AGENTIC ALGORITHM DESIGN: LLMS THAT PROPOSE, FORMALISE, AND PROVE RUNTIME BOUNDS FOR EVOLUTIONARY ALGORITHMS IN LEAN

Anonymous authors
Paper under double-blind review

ABSTRACT

LLMs now prove hard theorems in Lean. Can they also *design* algorithms and prove their correctness and efficiency? LLM-guided design of optimisation heuristics (such as AlphaEvolve) is purely empirical. Automated theorem provers with LLMs prove given statements. So far, there has been little evidence that LLMs can do both: design an algorithm and guarantee its efficiency by a mathematical proof. We present a staged pipeline modelled after research methodology in design and analysis of evolutionary algorithms. The agent proposes black-box optimisation algorithms, and Lean guards filter out flawed designs. The agent then defines a potential function and proves drift bounds that guarantee an expected runtime improvement over the baseline algorithm. At no point in the pipeline are the algorithms compared empirically. The agent generates a black-box optimisation algorithm with expected runtime $O(n^2)$ on a CYCLEUNCOVER problem, where the classical (1+1) EA needs runtime $e^{\Omega(n)}$. The pipeline also designs and analyses a black-box optimisation algorithm for integer-valued submodular minimisation with expected runtime $O(n^{2+2c})$, and it matches the optimal order $O(n^2)$ on LEADINGONES. Across 8 problem classes, 12 of 15 runs with frontier models complete the pipeline with a machine-checked expected-runtime bound. An ablation study shows that removing the staging and asking the agent to zero-shot an algorithmic design with a runtime guarantee leads to a copy of the seeded baseline algorithm (random search) or to incomplete proofs. Open-weight models (Goedel-Prover-V2-32B and Gemma 4) did not complete the pipeline. Our study suggests that machine-checked specifications in Lean can replace empirical evaluation in open-ended algorithm discovery. To the best of our knowledge, this is the first example of an automated pipeline that designs efficient black-box optimisation algorithms with machine-checked runtime bounds in Lean.

1 INTRODUCTION

From the perspective of theoretical computer science, algorithm design must produce two artefacts: First, the *algorithm* itself, a finite sequence of instructions that when executed in a computational model transforms within a finite number of steps any problem instance (the input) into an output (the solution) (Knuth, 1997). Secondly, the *analysis* of the algorithm, a mathematical proof that the algorithm produces the correct output on every problem instance, and a guarantee for the amount of computational resources required by the algorithm to produce that output (its complexity). In the context of machine learning, analysis also involves guaranteeing additional algorithmic properties, such as convergence rates, and regret and/or generalisation bounds. Inventing new and better algorithms involves an iterative developmental process between the two artefacts, a new candidate design drives the analysis in a certain direction, and obstacles in completion of the analysis can lead to insights into problem structure which motivate alterations to the algorithm design. The high cost and deep expertise needed for traditional algorithmic design have motivated research into automated design of heuristics. Recent LLM-based heuristic design approaches evaluate a candidate pool of heuristics on benchmark functions, and well-performing heuristics are varied through LLM-prompts to produce better ones. While this has been successful in many domains, the methodology is highly

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

empirical and does not lead to any guarantees on the correctness or performance of the heuristics produced.

It is a significant open problem whether AI agents can carry out algorithmic design as seen from theoretical computer science. It has been argued that automated theorem proving (ATP) is challenging for AI, because proofs are either correct or incorrect, not in-between. Furthermore, the space of proofs is infinitely large Buzzard (2024). Formal algorithmic design goes beyond ATP, because it requires both navigating a space of algorithmic designs, as well as automatically proving properties about these. Yet, the improvement in mathematical capability of frontier AI models over the last years has been remarkable. Long-standing conjectures have been closed using publicly available reasoning models with Lean-checked proof certificates (Sothanaphan, 2026). The time and cost needed to rigorously prove non-trivial mathematical statements have fallen significantly.

