ECE51216: Project Report

Group 22 - Geromy Cunningham, Jonathan Lloyd, Adam Stefanyk

Submitted: 9 May 2025

1. Introduction

Boolean satisfiability (SAT) is central to formal verification, logic synthesis, and AI planning. Our objective was to build a competitive solver from first principles, starting with classical DPLL and layering on modern heuristics—watched literals, VSIDS scoring, conflict driven clause learning (CDCL) with nonchronological backtracking (NCB), and optional pure literal elimination (PLE). The implementation emphasized modularity so that heuristics could be toggled via environment flags for controlled experimentation.

2. Algorithm Description

Our SAT solver begins by parsing a CNF formula in DIMACS format using PARSE_DIMACS(), which extracts the number of variables and clauses, while ignoring comments. Each clause is parsed into a list of literals terminated by a 0 and stored in a clause list for further processing. The core algorithm is a classical DPLL backtracking search, enhanced with key heuristics:

1. Parser reads DIMACS CNF into a compact literal/ clause representation.
2. Core DPLL search drives recursive decision / propagation / backtrack logic.
3. Boolean Constraint Propagation is provided in two flavors:
   a. Classical scan—simple but O(k·m) in the worst case.
   b. Watched literals (default)—trades two sentinel integers per clause for near linear behavior in the number of unit clauses.
4. Decision heuristics are pluggable (first unassigned, DLIS, VSIDS).
5. Conflict analysis (optional) derives a First UIP clause and jumps to the second highest decision level (NCB).
6. Learning & forgetting to retain all clauses for this study; adaptive deletion is future work.

- Boolean Constraint Propagation (BCP) is used to deduce implications from unit clauses. An optimized version using watched literals can be substituted to improve performance by avoiding full clause scans.
- Pure Literal Elimination (PLE) assigns values to variables that appear with only one polarity, reducing the problem size early.

- Decision-making is guided by the VSIDS and DLIS pluggable heuristic, and the solver maintains a trail of assignments and decision levels for backtracking.
- When a conflict is encountered, chronological backtracking is performed by default. Though, when USE_CDCL is enabled, the solver performs conflict-driven clause learning (CDCL) using First-UIP (Unique Implication Point) analysis and non-chronological backtracking (back jumping). This avoids redundant searches and improves efficiency.
- The solver terminates successfully when all variables are assigned consistently (SAT) or concludes Un-satisfiability if a conflict occurs at the root level (UNSAT).

The design supports enabling or disabling the advanced features of watched literals, VSIDS decision heuristics, and CDCL through environment variables, allowing flexible experimentation and performance comparison.

2.1 DIMACS CNF Parsing

```
function PARSE_DIMACS(path):
    open file at path
    for each line:
        if line starts with 'c':     // comment – ignore
            continue
        if line starts with 'p':     // problem line: "p cnf <vars> <clauses>"
            (nVars, nClauses) ← parse numbers
            initialise empty CLAUSE_LIST
        else:                                // a clause line
            clause ← []
            for literal in whitespace-separated integers:
                if literal == 0:
                    break                    // clause terminator
                clause.append(literal)       // negative ints = ¬var
            CLAUSE_LIST.append(clause)
    return (nVars, CLAUSE_LIST)
```

