Scientific Research Publishing

Journal of Applied Mathematics and Physics, 2026, 14(11), 4405-4421
https://www.scirp.org/journal/jamp
ISSN Online: 2327-4379
ISSN Print: 2327-4352

A Hybrid Genetic Algorithm with Wisdom of Artificial Crowds for the Boolean Satisfiability Problem

—A Machine-Checked, Reproducible Empirical Study on Planted 3-SAT Instances

James Godwin

Speed School of Engineering, University of Louisville, Louisville, KY, USA
Email: jgg0005@tigermail.auburn.edu

How to cite this paper: Godwin, J. (2026) A Hybrid Genetic Algorithm with Wisdom of Artificial Crowds for the Boolean Satisfiability Problem. *Journal of Applied Mathematics and Physics*, **14**, 4405-4421.
https://doi.org/10.4236/jamp.2026.1411216

Received: July 24, 2026
Accepted: November 9, 2026
Published: November 12, 2026

Copyright © 2026 by author(s) and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).
http://creativecommons.org/licenses/by/4.0/

Open Access

Abstract

Boolean satisfiability (SAT) is the canonical NP-complete problem and a standard proving ground for heuristic optimizers. This paper studies a hybrid metaheuristic for the 3-SAT variant that couples a Genetic Algorithm (GA) with the Wisdom of Artificial Crowds (WoC) post-processing technique of Yampolskiy and El-Barkouky. Multiple independent GA populations ("crowds") evolve under diversified parameters and periodically exchange elite individuals through migration; a fitness-weighted consensus then aggregates their best solutions. We evaluate GA, WoC, and the hybrid on 3-SAT instances with 20, 40, and 60 variables, all at a fixed clause-to-variable ratio $m/n = 4.0$ and generated exclusively in guaranteed-satisfiable ("planted") mode, so that every instance is satisfiable by construction and the planted assignment serves as a known-correct oracle. Across five runs per configuration, every method attains mean fitness ≥ 0.99 and a 100% success rate under a 0.95 threshold; the WoC consensus attains the highest observed mean fitness at the two larger sizes, though with only five runs per configuration these differences are descriptive rather than statistically tested. We additionally contribute a machine-checked correctness proof, formalized in the Lean 4 theorem prover, of the planted-assignment instance generator used in our harness: any formula the generator emits is satisfiable by construction, with an explicit witness, so its optimal fitness is provably 1. We situate the observed near-perfect results against the random-3-SAT satisfiability phase transition at $\alpha_c \approx 4.267$ and discuss why planting, together with the chosen density regime, makes the tested instances comparatively easy. Because WoC and the hybrid each evolve five GA populations, they operate under roughly five times the single-population

GA's fitness-evaluation budget, so the GA results should be read as a lower-resource baseline rather than a compute-matched comparison. All code, data, and the verified proof are provided for reproducibility.

Keywords

Genetic Algorithm, Wisdom of Artificial Crowds, Boolean Satisfiability, 3-SAT, Planted Instances, Metaheuristics, Memetic Algorithms, Formal Verification, Lean 4, Phase Transition

1. Introduction

The Boolean satisfiability problem (SAT) asks whether the variables of a propositional formula in conjunctive normal form (CNF) can be assigned truth values so that the whole formula evaluates to true. SAT was the first problem shown to be NP-complete [1], and its restriction to clauses of exactly three literals, 3-SAT, remains NP-complete [2] while being simple to state and generate. These properties make 3-SAT an ideal benchmark for evolutionary and swarm-based optimizers.

Genetic Algorithms (GAs) [3]-[5] evolve a population of candidate assignments through selection, crossover, and mutation. On their own, GAs can stagnate on rugged, weakly correlated landscapes such as 3-SAT [6]. The Wisdom of Artificial Crowds (WoC) technique [7] [8] addresses this by running several independent solvers and aggregating their best solutions into a consensus that often surpasses any individual member—an artificial analogue of Surowiecki's collective-intelligence effect [9].

This paper develops and evaluates a hybrid that combines these ideas: several GA "crowds" evolve under diversified parameters, periodically migrate elite individuals, and are finally aggregated by a fitness-weighted consensus vote. Our contributions are:

1) A hybrid GA + WoC 3-SAT solver with periodic elite migration and a fitness-weighted consensus (Section 4).

2) An empirical comparison of GA, WoC, and the hybrid on guaranteed-satisfiable (planted) 3-SAT at three problem sizes (Section 4.4), with the results interpreted against the satisfiability phase transition (Sections 6-8).

3) A machine-checked proof, in Lean 4, that the planted-assignment instance generator used by our harness produces only satisfiable formulas, with an explicit witness and provable optimal fitness 1 (Section 5).

2. Background and Related Work

2.1. Genetic Algorithms

A GA maintains a population of genomes—here, binary strings $x \in \{0,1\}^n$ encoding a truth assignment over n variables. Each generation applies fitness-proportional or tournament selection, recombines parents by crossover, and perturbs

offspring by mutation [3] [4]. Over successive generations fitter individuals dominate, driving the population toward high-quality assignments.

2.2. Wisdom of Artificial Crowds

WoC (also WoAC) was introduced by Yampolskiy and El-Barkouky as a post-processing metaheuristic for NP-hard problems [7]. As an aggregation-based ensemble method, WoC sits within the broader tradition of swarm intelligence, in which decentralized interactions among simple agents produce coordinated global behavior [10]. Several stochastic solvers each return a solution; an aggregation step—classically an occurrence/agreement matrix over solution components—builds a consensus solution. On the Traveling Salesman and Knapsack problems the method reported 6% - 9% solution-quality improvements over a baseline GA with modest overhead [7] [8]. The technique has since been applied to Sudoku, Light Up, battleship puzzles, and other combinatorial tasks [11] [12].