We take inspiration from the field of runtime analysis of evolutionary algorithms which has a well-defined approach to design and analysis. The field requires that algorithms fit in a black-box optimisation framework (Droste et al., 2006) (more often called the query model in TCS) where optimisation is formulated as an interaction between an oracle which knows the function to be optimised, and an algorithm which queries the oracle for fitness values. The field has developed a small number of highly generic mathematical tools that aid in the analysis. In particular, we build on drift analysis (Hajek, 1982; He & Yao, 2001) which measures the progress of the algorithm with a potential function. The potential function which is introduced for the purpose of the analysis and is not part of the algorithm itself, maps the state of the algorithm into a potential value. Bounds on the runtime can be obtained from properties about the distribution of the drift (change in potential per oracle query). The black-box model allows for an enormous variety of algorithms, most of which are useless. To constrain algorithm search, we use Lean guards to filter out provably flawed designs. We only allow algorithms that are able to visit all parts of the search space, that use randomness, and that take into account fitness responses from the oracle. The full staged pipeline is shown in Figure 1. The pipeline is seeded with a problem card that provides the definition of the problem, its problem instances, and basic properties. Additionally, the pipeline is seeded with an initial algorithmic design with a stated runtime bound. In all the experiments described here, we use uniform random search as algorithm with the exponential expected runtime $O(2^n)$. The staged pipeline consists of some informal stages where an informal model produces one or more prose-description of designs. There are also formal stages where the agent needs to synthesise a definition or a proof that is then verified by Lean. In case Lean reports a failure, the formal stage is repeated, until it backtracks (determined by a Luby sequence), or the formal stage passes. Some formal stages are gates to filter out flawed designs, others collect definitions that are used later, e.g., the definition of the algorithm or the potential. If the pipeline reaches the end of the pipeline, it obtains an algorithm A^{i+1} with a guaranteed improved runtime compared to algorithm A^i. The pipeline can be looped, e.g., producing an algorithm A^{i+2} which beats A^{i+1} etc. In the experiments described here, we only make one iteration of the loop.

Contributions To our knowledge, we propose the first automated system for formal design and analysis of black-box algorithms for pseudo-Boolean optimisation. The system does not involve any empirical evaluation or comparison of algorithms. The entire design process is driven by informal algorithm proposals followed by mathematical formalisation and analysis in Lean. Our contributions are 1) We formalise the human process of designing and analysing black-box optimisation algorithms through a staged pipeline which combines LLMs with Lean[1] 2) We demonstrate that an agent using the staged pipeline is able to design efficient black-box algorithms with associated machine-checked runtime bounds in Lean for a range of problems. On CYCLEUNCOVER, the constructed algorithm has expected polynomial runtime, whereas the classical (1+1) EA has exponential runtime. The agent is able to exploit structure in anonymised problem classes. 3) We show through ablation studies that without the staged pipeline, the agent fails to construct efficient algorithms.

Related work Research on automating the design and analysis of algorithms can be classified according to the level of automation in the design and the level of automation and rigour in the analysis (see Table 4 in Section G). *Firstly*, there are efforts into manually formalising with proof assistants such as Lean, Coq or Isabelle/HOL, existing (informal) mathematical analyses of algorithms. These contributions have the same strong benefits as formalising any other type of informal mathematics in

[1]Note to the reviewers: The pipeline uses a Lean drift library which is an independent contribution under review for a different conference.