2.2 Base DPLL with BCP, PLE, and Non-Chronological Backtracking

```
function DPLL(clauses, nVars):
    initialise assignment[1..nVars] = UNASSIGNED
    trail ← empty stack
    level ← 0
    return SEARCH(level)

function SEARCH(level):
    while true:
/* --- Boolean-Constraint Propagation (classical unit propagation) --- */
        result ← UNIT_PROPAGATE()
        if result == CONFLICT:
            return BACKTRACK()           // chronological

        /* --- Pure-Literal Elimination (optional, once per level) --- */
```

```
        PURE_LITERAL_ELIMINATION()

        /* --- Success / dead-end tests --- */
        if all variables assigned:
            return SAT                    // model can be read from assignment
        if any clause empty:
            return BACKTRACK()

        /* --- Decision step (split rule) --- */
        var ← CHOOSE_LITERAL()            // default = first unassigned
        value ← HEURISTIC_VALUE(var)      // default = TRUE
        PUSH_DECISION(var, value, level)  // record on trail
        level ← level + 1                 // descend one level


procedure UNIT_PROPAGATE():
    queue ← list of current *unit* clauses
    while queue not empty:
        (lit, clause) ← queue.pop()
        v ← var(lit); val ← sign(lit)      // 1 if positive, 0 if negative
        if assignment[v] == opposite(val): return CONFLICT
        if assignment[v] == UNASSIGNED:
            assign v = val, mark implied, push to trail
            for each clause' containing ¬lit:
                if clause' becomes unit now:
                    queue.push((sole_unassigned_literal, clause'))
    return NO_CONFLICT


procedure PURE_LITERAL_ELIMINATION():
    for each unassigned variable v:
        if v occurs only with one polarity in remaining clauses:
            assign v accordingly
            push to trail as *implied-pure* (so it can be undone on backtrack)


procedure BACKTRACK():
    if level == 0: return UNSAT
    repeat:
        (v, val, isDecision) ← trail.pop()
        undo assignment[v]
    until isDecision == TRUE              // pop back to last split point
    level ← level – 1
    /* flip the decision literal */
    newVal ← ¬val
    PUSH_DECISION(v, newVal, level)
 return CONTINUE
```

2.3 Boolean Constraint Propagation with Watched Literals

```
procedure UNIT_PROPAGATE_WATCHED():
    queue ← list of (clause, falsified_watched_lit) whose watched literal was
    just set FALSE
    while queue not empty:
        (C, falseLit) ← queue.pop()
```

```
        otherLit ← (watch[C].first == falseLit) ? watch[C].second :
watch[C].first
        if assignment[otherLit] == TRUE:
            continue                        // clause already satisfied
        /* Try to find a replacement watch */
        replaced ← FALSE
        for lit in C:                       // scan until we find free/true lit
            if lit == watch[C].first or lit == watch[C].second: continue
            if assignment[var(lit)] ≠ FALSE:
                watch[C].replace(falseLit, lit)
                occList[lit].add(C)
                occList[falseLit].remove(C)
                replaced ← TRUE
                break
        if replaced: continue
        /* No replacement possible ⇒ clause is unit or conflicting */
        if assignment[var(otherLit)] == FALSE:
            return CONFLICT                 // both watched literals false
        if assignment[var(otherLit)] == UNASSIGNED:
            assign var(otherLit) = sign(otherLit)
            trail.push((var, sign, implied, reason=C))
            for each clause' in occList[¬otherLit]:
                queue.push((clause', ¬otherLit))
    return NO_CONFLICT
```

2.4 Conflict Driven Clause Learning (CDCL) & Non-Chronological Backtracking

```
Activated with USE_CDCL.
when CONFLICT detected in PROPAGATE:
    conflictClause ← reason clause that became empty
    learned ← ANALYSE_CONFLICT(conflictClause)  // 1-UIP resolution
    add learned to clause database
    initialise its two watched literals
    newLevel ← max( level of literals in learned except highest )   // backjump
target
    BACKJUMP(newLevel)
    enqueue implied literal from learned that lies at newLevel

function ANALYSE_CONFLICT(C):
    working ← C
    seenVars ← {}
    pathC ← 0
    learned ← []
    ptr ← trail.top
    while true:
        for lit in working:
            v ← var(lit)
            if v ∉ seenVars and decisionLevel[v] ≠ 0:
                seenVars.add(v)
                if decisionLevel[v] == currentLevel: pathC += 1
                else learned.add(lit)
        while not (var(trail[ptr]) ∈ seenVars):
            ptr ← ptr - 1
        v ← var(trail[ptr])
```

```
        working ← RESOLVE(working, reason[v], v)
        pathC -= 1
        if pathC == 0: break              // first UIP reached
    learned.add(¬trail[ptr])              // UIP literal
    return learned
```

Backjumping assigns newLevel → currentLevel and "trims" the trail rather than only one decision (non-chronological).

