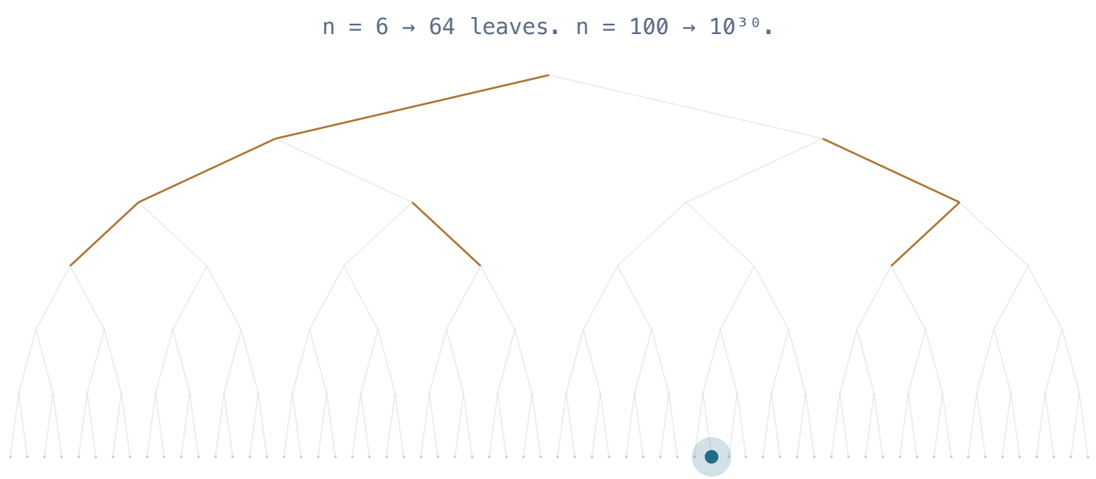
COMPILER CONSTRUCTION · WEEK 11 · STATION IV

SAT

NP-completeness, Cook and Levin through circuits, why SAT is hard, and how modern solvers answer it anyway: DPLL, CDCL, and babysat as the executable specification.

DECIDABLE ≠ DOABLE

P, NP, and the asymmetry between finding and checking.



2ⁿ branches · one satisfying leaf

P: questions some method answers in time polynomial in the input size. Parsing, sorting, running mipster for k steps. Affordable.

NP: questions whose yes-answers come with a certificate checkable in polynomial time. A satisfying assignment. A colouring of the register graph. A schedule. Finding may take 2^n; checking is a loop over the clauses.

P ⊆ NP, obviously. Whether P = NP is the open question of the exact sciences. Most researchers bet no, and this class is built on that bet.

COOK AND LEVIN · 1971–73

SAT is NP-complete: every question in NP is a formula in disguise.

Take any polynomial-time checker — a machine that reads a question and a certificate and says yes or no in p(n) steps. Its whole computation fits in p(n) states of p(n) bits. Write one variable per bit per step, and clauses saying each step follows from the last by the machine's rules — *next*, as a formula, p(n) times.

The result is satisfiable exactly if some certificate makes the checker say yes. Finding the certificate is now solving SAT. So SAT is at least as hard as everything in NP: NP-hard, and in NP itself, so NP-complete.

YOU SAW THIS LAST WEEK

Cook's construction is rotor's unrolling. A machine's computation written down as a formula, one copy of the state per step. The proof technique of 1971 is the tool of week 10, and the class has already run it.

Karp, 1972: twenty-one more problems, by reduction to and from SAT. By now thousands. Crack one, crack all.

WHY SAT IS HARD, AND WHY SOLVERS WORK

The exponential is a worst case. Real formulas have structure.

Random 3-SAT: below about 4.26 clauses per variable almost everything is satisfiable and easy; above, almost everything is unsatisfiable and easy; at the threshold, hard for every solver ever built. That phase transition is the closest thing to a picture of where the difficulty lives.

Formulas from chips, schedules and programs are not random. They have long implication chains, small backdoors, symmetries. Modern solvers handle millions of variables on them in seconds, and it is one of engineering's great quiet victories.

HARDNESS IS LOAD-BEARING

If SAT were easy, every certificate would be findable: every key, every signature, every password. Difficulty is the raw material of security and the reason abstraction and taste beat compute. The universe we live in trades that power for privacy and discovery.

BABYSAT

The executable specification: try true, try false, prune when a clause is already lost.

```
uint64_t babysat(uint64_t depth) {
if (depth == number_of_sat_variables)
return SAT;
*(sat_assignment + depth) = TRUE;
if (instance_may_be_true(depth))
if (babysat(depth + 1) == SAT) return SAT;
*(sat_assignment + depth) = FALSE;
if (instance_may_be_true(depth))
if (babysat(depth + 1) == SAT) return SAT;
return UNSAT;
}
```

Recursion over the variables in order. Set the next one true; if no clause is already false under the partial assignment, recurse. Then false, the same. If both fail, backtrack. `instance_may_be_true` is a loop over the clauses; a clause may still be true if some literal is true or unassigned.