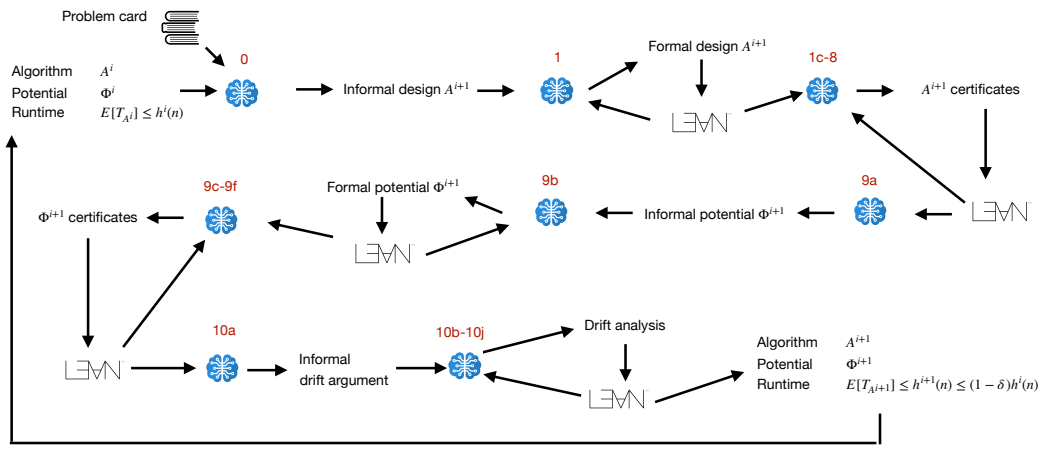


Figure 1: A simplified overview of the staged pipeline. The pipeline is seeded with a problem card, an algorithm A^i, and a proof of its running time. The pipeline produces a new algorithm A^{i+1} with improved runtime bound relative to algorithm A^i. Table 2 provides detailed descriptions of the pipeline stages shown in red.

Lean, including enabling machine checked proofs, curation of a digital library of mathematical results, and providing a foundation for machine-generated mathematics (Buzzard, 2024). Recent efforts in this direction include Li et al. (2024) who focused on first-order convex optimisation methods, with Lean proofs of convergence rates for some function classes. They considered deterministic algorithms, and the proofs therefore avoid technical challenges arising from stochasticity. In contrast, there are also Isabelle/HOL formalisations of complexity analysis of randomised tree data structures (Eberl et al., 2020). Kaminski et al. (2017) introduced a calculus for formally deriving the expected runtime of probabilistic programs, such as the coupon collector process. Also recent formalisation in machine learning consider stochastic effects (Vajjha et al., 2021). Schäffeler & Abdulaziz (2025) analysed convergence of value and policy iteration for finite state decision Markov decision processes. Some work also consider properties of neural networks, e.g., formalising a classic result about the expressive power of depth in neural networks (Cohen et al., 2016; Bentkamp et al., 2019). In contrast to our work, these work consider pre-existing algorithms and proofs, and proofs are human-designed. *Secondly*, the area of LLM-assisted automated theorem proving in Lean has gained significant momentum recently, with breakthrough results announced during Summer/Autumn 2026. There are recent examples of semi-automatic algorithm design and proof generation where humans with assistance from frontier models design new algorithms with Lean proofs. *Thirdly*, there is a large body of research where algorithms design is automated via LLMs, but where evaluation of candidate algorithms lacks machine-checked proofs and is entirely empirical (Liu et al., 2026). Systems such AlphaEvolve (Novikov et al., 2025), ShinkaEvolve (Lange et al., 2025), LLaMEA (Stein & Bäck, 2025), and EoH (Liu et al., 2024) formulate algorithm design as a search problem where candidate algorithms are evaluated empirically on a set of benchmark functions, and an LLM is used to generate algorithm variants. These systems do not provide any machine checked guarantees. Related approaches in highly restricted problem domains where correctness can be checked without formal proofs include AlphaTensor which optimise matrix multiplication algorithms via reinforcement learning (Fawzi et al., 2022).

Apart from automation of algorithmic design and analysis, there are other relevant works. An orthogonal task to algorithm design is problem formulation and modelling. Robbins et al. (2026) proposed FLARE to certify in Lean the correctness of mixed integer linear program (MILP) reformulations of combinatorial optimisation problems. Another related area is automated software verification. Here, the goal is to synthesise proofs (e.g., in Lean) that a piece of software satisfies a specification (pre-condition and post-condition) Li et al. (2026). More ambitiously, vericoding studies AI systems that take formal specifications as inputs and produce software along with proofs that the software satisfies the specification Bursuc et al. (2025).

