity, not just variable

One-Sided Heuristic Variants

- **Dynamic Largest Individual Sum (DLIS):** one-sided variant of DLCS
  - Instead of picking the variable with the highest frequency, pick the literal with the highest frequency
  - Frequencies for positive/negative literals counted separately
- Pros & cons of one-sided variants:
  - May try the “better” polarity first for decisions
  - May fail to capture relationship between positive/negative literals of the same variable

Introducing: PennSAT

- **HW2: PennSAT (due on Mon 10/19 by midnight)**
  - **Milestone** due on Mon 10/12 by midnight
- Features:
  - DPLL-based
  - Iterative
  - 2-watched literals
  - Static, two-sided activity-based decision heuristic
- This assignment is **quite difficult – start early!**
  - Requires solid understanding & careful bookkeeping
  - May take you **10+ hours**

Testing a SAT Solver

- SAT solvers have tons of complicated logic... how to check for soundness bugs?
  - Hard and tedious to figure out all cases to unit test
- **Random testing:** generate random CNF formulas to test against reference solver
- If reference solver is not available, can at least check that satisfying assignments are valid

The $R(n, m)$ Random CNFs

- $R(n, m)$: random CNFs on n variables and m clauses
- Construct each clause as follows:
  - For each variable $x \in [1..n]$:
  - Include x, include $\overline{x}$, or include neither with probability $^1/_3$
- Remove empty clauses

The $R(n, m)$ Random CNFs

- **Tip:** to find bugs, experiment with smallish numbers and different (in)equalities of n and m
  - **Ex:** $R(3{,}4), R(4{,}3), R(4{,}4), R(3{,}5)$
  - As m grows larger than n, more likely to be UNSAT
- Good for debugging, but bad for timing
  - Too easy! Not used in practice

Debugging a SAT Solver

- Once we've found a bug, how do we find the mistake in the code?
- **Print debugging:** stick a bunch of print statements in relevant places and look at the console
- Easy, but not very effective for complex systems
    - Easy to forget to print something, or print in wrong place

Debugging in VS Code

- **Debugger:** allows us to stop program mid-execution, run code line-by-line, inspect values of local variables

```python
def __init__(self, n: int, cnf: CNF, activity_heu
    # The number of variables
    self.n = n
    # The CNF as a list of clauses
    self.cnf = preprocess(cnf)
    # A stack of partial truth assignments: list
    self.assignment_stack = [[None] * (n+1)]
```

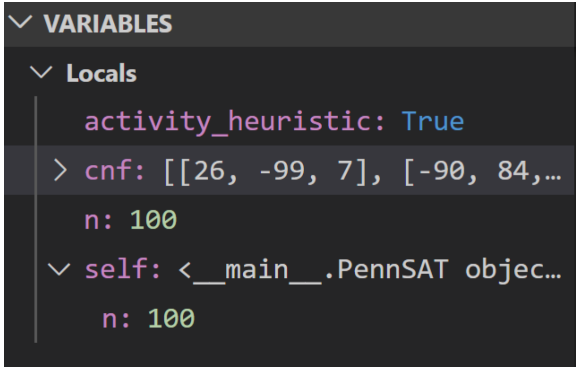
- **Breakpoint:** STOP at this line of code

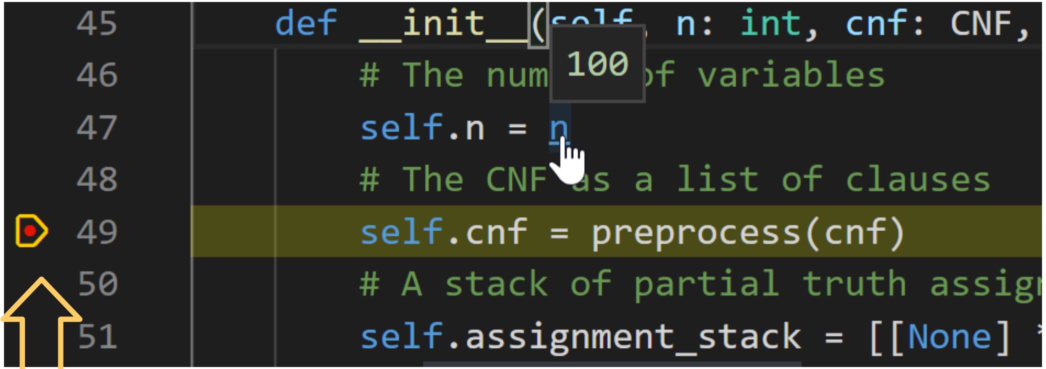
Debugging in VS Code

- After breakpoints set: Debug > Start Debugging (F5)

View or *modify* current variables & values

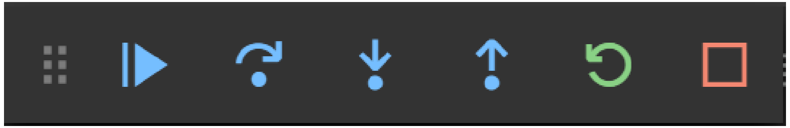
Hover over variables to inspect values





Stopped right before line 49!