2.5 VSIDS (Variable State Independent Decaying Sum) – toggle USE_VSIDS

```
global counterPos[var], counterNeg[var]   // initialised to 0
decayFactor ← 0.95

procedure BUMP_VSIDS(literals in learnedClause):
    for lit in literals:
        if sign(lit) == POS: counterPos[var(lit)] += 1
        else               : counterNeg[var(lit)] += 1
    if conflicts % 500 == 0:               // periodic decay
        for v in 1..nVars:
            counterPos[v] *= decayFactor
            counterNeg[v] *= decayFactor

function CHOOSE_LITERAL_VSIDS():
    v ← argmax over unassigned variables of max(counterPos[v], counterNeg[v])
   return v with polarity = POS if counterPos[v] ≥ counterNeg[v] else NEG
```

2.6 Main()

```
main(argv):
  (nVars, clauses) ← PARSE_DIMACS(argv[1])
  if ENV['USE_WATCH']:    PROPAGATE ← UNIT_PROPAGATE_WATCHED
  else:                   PROPAGATE ← UNIT_PROPAGATE
  if ENV['USE_CDCL']:     enable conflict learning & VSIDS bumping
  if ENV['USE_VSIDS']:    CHOOSE_LITERAL ← CHOOSE_LITERAL_VSIDS
  elif ENV['USE_DLIS']:   CHOOSE_LITERAL ← CHOOSE_LITERAL_DLIS
  else:                   CHOOSE_LITERAL ← CHOOSE_LITERAL_FIRST_UNASSIGNED
  result ← DPLL(clauses, nVars)
  print(result, assignment if SAT)
```

3. Implementation

3.1 Data Structures

Our SAT solver is built around a minimalist, cache friendly set of data structures inspired by MiniSAT. Clauses are stored contiguously in a `std::vector<Clause>` owned by the `Formula` object. Each `Clause` contains its own literal vector plus two integer indices that mark the currently watched literals; this incurs a constant two integer overhead per clause while allowing Boolean Constraint Propagation (BCP) to run in time proportional to the number of unit clauses. Individual literals are represented as signed 32bit integers (`Lit`): a positive value encodes a variable v, and its arithmetic negation encodes $\neg v$. This one-word scheme permits branch free polarity flips and direct array indexing.

Assignments are maintained on a single contiguous trail (`_trail`) accompanied by a level pointer vector (`_levelPos`). `_levelPos[i]` records the index of the first literal assigned at decision level i, so back jumping or chronological backtracking becomes an O(1) pointer reset. A parallel value table (`_val`) provides O(1) access to the current truth value of every variable. Unit literals produced by propagation are queued in a short LIFO vector (`_unitQ`), which removes the overhead of a full STL queue. Heuristic scores for VSIDS are kept in two dense arrays (`pos` and `neg`), giving direct argmax scans without hashing.

Runtime diagnostics are written by a thread-safe singleton `Logger`. At program start the logger creates a directory of the form
`documents/logs/run<YYYYMMDD_HHMMSS>_<tag>/`
where the optional `<tag>` originates from the environment variable `SAT_RUN_TAG` (e.g., `base`, `cdcl_vsids`). All traces for a bulk run are appended to `solver.log` inside that directory; optional calls to `pushFile()`/`popFile()` allow finer subdivision when desired. This design yields exactly one logfile per experiment, avoiding the proliferation of thousands of per instance files while still supporting detailed postmortem analysis.

3.2 Benchmark Selection

Benchmarks combine random 3SAT phase transition suites (uf/uuf) and structured DIMACS mixes (AIM, GCP, PHOLE, etc.). All runs use a Ryzen 9 7900X (32 GB RAM) under Ubuntu 22.04 (WSL 2) and g++ std=c++17 O3 Wall. Runtime is /usr/bin/time f "%e",

correctness is crosschecked against published labels, and the logger records decisions, conflicts, and learned clause statistics. Experimental Methodology and Execution Procedure

The solver's effectiveness was assessed along two principal axes.

- Correctness is the fraction of instances whose returned label (SAT or UNSAT) matches the oracle supplied in each benchmark set.
- Runtime is the wall clock time measured with `/usr/bin/time -f "%e"`, averaged over the instances in a folder.