2 RUNTIME ANALYSIS OF BLACK-BOX OPTIMISATION ALGORITHMS

To formally define a design space of optimisation algorithms, we will adopt the so-called *black-box scenario* from theory of discrete evolutionary algorithms (Droste et al., 2006), also called query (or decision tree) complexity in theoretical computer science (Buhrman & de Wolf, 2002). This scenario describes a broad class of algorithms by defining the goal of the algorithm, what information is available to an algorithm, what actions the algorithm can take, and how its runtime is measured. The algorithm is given a class of real-valued fitness functions F over a search space $\mathcal{X}$. An adversary picks a function $f \in F, f : \mathcal{X} \to \mathbb{R}$, and the goal of the algorithm is to find an optimal search point x^* such that $f(x) \leq f(x^*)$ for all x (i.e., maximisation of f). In each iteration t, the algorithm queries a search point x_t based on its previous history, and obtains from the oracle the function value $f(x_t)$. The search point x_t along with its fitness $f(x_t)$ is added to the history, and the process repeats until an optimum is found. The running time of the algorithm is defined as the number of times the algorithm queries the oracle until the algorithm queries an optimal search point for the first time. This definition is based on the assumption that the evaluation of the fitness function is the most time-consuming part of the algorithm. The entire interaction between the oracle and the optimisation algorithm is given in Algorithm 1. Within the black box scenario, the algorithm design is fully specified by a tuple $A = (\mathcal{Y}, \texttt{init}, \texttt{query}, \texttt{step})$, which will be described below. Executing an algorithm A involves executing Algorithm 1 with the given tuple A.

Algorithm 1 Black-Box Optimisation Algorithm

Require: Fitness function $f : \mathcal{X} \to \mathbb{R}$; random sequence $z_0, z_1, \ldots \in \mathcal{Z}$ with $z_t(i) \sim \mathrm{Unif}([0,1])$ i.i.d. for all $t, i \in \mathbb{N}$; algorithm $A = (\mathcal{Y}, \texttt{init}, \texttt{query}, \texttt{step})$ with $\texttt{init} : \mathcal{Z} \to \mathcal{Y}$, $\texttt{step} : \mathcal{Z} \times \mathcal{Y} \times \mathbb{R} \to \mathcal{Y}$ and $\texttt{query} : \mathcal{Y} \to \mathcal{X}$.

1: Initialise algorithm state $y_0 := \texttt{init}(z_0)$
2: **for** $t = 0$ to ∞ **do**
3: Select search point $x_t := \texttt{query}(y_t)$
4: Query oracle to obtain fitness $f_t := f(x_t)$
5: Compute the next state $y_{t+1} := \texttt{step}(z_{t+1}, y_t, f_t)$
6: **end for**

To allow the algorithm to make random decisions, the algorithm is provided with a two-way infinite random source $(z_t)_{t\in\mathbb{N}}$, where for each time step $t \in \mathbb{N}$ and index $i \in \mathbb{N}$, $z_t(i)$ is a uniformly distributed value in $[0, 1]$. We assume an underlying probability triple $(\Omega, \mathscr{G}, \Pr)$ and a filtration $(\mathscr{G}_t)_{t\in\mathbb{N}}$ to which $(z_t)_{t\in\mathbb{N}}$ is adapted. An algorithm A is associated with a state space $\mathcal{Y}$ where $y_t \in \mathcal{Y}$ corresponds to the memory accessible to the algorithm at iteration t. The initial state is obtained directly using the random source $\texttt{init}(z_0)$. In every iteration t, the algorithm uses the current state y_t to decide which search point x_t to query next. The algorithm transitions to the next state y_{t+1} by taking into account the next random value z_{t+1}, the previous state y_t, and the objective value $f(x_t)$ of the query corresponding to the current state.