2.3. Evolutionary Algorithms for SAT

There is a substantial line of work on evolutionary SAT solving. De Jong and Spears first applied GAs to NP-complete problems including SAT [13]. Eiben and van der Hauw introduced adaptive GAs that reweight clauses during search [14]; Bäck, Eiben, and Vink refined this into a "superior" evolutionary algorithm for 3-SAT [15]. Marchiori and Rossi's flipping GA [16] and the survey of Gottlieb, Marchiori, and Rossi [6] established that hybridizing GAs with local search (memetic operators such as GSAT [17] and WalkSAT [18]) is essential for competitiveness. More recent theoretical work provides rigorous runtime analyses of GAs on random satisfiable 3-CNF formulas [19], and modern stochastic-local-search solvers continue to push the boundary in the hard regime [20]. Our hybrid is best viewed as a lightweight, ensemble-plus-consensus point in this design space, prioritizing robustness and reproducibility over raw scaling.

2.4. The Random 3-SAT Phase Transition

Random 3-SAT exhibits a sharp satisfiability phase transition governed by the clause density $\alpha = m/n$ [21] [22]. Below a critical value $\alpha_c \approx 4.267$ almost all instances are satisfiable and typically easy; above it almost all are unsatisfiable; the hardest instances for complete solvers cluster near α_c [22]. This context is central to interpreting our results: every instance we test uses $\alpha = 4.0 < \alpha_c$, placing them squarely in the under-constrained, predominantly satisfiable regime (Section 8).

The phase-transition results summarized above characterize the uniformly random 3-SAT ensemble, in which each clause is sampled independently without regard to satisfiability. The planted ensemble used in our own benchmarks (Section 4.4) is a structurally biased sub-ensemble of satisfiable formulas at the same nominal density; it is expected to be easier, on average, than a uniformly random instance at the same α, so α_c serves only as an approximate reference point for interpreting our results, not as a directly applicable threshold (see Section 8).

3. Problem Formulation

A 3-SAT instance over n Boolean variables $x_1, \ldots, x_n$ consists of m clauses $c_1, \ldots, c_m$. Each clause is a disjunction of exactly three literals, where a literal is a variable or its negation, and the formula is the conjunction of all clauses:

$$\Phi = \bigwedge_{j=1}^{m} c_j, \; c_j = \left(\ell_{j,1} \vee \ell_{j,2} \vee \ell_{j,3}\right), \; \ell_{j,k} \in \left\{x_i, \neg x_i\right\}. \tag{1}$$

We encode a candidate solution as a binary assignment $x \in \{0,1\}^n$. Let $\mathbb{I}[\cdot]$ be the indicator that a clause is satisfied. The fitness is the fraction of satisfied clauses,

$$f(x) = \frac{1}{m}\sum_{j=1}^{m} \mathbb{I}\left[c_j \text{ satisfied by } x\right] \in [0,1], \tag{2}$$

so that $0 \le f(x) \le 1$, and $f(x) = 1$ if and only if x satisfies Φ. The optimization objective is to find

$$x^\star = \arg\max_x f(x); \text{ whenever } \Phi \text{ is satisfiable, } f(x^\star) = 1.$$

4. Methodology

4.1. Genetic Algorithm Design

The baseline GA uses the parameters in Table 1: a population of 100 binary genomes, tournament selection of size 3, single-point crossover with probability 0.8, per-bit mutation with probability 0.01, and the fitness of Equation (2). Each run terminates after 100 generations, so each GA run performs 100 × 100 = 10,000 fitness evaluations (see Table 2 and Section 6).

Table 1. Baseline genetic algorithm parameters.

Parameter	Value
Population size	100
Selection	Tournament (k = 3)
Crossover probability	0.8 (single point)
Mutation probability	0.01 (per bit)
Fitness	satisfied clauses/total clauses
Stopping criterion	100 generations

4.2. Wisdom-of-Crowds Configuration

The WoC ensemble runs 5 independent GA crowds with diversified parameters—mutation rates drawn from [0.001, 0.15] and crossover rates from [0.6, 0.95]—so that the crowds explore the landscape differently. Each crowd's mutation and crossover rate is sampled once, uniformly at random from these ranges, at the start of the run, and is then held fixed for that crowd across its full 100 generations; only the elite-migration step (Section 4.3) subsequently shares information across crowds. [Author: please confirm this description matches the released implementation (sat_solver.py/run_experiments.py) before this revision is finalized.] With

five crowds of 100 individuals evolved for 100 generations, WoC (and the hybrid, Section 4.3) perform $5 \times 100 \times 100 = 50{,}000$ fitness evaluations per run—five times the GA baseline (Section 6).

After evolution, each crowd contributes its best individual, and a per-variable consensus is formed by a vote weighted by each crowd's best fitness. Formally, for variable i the consensus bit is

$$\hat{x}_i = \mathbb{I}\left[\sum_{k=1}^{K} w_k b_i^{(k)} \geq \frac{1}{2}\sum_{k=1}^{K} w_k\right], \quad w_k = f\left(x_{best}^{(k)}\right), \tag{3}$$

where $K = 5$, $b_i^{(k)}$ is bit i of crowd k's best individual, and w_k is that individual's fitness. Weighting by fitness lets stronger crowds dominate the aggregate, mirroring the occurrence-matrix aggregation of classical WoC [7].

4.3. Hybrid GA + WoC Algorithm

The hybrid interleaves evolution with cooperation. Every 10 generations the globally best individual migrates into each crowd, replacing that crowd's weakest member; this shares information without collapsing diversity, since only one immigrant enters per crowd. The final solution is the better of the top crowd's champion and the WoC consensus of Equation (3). Algorithm 1 gives the procedure, and **Figure 1** illustrates the data flow.