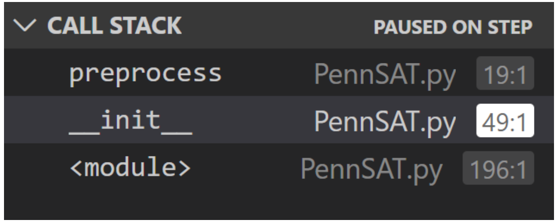
Debugging in VS Code

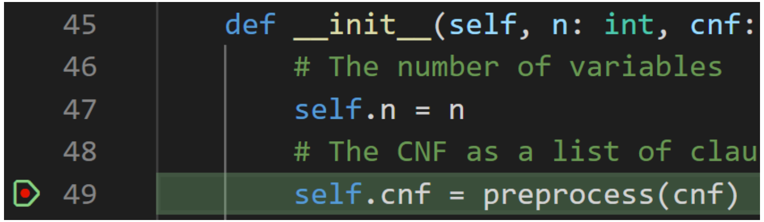
- Control flow: 

- **Continue (F5):** run until next breakpoint hit
- **Step Over (F10):** run just one more line of code
- **Restart (Ctrl+Shift+F5):** start over from beginning
- **Stop (Shift+F5):** quit the debugger

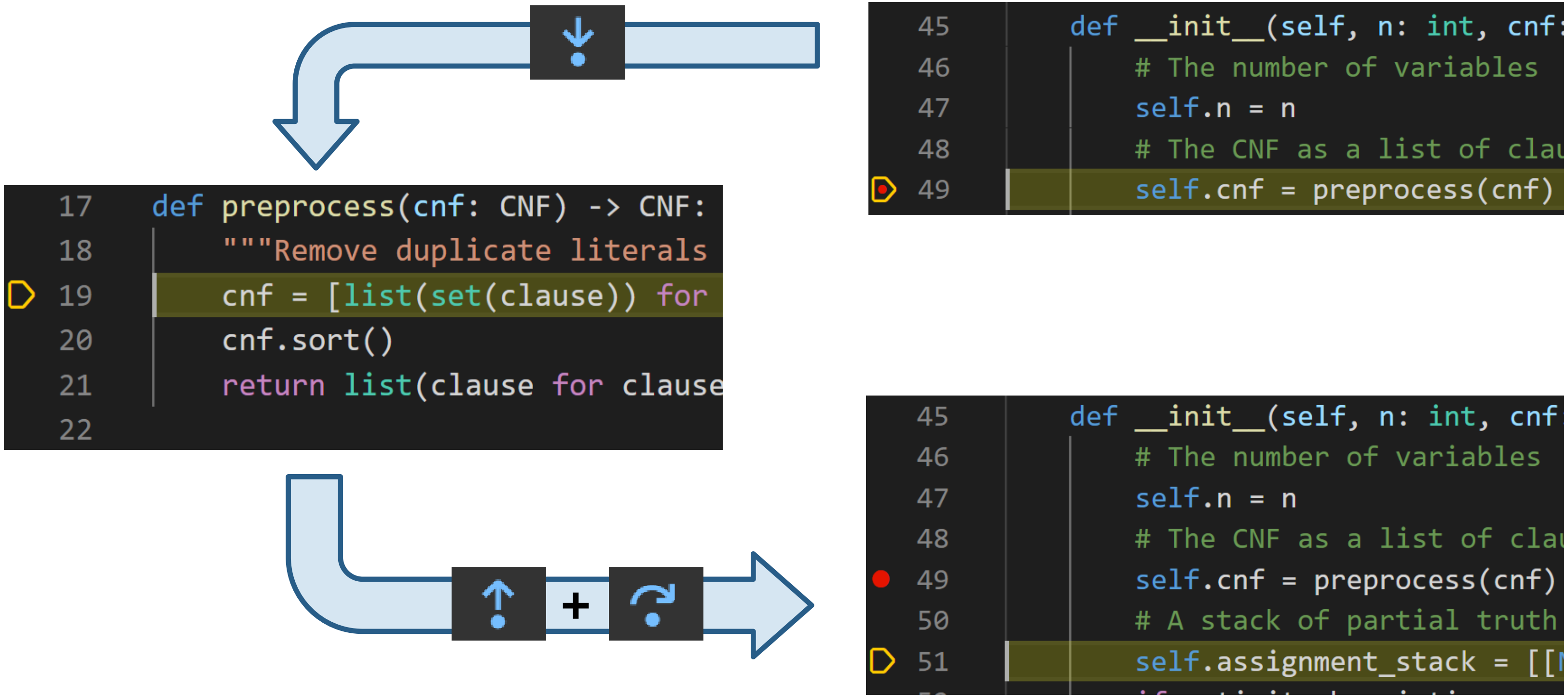
Debugging in VS Code

- **Step Into (F11):** enter code of first function called on the current line and resume debugging there
- **Step Out (Shift+F11):** run until the current function returns; resume debugging from parent function
- Can click to view different levels of the call stack
    - Useful for inspecting values of local vars in different scopes