Following (Droste et al., 2006), the running time of algorithm A on f can then be defined by $T_{A,f} = \min\{t \in \mathbb{N} \mid \forall x \in \mathcal{X}, f(x_t) \geq f(x)\}$. By construction $(x_t)_{t\in\mathbb{N}}$ and $(y_t)_{t\in\mathbb{N}}$ are adapted to $(\mathscr{G}_t)_{t\in\mathbb{N}}$, and $T_{A,f}$ is a stopping time. The worst-case expected running time of A over a class of functions F is then $T_{A,F} = \max_{f\in F} \mathbb{E}\left[T_{A,f}\right]$. All aspects of the above formalisation are indexed by an implicit problem instance size $n \in \mathbb{N}$. In particular, for each instance size n, we assume a corresponding search space $\mathcal{X}_n$, a fitness function $f_n : \mathcal{X}_n \to \mathbb{N}$, an algorithm A_n, and a worst-case expected running time T_{A_n,F_n}. We are primarily interested in the asymptotic growth of T_{A_n,F_n} as a function of n. Note that this formalisation guarantees that the algorithm is truly black box: it must be defined independently of the objective function f. Yet, due to the free choice of $\mathcal{Y}$ and `init`, `query`, `step`, the framework is highly general and can express most commonly used black-box optimisation algorithms, such as simulated annealing, population-based evolutionary algorithms, and estimation of distribution algorithms.

We now describe drift analysis (Hajek, 1982; He & Yao, 2001) which is the main techniques used to derive bounds on the runtime T_A, and which is incorporated into our pipeline. Most evolutionary algorithms are Markov processes over its state space $\mathcal{Y}$: the state $y_t \in \mathcal{Y}$ of the algorithm at iteration t completely determines the distribution of the state y_{t+1} in the next iteration. In principle, Markov chain theory could be applied to estimate the hitting time of the process from its state transition matrix.

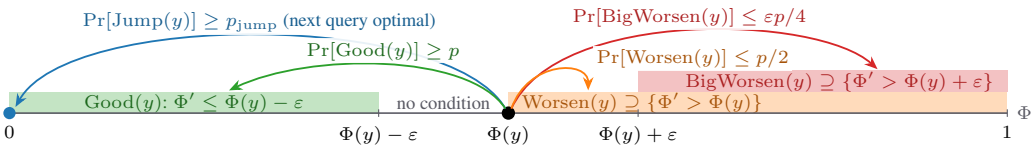


Figure 2: Drift conditions checked by the pipeline (Corollary 2.1, Stages 10f–10i) at a reachable, non-optimal state y. The sets of random draws $\mathrm{Good}(y)$, $\mathrm{Worsen}(y)$, $\mathrm{BigWorsen}(y)$ and $\mathrm{Jump}(y)$ are shown by where they send the next potential Φ'; the arrows give the required probability bounds, which together yield $\mathbb{E}[T_A] \leq 4/(\varepsilon p)$.

However, the state spaces induced by the algorithms are often so large that this approach becomes impractical. Instead, one can introduce a *potential function* $\Phi : \mathcal{Y} \to \mathbb{R}$ which given the current state of the algorithm measures how far the algorithm is from reaching its goal, i.e. $Z_t = \Phi(y_t)$. Note that the potential function is not part of the design of the algorithm, but is only used for the drift analysis. Designing a good potential function is often the most challenging part of the analysis of the algorithm. Given information about the distribution of the change in potential per time step, i.e., the *drift* $Z_{t+1} - Z_t$ of the process drift theorems provide statements about the long term properties of the process. Informally, the *additive drift theorem* (Theorem A.1) states that if the expected drift is at least ε, then the expected time to reach distance zero is Z_0/ε. The proof of the theorem (see (Corus et al., 2018)), uses standard arguments from martingale theory; the precise statement is Theorem A.1 in Section A.1. A Lean formalisation of the general additive drift theorem will be published in a separate manuscript.

Theorem A.1 requires evaluation of an expectation conditional on a σ-algebra. To not expose the AI agent to producing proofs incorporating measure-theory, we introduce the following simplified theorem which implies condition 1 in Theorem A.1. Proving drift condition 1 is now reduced to proving bounds on the probabilities of improving and worsening steps. To satisfy condition 2 in the additive drift theorem, we assume the potential is bounded in $[0, 1]$. Finally, to prove condition 3, i.e., integrability of the runtime, we require a lower bound of jumping directly to the target from every iteration. The set of conditions exposed in the pipeline are illustrated in Figure 2. Together, they yield the runtime bound $4/(\varepsilon p)$ used in the final stage of the pipeline. The proof of Corollary 2.1 is given in Section A.1.