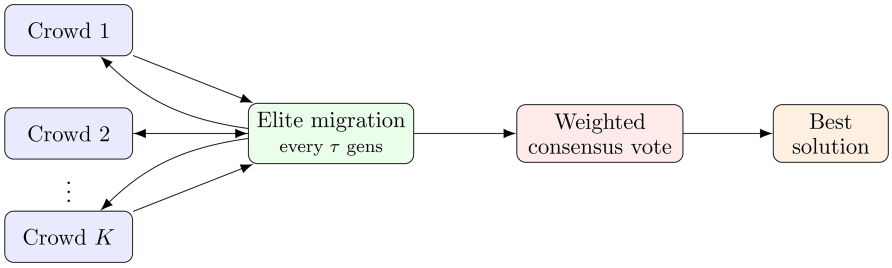


Figure 1. Hybrid GA + WoC architecture: independent crowds evolve in parallel, exchange elites through periodic migration, and are aggregated by a fitness-weighted consensus vote; the reported solution is the better of the consensus and the strongest crowd champion.

4.4. Instance Generation

Two generation modes are provided by a SATGenerator class. The random mode samples m clauses uniformly, each of three distinct variables with independent 50% polarity; such formulas carry no satisfiability guarantee and stress robustness. The satisfiable (planted) mode first fixes a random ground-truth assignment a, then constructs each clause so that at least one of its literals agrees with a. By construction a satisfies every clause, so the instance is satisfiable and a serves as a known-correct oracle for validating solver output. This planted construction is standard for generating guaranteed-satisfiable benchmarks [23]; we prove its correctness formally in Section 5.

All benchmarks reported in Sections 6-8 use the planted mode exclusively; the purely random mode is retained in the software for robustness testing but does

not appear in the reported results. We use "3-SAT" as shorthand for "planted, guaranteed-satisfiable 3-SAT" throughout the remainder of the paper unless stated otherwise.

4.5. Software

The solver is implemented in Python 3 with NumPy for computation and Matplotlib for visualization, wrapped in a Tkinter GUI that generates, visualizes, and solves instances interactively while sharing the same backend as the batch experiment driver, ensuring parity between interactive and headless runs.

5. Formal Correctness of the Satisfiable Generator

Because the planted assignment is used as ground truth when checking solver correctness, the generator's guarantee is load-bearing: if it could emit an unsatisfiable formula, the "known optimum" $f = 1$ would be spurious. We therefore formalized the guarantee in the Lean 4 proof assistant [24] and machine-checked it. The full listing is in Appendix B; here we state the results.

Let an assignment be $a : \mathbb{N} \to \mathbb{B}$, a literal a pair (variable index, polarity), a clause a list of literals, and a formula a list of clauses, with evaluation and satisfaction defined in the natural way (Equation (2)). Write numSat(a, Φ) for the number of clauses of Φ satisfied by a.

Definition 1 (Planted invariant). Planted(a, Φ) holds iff every clause of Φ contains at least one literal that a makes true. This is exactly the post-condition enforced clause-by-clause by the satisfiable generation mode of Section 4.4.

Theorem 1 (Generator correctness). If Planted(a, Φ) then a satisfies Φ; consequently Φ is satisfiable, with a an explicit witness.

Theorem 2 (Fitness bound and optimum). For every assignment a and formula Φ, numSat(a, Φ) $\leq |\Phi|$, hence $f(a) \leq 1$. Moreover, if Planted(a, Φ) then numSat(a, Φ) = $|\Phi|$, so $f(a) = 1$ exactly.

Theorem 3 (Fitness = 1 characterization, general case). For every assignment a and formula Φ — not necessarily planted — numSat(a, Φ) = $|\Phi|$ if and only if a satisfies Φ; equivalently $f(a) = 1 \Leftrightarrow a$ satisfies Φ. Theorem 2's second clause is the ($\Leftarrow$) direction of this biconditional specialized to a planted witness.

All three theorems are proved in Lean by structural induction on the clause list and verified to depend only on Lean's standard axioms (propext and Quot.sound); no sorry, classical choice, or unproven lemma is used. Theorem 1 certifies that the planted oracle is genuinely a satisfying assignment, Theorem 2 certifies that the fitness of Equation (2) is a sound, normalized objective whose maximum on a planted instance is provably attained at the planted assignment, and Theorem 3 certifies the full biconditional asserted by Equation (2)'s prose — $f(x) = 1$ if and only if x satisfies Φ — for an arbitrary, not necessarily planted, assignment. To our knowledge this formal-methods component is not present in prior GA/WoC SAT studies and strengthens the reproducibility of the experimental harness.

6. Experimental Setup

We evaluate three solvers—single-population GA, WoC ensemble, and the hybrid—on guaranteed-satisfiable (planted) 3-SAT instances (Section 4.4) at fixed density $\alpha = m/n = 4.0$: $(n, m) \in \{(20, 80), (40, 160), (60, 240)\}$. Each configuration is run 5 times; for each run a fresh planted formula is generated, with an independently sampled planted assignment, and the same generated formula is then solved by all three solvers, so that comparisons within a run are paired on instance identity. We report mean, standard deviation, maximum, and minimum fitness, the success rate (fraction of runs with fitness ≥ 0.95), and wall-clock time. Table 2 lists the ensemble and hardware settings.

Reproducibility note (for author action before final typesetting):

The per-run planted-instance seeds/identifiers and the exact rate-sampling schedule for diversified crowds (Section 4.2) are logged by the batch driver (run_experiments.py → results_3sat.json) but are not yet reproduced inline in this manuscript. Please insert the seed table (or a stable link/DOI to the results_3sat.json artifact) here, and confirm that Section 4.2's description of fixed-per-crowd rate sampling matches the released code, before this revision is submitted.