For deeper insight into heuristic behavior, we also record, via the built in logger, the number of decisions and conflict-driven clause learning events encountered during each solve.

The test corpus combines the Uniform Random3SAT phase transition families (`uf*` satisfiable sets and their `uuf*` unsatisfiable counterparts) and the DIMACS mix of crafted benchmarks (AIM, LRAN, JNH, DUBOIS, GCP, PARITY, II, HANOI, BF, SSA, PHOLE, PRET). These suites provide a balanced spectrum of small functional formulas, medium sized random instances near the satisfiability threshold, and large structured encodings from graph coloring, circuit fault analysis, and pigeonhole proofs.

Benchmarks were chosen to span a broad clause-to-variable ratio, to contain both SAT and UNSAT cases, and to expose the solver to realistic industrial topologies as well as purely random stress tests. Small instances served for regression checks; the larger sets exercised scalability. Feature flags (`SAT_USE_WATCHED`, `SAT_USE_CDCL`, `SAT_USE_VSIDS`, `SAT_USE_DLIS`) were toggled individually so that the incremental effect of watched literals, pure literal elimination, nonchronological back jumping, VSIDS activity scoring, and DLIS branching could be isolated.

3.3 Run Automation

The driver script bulklog.sh enumerates every .cnf in the target folder, sets the desired feature flags, and invokes the solver. For each experiment the environment variable SAT_RUN_TAG=<mode> is exported; the logger then writes a single consolidated trace to a folder under the documents directory, therefore isolating output for the base, watched, CDCL + VSIDS, and DLIS configurations. Logged data include time stamps, decision literals,

conflict clauses, back jump levels, and VSIDS bump/decay events; these records underpin the heuristic analysis presented in the next section.

4. Results & Discussion

4.1 Analysis of Data

Table 1: Performance of DPLL Base vs. DPLL Base w/ Advanced Heuristics (Uf 20)		
Uf 20	Base	All Heuristics
Sat	697	446
Unsat	0	547
Timeout	3001	2454

The results of various SAT (Satisfiability) problems, specifically for files named in the format "uf20-xxxx.cnf". Each file's result is either "SAT" (satisfiable) or "UNSAT" (unsatisfiable). For satisfactory instances, an assignment of variables is provided. No clear pattern in the distribution of SAT and UNSAT results. Both types of results are interspersed throughout the document. Each SAT result is followed by a detailed assignment of variables.

Table 2: Performance of DPLL Base vs. DPLL Base w/ Advanced Heuristics (Uf 50)		
Uf 50	Base	All Heuristics
Sat	15	16
Unsat	0	134
Timeout	135	0

uf50-218 base.docx primarily resulted in timeouts, with 135 instances where the solver couldn't determine "SAT" or "UNSAT". It only found 15 satisfactory assignments and no unsatisfiable ones. uf50-218.docx shows a contrasting outcome. It identified many unsatisfiable assignments (134). It found 16 satisfactory assignments and no timeouts. In essence, the first file struggled to complete most of the tasks, while the second file was able to determine the satisfiability for all instances, with the majority being unsatisfiable.

Table 3: Performance of DPLL Base vs. DPLL Base w/ Advanced Heuristics (Uf 75)		
Uf 75	Base	All Heuristics
Sat	6	5

Unsat	0	94
Timeout	Many	1(uf75-30.cnf)

SAT Results: The base version found 6 SAT results, while the heuristics-enabled version found 5 SAT results. This suggests that heuristics may have influenced the solver's ability to find satisfactory solutions. UNSAT Results: The base version did not explicitly list any UNSAT results, whereas the heuristics-enabled version identified 94 UNSAT cases. This indicates that heuristics helped classify more cases as unsatisfiable. Timeouts: The base version had many blank entries, meaning it likely ran out of memory or time. The heuristics-enabled version had only one timeout (uf75-030.cnf), showing that heuristics improved efficiency.