Corollary 2.1. *Assume for all $t \in \mathbb{N}$: $X_t \in [0, 1]$, (1) $\mathbb{1}_{\{t<T\}} p \leq \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{\{X_{t+1} \leq X_t - \varepsilon\}} \mid \mathscr{G}_t\right]$, (2) $\mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{\{X_t < X_{t+1}\}} \mid \mathscr{G}_t\right] \leq \mathbb{1}_{\{t<T\}} p/2$, and (3) $\mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{\{X_t + \varepsilon < X_{t+1}\}} \mid \mathscr{G}_t\right] \leq \mathbb{1}_{\{t<T\}}\varepsilon p/4$. Then $\mathbb{E}\left[\mathbb{1}_{\{t<T\}}(X_{t+1} - X_t + \varepsilon p/4) \mid \mathscr{G}_t\right] \leq 0$.*

3 The Staged Pipeline

We now describe the motivation and details of the staged pipeline shown in Figure 1. The task of zero-shotting an algorithmic design along with its runtime analysis is challenging. We therefore break down this task into multiple stages according to how a human researcher in theory of evolutionary algorithms would solve the problem. Starting from an initial algorithmic design we want to improve, we make some informal design proposals. We describe the best algorithm formally in pseudo-code, then design a potential function which measures its progress, finally we apply a drift theorem. The most challenging aspects of this process is to design the algorithm and choose the potential function. The space of algorithmic designs allowed by our framework is vast. We therefore propose a staged pipeline where candidate designs are filtered through formal proof obligations via proof guards. The formal guards are designed to filter out flawed designs.

The implementation, with its two back-ends (CLI and vLLM) and the Kimina Lean server, and the structure of the stage prompts are described in Section B.

4 RESULTS

Table 1 summarises all runs; Figure 4 in Section F shows how each run progressed through the pipeline. Across the completed runs, the drift and runtime stages (10a–10j) take about half of the wall-clock time, and the incomplete frontier-model runs all stop at a drift stage, most often the mass bound of Stage 10h (Section F).

Theoretical benchmark functions Runtime analysis of new evolutionary algorithms is often initiated on simple theoretical benchmark functions which has a clear structure. Here, we consider the three most common functions, ONEMAX, LEADINGONES, and JUMPwhich are defined in Section C.

The ONEMAX value of a bitstring is the number of bits it differs from some hidden bitstring z. Standard evolutionary algorithms has expected runtime $O(n \log n)$, which can be seen via a multiplicative drift argument Doerr et al. (2012): The probability that a random bit-flip in the current bitstring improves the fitness is proportional to the number of bits it has in common with z. The optimal algorithm runs in $\Theta(n/logn)$ queries by querying the fitness of $\Theta(n/logn)$ uniformly sampled bitstring from which the hidden bitstring z, and the optimum, can be determined with high probability. One generation of the staged pipeline rediscovers a variant of the (1+1) EA with mutation rate $1/(n+1)$ rather than the standard $1/n$. The true expected runtime of this algorithm is $O(n \log n)$, however the agent obtains the weaker (but still mathematically correct) upper bound $O(n^3)$. As potential function, the agent uses essentially the normalised Hamming-distance to the optimum. The weak runtime bound is partly due to the pipeline which forces the agent to use Corollary 2.1 which does not allow for multiplicative drift.