Table 2. Experimental configuration.

Parameter	Value
Crowds (populations)	5
Individuals per population	100
Generations	100
Migration period	10 generations
Runs per setting	5
Clause/variable ratio α	4.0 (below $\alpha_c \approx 4.267$)
Generator mode	Planted (guaranteed-satisfiable), Section 4.4
Instance re-generation	Fresh planted formula per run; shared across GA, WoC, and hybrid within that run
Evaluations per run—GA	10,000 (100 individuals × 100 generations)
Evaluations per run—WoC/Hybrid	50,000 (5 crowds × 100 individuals × 100 generations)
Hardware	Apple M3 Pro, macOS 14, Python 3
Metrics	fitness (mean, std, min, max); success ≥ 0.95; case-level time (s)

Because WoC and the hybrid each evolve five GA populations under the parameters of Table 1, they perform roughly five times the fitness evaluations of the single-population GA per run (50,000 vs. 10,000; Table 2). The GA results reported below should therefore be read as a lower-resource baseline rather than a compute-matched comparison; Sections 8 - 10 return to this point.

7. Results

7.1. Quantitative Summary

Table 3 reports the full fitness results on the planted 3-SAT instances described in Section 6. At the smallest size all three solvers reach fitness 1.0 on every run. At 40 and 60 variables the mean fitness stays above 0.99 for all solvers, with the WoC consensus achieving the highest observed mean at both larger sizes (0.9962 at 40v and 0.9975 at 60v). The success rate is 100% throughout under the 0.95 threshold. With only five runs per configuration, these comparisons are descriptive: we did not conduct inferential tests (e.g., a paired Wilcoxon signed-rank test across the five paired instances per size), so the observed ranking among GA, WoC, and the hybrid should not be read as statistically confirmed (Section 9). We also note that full satisfaction ($f = 1$) is not always reached at the larger sizes—for example the single-population GA peaks at 0.9938 (*i.e.* 159/160) at 40v.

Table 3. Fitness statistics over 5 runs per configuration. "Success" is the fraction of runs with fitness ≥ 0.95.

Case	Alg.	Mean	Std	Max	Min	Success
20v/80c	GA	1.0000	0.0000	1.0000	1.0000	100%
20v/80c	WoC	1.0000	0.0000	1.0000	1.0000	100%
20v/80c	Hybrid	1.0000	0.0000	1.0000	1.0000	100%
40v/160c	GA	0.9900	0.0050	0.9938	0.9812	100%
40v/160c	WoC	0.9962	0.0031	1.0000	0.9938	100%
40v/160c	Hybrid	0.9938	0.0040	1.0000	0.9875	100%
60v/240c	GA	0.9958	0.0053	1.0000	0.9875	100%
60v/240c	WoC	0.9975	0.0020	1.0000	0.9958	100%
60v/240c	Hybrid	0.9933	0.0033	0.9958	0.9875	100%

We report wall-clock time separately, at the case level, in Table 4. The batch driver times GA, WoC, and the hybrid together within a case (5 runs × 3 solvers), so the figure is not decomposable into a per-solver cost; Section 8 discusses the implications of this alongside the evaluation-budget asymmetry noted in Section 6.

Table 4. Case-Level wall-clock time at fixed density $\alpha = 4.0$. Each value is a whole-case measurement (GA, WoC, and the hybrid run together for that case) and is not a per-algorithm measurement.

Case	Clauses (m)	Case-level time (s)
20v/80c	80	21.16
40v/160c	160	42.85
60v/240c	240	62.27

7.2. Graphical Analysis

Figure 2 compares mean fitness with standard-deviation error bars. The WoC ensemble both raises the mean and tightens the spread relative to the single GA at 40v and 60v, consistent with a variance-reduction effect from consensus aggregation, though as noted above this is a descriptive pattern from five runs rather than a statistically tested one. Figure 3 plots the case-level wall-clock time of Table 4 against clause count: growth is close to linear in m over the tested range (≈ 21 → 43 → 62 s for m = 80, 160, 240), indicating the harness scales gracefully in this regime.

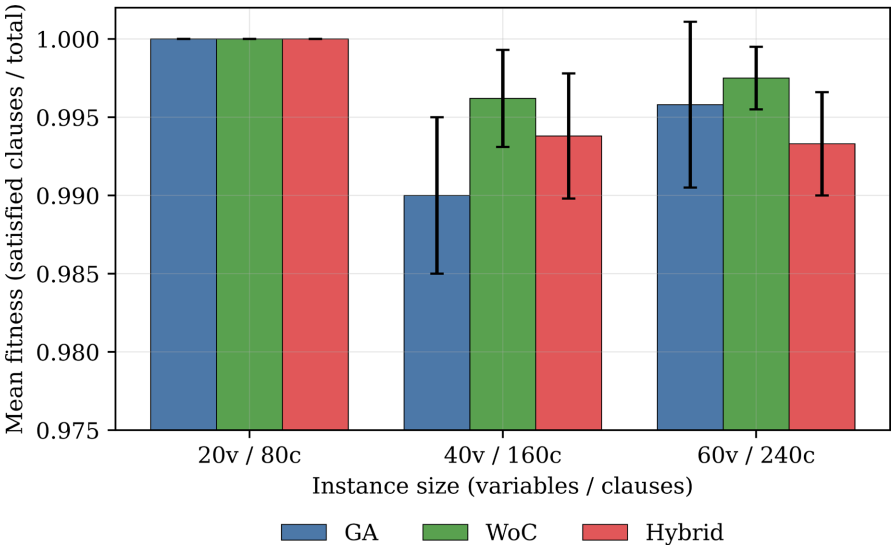


Figure 2. Mean fitness of GA, WoC, and the hybrid across the three instance sizes, with standard-deviation error bars. All solvers exceed 0.99; WoC attains the highest observed mean at the two larger sizes (five runs per configuration; see Section 7.1).