Four hundred lines in C*, with a DIMACS parser, on top of selfie, so mipster runs it. It is what a SAT solver *is*, with the search tree of the first slide walked depth-first and pruned at dead branches. Everything below is what makes that walk short.

```
$ make sat
./babysat: examples/sat/rivest.cnf is satisfiable with -1 -2 3 4
```

DPLL · 1962

Propagate what is forced. Decide only when nothing is.

Unit propagation. A clause with all but one literal false forces the last one. Set it, and repeat, because that may make other clauses unit. A human does this without thinking; babysat does not do it at all.

Pure literals. A variable occurring only positively can be set true harmlessly; dually for negative.

Decide, backtrack. Only when neither applies, pick a variable and try a value; on conflict, go back to the last decision and try the other. Depth-first search with the subtrees that propagation kills never entered.

WHAT IT FORGETS

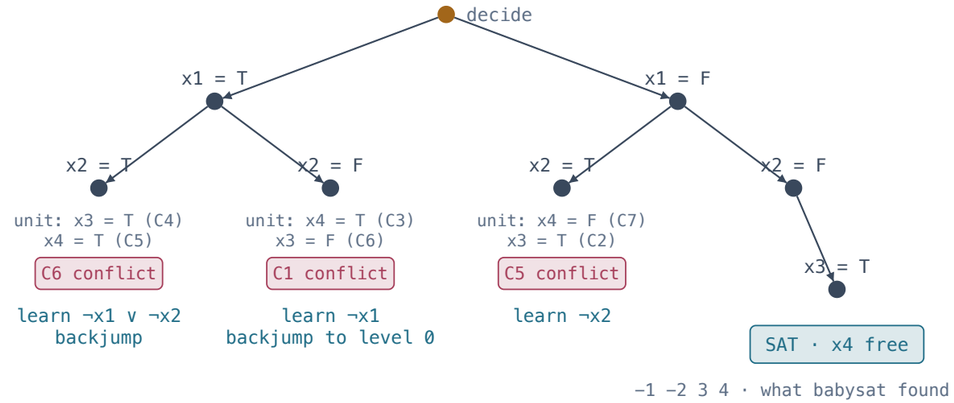
DPLL backtracks to the most recent decision, whether or not that decision caused the conflict, and may run into the same conflict again elsewhere. It does not learn. That is the gap between 1962 and 1996.

CDCL · ON RIVEST'S FORMULA

A conflict is information. Write down why, and never fail that way again.

CDCL on Rivest's seven clauses

```
C1 (2 v 3 v ¬4)
C2 (1 v 3 v 4)
C3 (¬1 v 2 v 4)
C4 (¬1 v ¬2 v 3)
C5 (¬2 v ¬3 v 4)
C6 (¬1 v ¬3 v ¬4)
C7 (1 v ¬2 v ¬4)
```



−1 −2 3 4 · what babysat found

four assignments examined, three clauses learned — instead of sixteen assignments enumerated

decide · propagate · conflict · learn · backjump

Decide x1 = T, then x2 = T; propagation forces x3 and x4 and C6 dies. Trace the conflict back: x4 came from x3 and x2, x3 from x1 and x2. The decisions responsible are x1 and x2. Learn ¬x1 ∨ ¬x2, a clause implied by the formula.

Backjump to right after x1 = T, where the learned clause is unit: x2 = F. Propagate: x4 = T, x3 = F, C1 dies. Everything traces to x1 alone. Learn ¬x1. Level 0: x1 = F.

Decide x2 = T: x4 = F, x3 = T, C5 dies. Learn ¬x2. Then x2 = F, x3 = T satisfies everything with x4 free. Four assignments, three learned clauses — and the learned clauses are theorems: x1 and x2 are false in every model, which is the comment in the file.

ASSIGNMENT

struct-declaration.

1. **Specify.** Extend the grammar with `struct` declarations: a name, a braced list of fields with types, and the type `struct name *` for pointers to them.
2. **Represent.** The symbol table needs a new class of entry: a struct with its list of fields, each with a type and an offset. Fields are laid out in declaration order, eight bytes each.
3. **Check.** Declaring a variable of a struct pointer type, and the type checking of week 4 extended to the new types. No access yet; that is next week.
4. `./grader/self.py struct-declaration` and `self-compile`.

WHY STRUCTS LAST

They are the one extension that changes the symbol table's shape rather than adding an entry, and the one that selfie itself would most benefit from: its own symbol table entries are hand-laid-out words with getter procedures, because C* has no structs. Next week you will be able to write the table you have been reading all semester.