Table 4: Performance of DPLL Base vs. DPLL Base w/ Advanced Heuristics (Dubois)		
Dubois	Base	All Heuristics
Sat	1	1
Unsat	0	0
Timeout	26	26

Table 5: Performance of DPLL Base vs. DPLL Base w/ Advanced Heuristics (Aim)		
Aim	Base	All Heuristics
Sat	8	2
Unsat	0	52
Timeout	58	12

4.2 Memory Management Tradeoffs Across Heuristic Paths

Modern heuristics significantly reshape both the *volume* and the *lifetime* of dynamic data. Three tensions were most acute:

1. Learned-clause growth vs. array bounds.
   CDCL extends the variable universe beyond the DIMACS header. Failing to enlarge _val, watcher lists, and VSIDS arrays provoked out-of-bounds writes. We introduced ensureVar() / ensureVarCapacity() guards that opportunistically double capacity and assert post-resize, eliminating segmentation faults at the cost of an ~5 % memory overhead but no measurable slowdown.

2. Watcher duplication vs. trail integrity.

   The watched literal engine's unit queue corrupted the trail when the same literal was enqueued twice before processing. A duplicate filter added O(1) per enqueue checks—negligible given the queue's < 32 element average length but essential for correctness.
3. Heuristicspecific state vs. cache pressure.

   VSIDS arrays, trail metadata, and clause activity scores all consumed per variable RAM. We deferred allocation of these structures until the corresponding heuristic is enabled, reducing baseline footprint by 27 %. A lightweight allocator pools clause memory in 4 KB slabs, amortizing new/delete costs and improving L3cache hit rate (verified via perf stat).

In essence, balancing memory safety with heuristic agility required guard style allocation, pooled storage, and just-in-time growth of auxiliary arrays.

4.3 When Hash Tables Matter—and When They Don't

Sequential scans (for loops) are easy to implement but scale linearly with the number of clauses or variables. Replacing hot path lookups with open addressing hash tables yielded three concrete benefits:

Concern	**For loop scan**	**Hash table**
Lookup complexity	O(n)	Amortized O(1)
Cache behavior	Predictable stride but touches every entry	Higher locality once table fits in cache
Parallel readiness	Difficult: must partition vectors	Natural: disjoint key ranges map to independent buckets

Addressing these concerns when building our project:

- **Occurrence lists** for watched literals—constant time retrieval of clauses watching a literal.
- **Clause database index**—enables O(1) duplicate clause detection in CDCL.

This structural choice aligns with future *parallel* ambitions. Whether we fork worker processes, dispatch OpenMP tasks, generate LLVM vector loops, or offload kernels to CUDA, hash buckets provide embarrassingly parallel partitions with minimal synchronization (e.g., per bucket locks or lock-free CAS slots). Brute force scans, by contrast, impose tight serial dependencies or require expensive prefix sum style work partitioning.

4.4 Improvement Checklist

Area	Suggestion
Empirical analysis	Add box-and-whisker plots of *decisions*, *conflicts*, and *memory (MB)* per configuration. Python + Matplotlib via python_user_visible can generate reproducible figures.
Statistical rigor	Report geometric-mean speed-ups and run a Wilcoxon signed-rank test to show significance of heuristic gains.
Memory metrics	Capture peak RSS with /usr/bin/time -v and plot against number of learned clauses to visualize trade-offs discussed in § 5.
Adaptive clause deletion	Implement a Lubybased restart + “glue” score forgetting policy to keep memory bound without sacrificing proof quality.
Parallel prototype	Start with a fork () based portfolio (independent VSIDS seeds) to measure parallel speedups before porting kernels to CUDA.
Document polish	Convert tables to LaTeX or Word-style “Table X” with captions; ensure figure and section references are hyperlinked.

5. Conclusion

5.1 Summary of Key Findings.

The project demonstrated that the stability and competitiveness of a modern SAT solver hinge equally on algorithmic sophistication and rigorous defensive programming. Memory safety guards (`ensureVar`/`ensureVarCapacity`) and aggressive assertions eradicated segmentation faults that surfaced once clause learning enlarged the variable universe beyond initial DIMACS bounds. Precision in decision level bookkeeping prevented illegal back jumps, while duplicate detection in the watched literal queue eliminated latent trail corruption. Tuning VSIDS decay frequency, consolidating runtime logging, and enforcing cross platform line ending hygiene further transformed a brittle prototype into a production grade solver that completes tens of thousands of benchmarks without memory violations or heuristic drift.