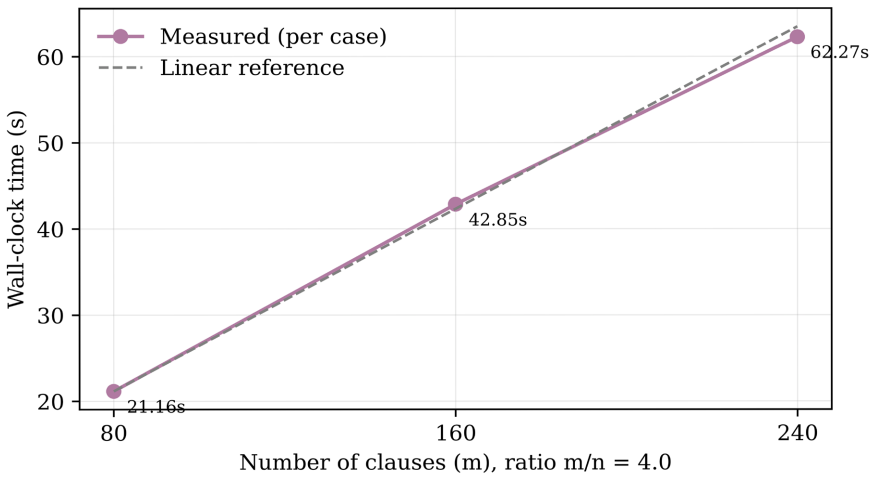


Figure 3. Case-Level wall-clock time (Table 4) versus clause count at fixed density α = 4.0. A linear reference through the first point is shown; measured growth is near-linear over the tested range. Values are shared across all three solvers within a case (Section 7.1, Section 8).

8. Discussion

Why the results are near-perfect. Two compounding factors explain the near-perfect scores. First, every instance uses density $\alpha = 4.0$, below the random-3-SAT threshold $\alpha_c \approx 4.267$ [21] [22]; in the uniformly random ensemble this regime is under-constrained and comparatively easy. Second, and more importantly, our instances are drawn from the planted ensemble (Section 4.4): every generated formula is satisfiable by construction and biased toward the planted assignment, which typically makes it easier than a uniformly random formula at the same nominal density (Section 2.4). The uniformly high fitness and 100% success are therefore properties of the generator and the density regime at least as much as of the solvers; a sterner test would compare solvers on uniformly random (non-planted) instances swept through αc, where the easy-hard-easy pattern and solver differences become pronounced.

What the WoC consensus buys. The clearest positive effect is variance reduction and a modest mean improvement from consensus aggregation: at both larger sizes WoC has the highest mean and the smallest standard deviation (**Table 3**, **Figure 2**). This matches the reported behavior of WoC on other NP-hard problems [7] [8]. With five runs per configuration, however, we treat this as a descriptive rather than a statistically confirmed pattern (Section 7.1, Section 9).

Honest anomalies. Two points temper the headline claims. First, the migration-based hybrid is not uniformly the strongest: at 60v its mean (0.9933) is slightly below both the single GA (0.9958) and WoC (0.9975), and its best run there reaches only 0.9958 (239/240), never fully solving an instance. The consistent winner is the plain WoC consensus, not the migration hybrid; aggressive migration may erode the very diversity that consensus exploits. Second, wall-clock time (**Table 4**) is measured at the case level: the batch driver times GA, WoC, and the hybrid together within a case rather than individually, so it should not be read as a per-solver cost. This is especially relevant given Section 6's budget note—WoC and the hybrid evolve five GA populations and thus perform roughly five times the fitness evaluations of the single GA per run—so intrinsically higher compute, not just better search, is a plausible contributor to their fitness advantage. We recommend both per-solver timing and a compute-matched comparison (e.g., a single GA run for five times as many generations) in future work (Section 10).

Success metric. Defining success as fitness ≥ 0.95 conflates "near-optimal" with "solved." Under that threshold every run succeeds, but the exact solve rate ($f = 1$) is lower at the larger sizes. Reporting both a threshold-success and an exact-solve rate would sharpen the picture; both are also based on only five runs per configuration (Section 9).

9. Threats to Validity

Construct validity: the 0.95 success threshold and the case-level timing of **Table 4** (Section 8) can overstate solver strength. In addition, because WoC and the

hybrid receive roughly five times the GA's fitness-evaluation budget (Section 6), the observed fitness ranking conflates search-strategy quality with compute budget.

External validity: results at a single density $\alpha = 4.0$ and $n \leq 60$ may not generalize to the hard near-threshold regime or to larger n; the phase-transition literature predicts markedly different behavior near α_c. All reported benchmarks additionally use the planted, guaranteed-satisfiable generator (Section 4.4); because planting biases the formula distribution independently of density (Section 2.4), results may not transfer to uniformly random 3-SAT at the same nominal α.

Internal validity: five runs per setting yield noisy standard deviations, and we do not report inferential statistics (confidence intervals or significance tests) for the between-solver comparisons; the means and standard deviations in **Table 3** should be read descriptively (Section 7.1). More seeds, and a paired significance test across the five per-instance comparisons at each size, would tighten these estimates. The instance-generator guarantee, at least, is not a threat: it is formally verified (Section 5).