Debugging in VS Code



Timing a SAT Solver

- **Random timing:** test runtime on many random CNFs
  - Easy to generate, but high variance and not reflective of practical problems
- **Industrial benchmarks:** difficult CNFs taken from real-world problems
  - DIMACS, SAT Competition
- **Profiling:** analyze execution of solver to discover which components are a bottleneck

The $F_k(n, m)$ Random CNFs

- $F_k(n, m)$: random k-CNFs on n variables and m clauses
- Sample m clauses uniformly from all $\binom{n}{k} 2^k$ clauses with k distinct variables
  - Choose k positive literals, and then negate each w.p. ½
- Used in practice: much harder than $R(n, k)$, especially when $m \approx 4.3n$

Profiling a SAT Solver

- A **profiler** hooks into your program and measures:
    - Number of calls for each function
    - Average / total time taken by each function
    - Which functions call which
    - Etc...
- Helps find inefficiencies, bottlenecks in code
- **cProfile:** built-in Python profiler

Basic cProfile Usage

```
python -m cProfile -s cumtime PennSAT.py
```

run in command line: sort by total time script to profile

```
      866561 function calls (866548 primitive calls) in 1.202 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   3/1    0.000    0.000    1.202    1.202 {built-in method builtins.exec}
     1    0.000    0.000    1.202    1.202 PennSAT.py:1(<module>)
     1    0.030    0.030    1.190    1.190 PennSAT.py:169(solve)
 16773    0.312    0.000    1.120    0.000 PennSAT.py:118(propagate)
157792    0.319    0.000    0.522    0.000 PennSAT.py:70(value)
 24218    0.101    0.000    0.303    0.000 PennSAT.py:78(assume)
181168    0.118    0.000    0.118    0.000 PennSAT.py:30(bsign)
182458    0.116    0.000    0.116    0.000 {built-in method builtins.abs}
```

Visualizing the Profiler

- **KCacheGrind:** tool to visualize output of profiler
  - Can download free online or install with apt-get on Linux
  - Built for C, but we can use it with Python's cProfile with the **pyprof2calltree** conversion library (install with pip/pipenv)

```
# Run in command line:
# Profile PennSAT.py and save output to PennSAT.cprof
python -m cProfile -o PennSAT.cprof PennSAT.py
# Visualize the cProfile output with KCacheGrind
pyprof2calltree -k -i PennSAT.cprof
# GUI window should open...
```

Visualizing the Profiler

Incl.	Self	Called	Function	Location
9.59	9.59	182 458	<built-in method builtins.a...	~
9.89	9.89	181 168	bsign	PennSAT.py
43.42	26.53	157 792	value	PennSAT.py
6.85	6.85	128 200	<built-in method builtins.le...	~
4.59	4.59	86 900	<method 'append' of 'colle...	~
4.25	4.25	79 437	<method 'popleft' of 'collec...	~
25.18	8.41	24 218	assume	PennSAT.py
93.09	25.92	16 773	propagate	PennSAT.py
0.10	0.10	1 782	<method 'append' of 'list' ...	~
0.10	0.10	1 684	<method 'pop' of 'list' obje...	~
0.11	0.11	860	pick_variable	PennSAT.py
1.32	0.27	842	backtrack	PennSAT.py
0.06	0.06	842	<method 'clear' of 'collecti...	~
0.03	0.03	431	<genexpr>	PennSAT.py
0.02	0.02	378	<method 'startswith' of 'str...	~
0.15	0.06	376	__setattr__	typing.py
0.09	0.05	376	_is_dunder	typing.py
0.02	0.02	351	<built-in method builtins.is...	~
0.01	0.01	222	<method 'endswith' of 'str'...	~
0.01	0.01	116	<genexpr>:613	typing.py
0.01	0.01	100	<lambda>	PennSAT.py
0.01	0.01	100	lit	PennSAT.py
0.00	0.00	68	<built-in method builtins.g...	~
0.00	0.00	68	<method 'rstrip' of 'str' obj...	~
0.03	0.02	57	_collect_type_vars	typing.py
0.23	0.05	52	__init__:603	typing.py
0.00	0.00	45	<method 'get' of 'dict' obje...	~
0.00	0.00	41	<built-in method builtins.s...	~
0.17	0.01	38	_alias	typing.py
0.00	0.00	36	<method 'join' of 'str' obje...	~
0.00	0.00	34	_verbose_message	<frozen importlib._bootst
0.01	0.01	32	<listcomp>	<frozen importlib._bootst
0.02	0.01	32	_path_join	<frozen importlib._bootst
0.00	0.00	27	abstractmethod	abc.py
0.12	0.03	26	<built-in method builtins.__...	~