5.2 Reflection on heuristic effectiveness.

Among the heuristics evaluated, three emerged as particularly influential. Watched literals delivered the largest raw speedup in Boolean Constraint Propagation by reducing clause scans to two sentinel positions, an improvement that remained robust once duplicate enqueues were suppressed. VSIDS, after its decay schedule was realigned to the MiniSATstyle 1 ∶ 100 cadence, provided the best overall search guidance—cutting average decision counts by 35 % on random3SAT and by 42 % on industrial instances. Clause learning with nonchronological backtracking added the decisive edge on the hardest UNSAT problems, slashing proof lengths by an order of magnitude; but its benefits were contingent on the aforementioned memory safety fixes. Pure Literal Elimination, in contrast, offered negligible incremental gain once the other heuristics were active and was therefore left disabled in the default build.

5.3 Lessons Learned

Developing the solver quickly revealed that robust systems programming is as critical as sound algorithm design. Early prototypes allocated fixed length vectors sized only from the DIMACS header, so clause learning occasionally introduced literals whose indices exceeded the original bound, overrunning the _val and _watchList arrays and producing nondeterministic segmentation faults. Wrapping every access in ensureVar/ensureVarCapacity guards, followed

by hard assertions after each resize, eliminated all Valgrinddetected writes beyond bounds. A second flaw lay in the decision level bookkeeping: an off Byone in _levelPos allowed back jumping to level −1, triggering undefined behavior in std::vector::operator[]. Replacing raw arithmetic with an explicit clamp—while emitting a logger warning—kept the program alive long enough to expose the higher level logic error that requested the illegal jump.

Other defects emerged in the interaction of heuristics and infrastructure. In the watched literal engine, a literal could be pushed onto the unit queue twice before its first instance was serviced; the duplicate assignment was then popped only once, corrupting the trail. A linear time duplicate check, innocuous given the queue's small size, resolved the issue. The initial VSIDS implementation decayed activity scores after every conflict, rapidly driving the floating point weights toward zero and destroying heuristic discrimination; throttling decay to once per hundred conflicts restored behavior consistent with MiniSAT's empirical tuning. Operational concerns also surfaced: per instance timestamped logging generated hundreds of tiny files in long runs, so the logger was refactored to append all traces to a single solver.log within a run tagged directory, with optional pushFile() calls retained for targeted debugging. Finally, shell scripts edited on Windows retained CRLF line endings that Bash refused despite the executable bit; a repository root sanitation pass (sed -i 's/\r$//') now standardizes text files on commit.

Each episode reinforced three practices that now underpin the codebase: apply surgical, unit tested patches; fail fast through aggressive assertions to convert silent corruption into reproducible crashes; and rely on structured, hierarchical logging for postmortem analysis without overwhelming storage. Adhering to these principles transformed a fragile prototype into a solver capable of running tens of thousands of benchmarks without memory errors or heuristic degradation.

5.4 Closing Remarks

The investigation underscores three enduring lessons for solver development: implement small, test backed patches to localize risk; fail fast via assertions to expose corruption early; and maintain structured, low overhead logging to accelerate diagnosis without compromising throughput. Adhering to these principles allowed the team to reconcile performance ambition with software robustness, yielding a solver that is both fast and demonstrably reliable—a

foundation that can now support deeper experimentation with advanced learning schemes and portfolio strategies.

Appendix A. Use of Generative AI

OpenAI ChatGPT (model **o3**) was consulted during development for design sketches, debugging strategies, shell script templates, and wording suggestions for this report. Typical prompts included *"rewrite my logger so each bulk run gets one file"* and *"explain VSIDS bump/decay and how often to apply global decay."* No code was pasted verbatim; every suggestion was reviewed, hand translated into C++, compiled, and validated by the project authors. Thus, ChatGPT functioned strictly as an assistant for ideation and proofreading, not an automated code generator.