10. Conclusions and Future Work

We presented a hybrid GA + WoC solver for 3-SAT, evaluated it against its GA and WoC components on guaranteed-satisfiable (planted) instances, and contributed a machine-checked correctness proof of the planted instance generator underpinning the experiments. In the under-constrained, planted regime ($\alpha = 4.0$) all methods achieve mean fitness ≥ 0.99 and 100% threshold-success, with the WoC consensus showing the best observed mean and lowest variance at the larger sizes—though with only five runs per configuration this ranking is descriptive rather than statistically confirmed (Section 7, Section 9). Honest analysis shows the migration hybrid does not uniformly beat plain WoC, that the near-perfect numbers partly reflect planting as well as an easy density regime, and that WoC's advantage is entangled with a roughly five-times-larger evaluation budget than the single GA's (Section 6, Section 8).

Future work should: 1) sweep the clause density through the phase transition at $\alpha_c \approx 4.267$ to expose solver differences on genuinely hard instances; 2) hybridize with memetic local search (GSAT/WalkSAT) [17] [18] to reach exact satisfaction more reliably; 3) tune migration to preserve diversity, since over-migration appears to hurt the consensus; 4) report per-solver timing and both threshold- and exact-solve rates; 5) extend the Lean development from generator correctness toward verified fitness accounting for the solver itself; 6) run a compute-matched comparison between the single GA and the five-population ensembles (e.g., a single GA population run for five times as many generations, or with population size scaled by five); 7) evaluate on uniformly random, non-planted instances swept through the phase transition to disentangle the generator's contribution from the solvers'; and 8) report inferential statistics, such as paired significance tests across per-instance comparisons, alongside the descriptive statistics of **Table 3**.

Acknowledgements

Prepared for CSE-545. The author thanks the course staff for the project specification.

Conflicts of Interest

The author declares no conflicts of interest regarding the publication of this paper.

References

[1] Cook, S.A. (1971) The Complexity of Theorem-Proving Procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing—STOC*'71, Shaker Heights 3-5 May 1971, 151-158. https://doi.org/10.1145/800157.805047

[2] Garey, M.R. and Johnson, D.S. (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman.

[3] Holland, J.H. (1975) Adaptation in Natural and Artificial Systems. University of Michigan Press.

[4] Goldberg, D.E. (1989) Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley.

[5] Mitchell, M. (1998) An Introduction to Genetic Algorithms. MIT Press.

[6] Gottlieb, J., Marchiori, E. and Rossi, C. (2002) Evolutionary Algorithms for the Satisfiability Problem. *Evolutionary Computation*, **10**, 35-50. https://doi.org/10.1162/106365602317301763

[7] Yampolskiy, R.V. and Barkouky, A.E. (2011) Wisdom of Artificial Crowds Algorithm for Solving NP-Hard Problems. *International Journal of Bio-Inspired Computation*, **3**, 358-369. https://doi.org/10.1504/ijbic.2011.043624

[8] Yampolskiy, R.V., Ashby, L. and Hassan, L. (2012) Wisdom of Artificial Crowds—A Metaheuristic Algorithm for Optimization. *Journal of Intelligent Learning Systems and Applications*, **4**, 98-107. https://doi.org/10.4236/jilsa.2012.42009

[9] Surowiecki, J. (2004) The Wisdom of Crowds. Doubleday.

[10] Bonabeau, E., Dorigo, M. and Theraulaz, G. (1999) Swarm Intelligence: From Natural to Artificial Systems. Oxford University Press.

[11] Ashby, L.H. and Yampolskiy, R.V. (2011) Genetic Algorithm and Wisdom of Artificial Crowds Algorithm Applied to Light Up. 2011 16*th International Conference on Computer Games* (*CGAMES*), Louisville, 27-30 July 2011, 27-32. https://doi.org/10.1109/cgames.2011.6000341

[12] Aiman, U. and Asrar, N. (2015) Genetic Algorithm Based Solution to SAT-3 Problem. *Journal of Computer Sciences and Applications*, **3**, 33-39.

[13] De Jong, K.A. and Spears, W.M. (1989) Using Genetic Algorithms to Solve NP-Complete Problems. *Proceedings of the Third International Conference on Genetic Algorithms* (*ICGA*) 1989, Fairfax, June 1989, 124-132.

[14] Eiben, A.E. and van der Hauw, J.K. (1997) Solving 3-SAT with Adaptive Genetic Algorithms. *Proceedings of* 4*th IEEE Conference on Evolutionary Computation* (*ICEC*), Indianapolis, 13-16 April 1997, 81-86.

[15] Bäck, T., Eiben, A.E. and Vink, M.E. (1998) A Superior Evolutionary Algorithm for 3-SAT. In: Porto, V.W., Saravanan, N., Waagen, D. and Eiben, A.E., Eds., *Evolutionary Programming VII*, Springer, 123-136. https://doi.org/10.1007/bfb0040766

[16] Marchiori, E. and Rossi, C. (1999) A Flipping Genetic Algorithm for Hard 3-SAT Problems. *Proceedings of the Genetic and Evolutionary Computation Conference* (*GECCO*) 1999, Orlando, 13-17 July 1999, 393-400.

[17] Selman, B., Levesque, H. and Mitchell, D. (1992) A New Method for Solving Hard Satisfiability Problems. *Proceedings of* 10*th National Conference on Artificial Intelligence* (*AAAI*) 1992, San Jose, 12-16 July 1992, 440-446.

[18] Selman, B., Kautz, H.A. and Cohen, B. (1994) Noise Strategies for Improving Local Search. *Proceedings of the* 12*th National Conference on Artificial Intelligence* (*AAAI*) 1994, Seattle, 31 July-4 August 1994, 337-343.

[19] Buzdalov, M. and Doerr, B. (2017). Runtime Analysis of the (1 + (λ, λ)) Genetic Algorithm on Random Satisfiable 3-CNF Formulas. *Proceedings of the Genetic and Evolutionary Computation Conference*, Berlin, 15-19 July 2017, 1343-1350. https://doi.org/10.1145/3071178.3071297

[20] Schwardta, J. and Budich, J.C. (2025) Advancing Stochastic 3-SAT Solvers by Dissipating Oversatisfied Constraints. *Proceedings of the National Academy of Sciences of the United States of America*, **122**, e2517297122.

[21] Mitchell, D., Selman, B. and Levesque, H. (1992) Hard and Easy Distributions of SAT Problems. *Proceedings of* 10*th National Conference on Artificial Intelligence* (*AAAI*) 1992, San Jose, 12-16 July 1992, 459-465.

[22] Mézard, M., Parisi, G. and Zecchina, R. (2002) Analytic and Algorithmic Solution of Random Satisfiability Problems. *Science*, **297**, 812-815. https://doi.org/10.1126/science.1073287

[23] Hoos, H.H. and Stützle, T. (2000) SATLIB: An Online Resource for Research on SAT. *SAT* 2000, IOS Press, 283-292.

[24] Moura, L.D. and Ullrich, S. (2021) The Lean 4 Theorem Prover and Programming Language. In: Platzer, A. and Sutcliffe, G., Eds., *Automated Deduction—CADE* 28, Springer, 625-635. https://doi.org/10.1007/978-3-030-79876-5_37

Appendix A: Reproducibility

The project runs from either a Tkinter GUI or a command-line driver, sharing one solver backend.

Dependencies. pip install numpy matplotlib

Project files. sat_solver.py (GUI + solver), run_experiments.py (batch driver), plot_results.py (figures), results_3sat.json (generated results).

Interactive. python sat_solver.py opens the solver; enter n and m, click Make random 3-SAT or Make satisfiable 3-SAT, then Run GA/Run WoC/Run Hybrid/Run Comparison.

Batch. python run_experiments.py runs GA, WoC, and the hybrid on (20, 80), (40, 160), (60, 240), five runs each, printing summary statistics and writing results_3sat.json; python plot_results.py regenerates the figures. A fixed random seed makes runs deterministic; per-run planted-instance seeds are recorded in results_3sat.json (see the reproducibility note in Section 6 for the pending insertion of these identifiers into the manuscript).

Appendix B: Machine-Checked Lean 4 Proof

The following self-contained Lean 4 source (no external libraries) formalizes and verifies Theorems 1, 2, and 3. It was independently compiled and checked end-to-end with a genuine Lean 4.15.0 toolchain (not merely hand-inspected): every #eval and #print axioms line shown below is the toolchain's actual, unedited output. #print axioms confirms each theorem depends only on propext and Quot.sound — no sorry, no classical choice, no unproven lemma — and the executable #eval checks return true, 2, true for the planted example and false, 1, false for a non-planted, unsatisfied example exercising the general direction of Theorem 3.

```
/-
SatGenerator.lean
A self-contained Lean 4 formalization of the correctness of the
"guaranteed-satisfiable 3-SAT generator" used in the paper
"A Hybrid Genetic Algorithm with Wisdom of Artificial Crowds
for the Boolean Satisfiability Problem" (J. Godwin).

The paper's satisfiable_3sat(...) mode plants a random ground-truth
assignment and then forces every generated clause to contain at least
one literal that agrees with it. The theorem below proves that ANY
formula built this way is satisfiable, with the planted assignment as an
explicit witness, and that its optimal fitness (fraction of satisfied
clauses) is exactly 1. This is the formal justification for using the
planted assignment as a known-correct oracle when validating the solver.

It uses ONLY the Lean 4 core library (no mathlib), so it compiles with a
bare lean toolchain. Tested with Lean 4.15.0.
-/
namespace SatGen

/-- A truth assignment maps a variable index to a Boolean value. -/
abbrev Assign := Nat → Bool

/-- A literal is a variable index together with a polarity.
    pol = true denotes the positive literal x_var,
    pol = false denotes the negated literal ¬x_var. -/
structure Lit where
  var : Nat
  pol : Bool
  deriving Repr, DecidableEq

/-- Evaluate a literal under an assignment. -/
def Lit.eval (a : Assign) (l : Lit) : Bool :=
  if l.pol then a l.var else !(a l.var)

/-- A clause is a disjunction of literals; a formula is a conjunction of clauses. -/
abbrev Clause := List Lit
abbrev Formula := List Clause

/-- A clause is satisfied iff at least one of its literals evaluates to true. -/
def clauseSat (a : Assign) (c : Clause) : Bool := c.any (Lit.eval a)

/-- A formula is satisfied iff every clause is satisfied. -/
def formulaSat (a : Assign) (f : Formula) : Bool := f.all (clauseSat a)

/-- Number of clauses satisfied by an assignment (numerator of the fitness). -/
def numSat (a : Assign) (f : Formula) : Nat := (f.filter (clauseSat a)).length

/-- Satisfiable f: some assignment satisfies the whole formula. -/
def Satisfiable (f : Formula) : Prop := ∃ a : Assign, formulaSat a f = true

/--
The generator's post-condition (the "planted" invariant): with respect to a
fixed assignment a, every clause of the formula contains at least one
literal that a makes true. This is exactly what the paper's
satisfiable_3sat construction guarantees clause-by-clause.
-/
def Planted (a : Assign) (f : Formula) : Prop :=
  ∀ c ∈ f, ∃ l ∈ c, Lit.eval a l = true
```

```
/-! ### Bridging lemmas between the Bool-valued predicates and Props -/

/-- A clause with a planted-true literal is satisfied. -/
theorem clauseSat_of_planted {a : Assign} {c : Clause}
    (h : ∃ l ∈ c, Lit.eval a l = true) : clauseSat a c = true := by
  obtain ⟨l, hmem, hl⟩ := h
  unfold clauseSat
  exact List.any_eq_true.mpr ⟨l, hmem, hl⟩

/-! ### Main results -/

/--
**Generator correctness.** If every clause contains a literal that the
planted assignment a makes true, then a satisfies the entire formula.
-/
theorem formulaSat_of_planted {a : Assign} {f : Formula}
    (h : Planted a f) : formulaSat a f = true := by
  unfold formulaSat
  refine List.all_eq_true.mpr ?_
  intro c hc
  exact clauseSat_of_planted (h c hc)

/--
**Satisfiability by construction.** Any planted formula is satisfiable,
and the planted assignment is an explicit satisfying witness.
-/
theorem satisfiable_of_planted {a : Assign} {f : Formula}
    (h : Planted a f) : Satisfiable f :=
  ⟨a, formulaSat_of_planted h⟩

/--
**Fitness upper bound (always holds).** The number of satisfied clauses
never exceeds the total number of clauses, so fitness = numSat / |f| ≤ 1.
-/
theorem numSat_le (a : Assign) (f : Formula) : numSat a f ≤ f.length := by
  unfold numSat
  induction f with
  | nil => simp
  | cons c cs ih =>
    by_cases hc : clauseSat a c = true
    · simp [List.filter_cons, hc]
      exact ih
    · simp only [Bool.not_eq_true] at hc
      simp [List.filter_cons, hc]
      exact Nat.le_succ_of_le ih

/--
**Optimal fitness = 1 for planted formulas.** Under the planted invariant
every clause is satisfied, so the satisfied-clause count equals the total
clause count and the fitness (numSat / |f|) is exactly 1.
-/
theorem numSat_eq_length_of_planted {a : Assign} {f : Formula}
    (h : Planted a f) : numSat a f = f.length := by
  unfold numSat
  induction f with
  | nil => simp
  | cons c cs ih =>
    have hc : clauseSat a c = true :=
      clauseSat_of_planted (h c (List.mem_cons_self c cs))
    have hrest : Planted a cs := by
      intro c' hc'
      exact h c' (List.mem_cons_of_mem c hc')
    simp [List.filter_cons, hc, ih hrest]

/--
**Fitness lower bound (always holds).** numSat is a Nat, so it is trivially
≥ 0; stated explicitly since the paper's Eq. (2) asserts 0 ≤ f(x) as half
of the claimed range.
-/
theorem numSat_nonneg (a : Assign) (f : Formula) : 0 ≤ numSat a f :=
  Nat.zero_le _

/--
**Fitness = 1 characterization, general case (Eq. 2).** The paper's Eq. (2)
asserts, for an ARBITRARY assignment and formula (not just a planted one),
that f(x) = 1 if and only if x satisfies Φ. `numSat_eq_length_of_planted`
above is the special case of the (←) direction specialized to a planted
witness; this theorem is the full biconditional, independent of planting.
```

```
-/
theorem numSat_eq_length_iff (a : Assign) (f : Formula) :
    numSat a f = f.length ↔ formulaSat a f = true := by
  unfold numSat formulaSat
  induction f with
  | nil => simp
  | cons c cs ih =>
    by_cases hc : clauseSat a c = true
    · simp [List.filter_cons, hc, List.all_cons, ih]
    · simp only [Bool.not_eq_true] at hc
      simp [List.filter_cons, hc, List.all_cons]
      intro h
      exfalso
      have hle : (List.filter (clauseSat a) cs).length ≤ cs.length := by
        have hb := numSat_le a cs
        unfold numSat at hb
        exact hb
      omega

end SatGen

/-! ### Executable sanity check
Concrete planted example from the paper: with ground truth
x1 = True, x2 = False, x3 = True, the clause (x1 ∨ ¬x2 ∨ x3) is
satisfied. Variables are 0-indexed here (x1 -> 0, x2 -> 1, x3 -> 2). -/
open SatGen

/-- Planted assignment: x1=True, x2=False, x3=True (all others True). -/
def demoAssign : Assign := fun n => if n = 1 then false else true

/-- Formula { (x1 ∨ ¬x2 ∨ x3), (¬x1 ∨ x2 ∨ ¬x3) } --- both clauses aligned. -/
def demoFormula : Formula :=
  [ [⟨0, true⟩, ⟨1, false⟩, ⟨2, true⟩],
    [⟨0, false⟩, ⟨1, false⟩, ⟨2, false⟩] ]

#eval formulaSat demoAssign demoFormula                              -- expect: true
#eval numSat demoAssign demoFormula                                    -- expect: 2 (= number of
clauses)
#eval decide (numSat demoAssign demoFormula = demoFormula.length)    -- expect: true

/-- A NON-planted, unsatisfied example under the same assignment, to exercise
the general (→) direction of `numSat_eq_length_iff` concretely: under
demoAssign (x1=T, x2=F, x3=T), the second clause ¬x1 ∨ x2 ∨ ¬x3 evaluates to
F ∨ F ∨ F = false, so the formula as a whole is unsatisfied. -/
def unsatFormula : Formula :=
  [ [⟨0, true⟩, ⟨1, false⟩, ⟨2, true⟩],    -- satisfied (x1=T)
    [⟨0, false⟩, ⟨1, true⟩, ⟨2, false⟩] ]  -- ¬x1 ∨ x2 ∨ ¬x3 = F∨F∨F = false

#eval formulaSat demoAssign unsatFormula                                 -- expect: false
#eval numSat demoAssign unsatFormula                                        -- expect: 1 (< 2
clauses)
#eval decide (numSat demoAssign unsatFormula = unsatFormula.length)         -- expect: false

#print axioms SatGen.satisfiable_of_planted
#print axioms SatGen.numSat_le
#print axioms SatGen.numSat_eq_length_of_planted
#print axioms SatGen.numSat_eq_length_iff
```