The LEADINGONES value of a bitstring is the number of consecutive bits it has in common with some hidden bitstring z, when the bit positions are ordered by a hidden permutation σ. Typical evolutionary algorithms has runtime $O(n^2)$ which can be seen by an additive drift argument: a random bit-flip in a selected bitstring improves its fitness with probability $\Theta(1/n)$, and $\Theta(n)$ improvements are needed to reach the optimum. One generation of the staged pipeline discovers a variant of (1+1) EA, but where the mutation rate is chosen as $1/(f(x)+2)$ where $f(x)$ is the fitness of x. This fitness-adaptive mutation rate is nearly identical to the optimal adaptive mutation rate of $1/(f(x)+1)$ (Böttcher et al., 2010). For this algorithm, the agent obtains a sub-optimal runtime of $4(n+1)(n+2)^2 = O(n^3)$, while the true runtime for this algorithm is $(e/4)n^2(1+o(1))$ (Theorem 5 in (Böttcher et al., 2010)). While inspecting the drift analysis of the agent, we observed that the a missing factor n comes from the weak lower bound $1/n \leq (1-1/r)^{n-1}$. We therefore created another problem card which provided the inequality $1-1/r \geq \exp(-1/(r-1))$. The agent now produced a (1+1) EA-type algorithm with fixed mutation rate and runtime $4e(n+1)(n+2) = O(n^2)$.

The JUMP_k function takes the value $k+\text{ONEMAX}$, except for non-optimal search points at Hamming distance $d < k$ from the optimum which take the value $n - \text{ONEMAX}$. The problem therefore has a fitness gap (a local optimum) at distance k from the optimum. Typical evolutionary algorithms have runtime $O(n^k)$ because they need to flip k bits simultaneously to jump from the local optimum to the global optimum. By using more sophisticated diversity and variation operators, it is possible to reduce the runtime further. On generation of the staged pipeline discovered a variant of (1+1) EA with fixed mutation rate $1/(n+1)$. The agent proved a runtime bound of $O(n^{k+2})$ which is a factor of n^2 weaker than the true runtime for this algorithm which is $O(n^k)$. A factor n was lost due to using a weak lower bound $p^k(1-p)^n \geq p^{k+1}$ for $p = 1/(n+1)$. In a second set of experiments, we ran the pipeline for three generations and added the inequality $1-1/r \geq \exp(-1/(r-1))$ to the problem card. This produced an algorithm with machine-checked runtime $O(n^{k+1})$.

CYCLEUNCOVER The runtime analyses of evolutionary algorithms on ONEMAX, LEADINGONES, and JUMPare widely known. It is likely that the LLMs used in these studies to have seen many of these during their pre-training (although their Lean formalisations are as far as we know not available at the time of writing). To rule out the possibility that the agent simply restates Lean formalisations of well-known algorithms and analyses, we would like to test the agent on a new problem not previously discussed in the literature. We therefore designed CYCLEUNCOVER (defined in Figure 3 in Section D), which is a problem solvable in polynomial time, but not by a typical evolutionary algorithm (Theorem 4.1). The problem is essentially independent set on a cycle graph, but where infeasible solutions lead to a deceptive trap at large Hamming distance from the optimum. Given

the hidden permutation σ, a solution is simply to choose every second vertex along the permutation cycle.

Theorem 4.1. *The expected runtime of the (1+1) EA on* CYCLEUNCOVER *is* $e^{\Omega(n)}$.

The agent produced an algorithm for CYCLEUNCOVER with a machine-checked expected runtime of $O(n^2)$ (row 6 of Table 1). The algorithm works differently from an evolutionary algorithm: it cycles through $\Theta(n^2)$ pairs of indices to learn the edges in the graph, and thereby the hidden permutation σ. Once σ is known, it can sample the optimum. This result demonstrates that for a simple problem not previously studied in the literature (exactly as stated), the pipeline can recognise the problem structure and provide a tailored and efficient algorithm.

VERTEXCOVER We now turn to instances classes of the NP-hard minimum VERTEXCOVER problem. Given an undirected graph $G = (V, E)$, the minimum vertex cover problem yeasks for the smallest subset $V' \subseteq V$ of the vertices such that for each edge $\{u, v\} \in E$, at least one of the vertices u and v belong to V'. This problem is NP-hard, but a simple greedy algorithm returns a 2-approximate solution in polynomial time. We adopt the conventional pseudo-Boolean representation of VERTEXCOVER instances. Given a graph with n vertices $V = \{1, 2. \ldots, n\}$, we encode a subset of the vertices V' by a bitstring $x \in \{0, 1\}^n$, where $x_i = 1$ if and only if vertex $i \in V'$. We adapt the multi-objective formulation from (Kratsch & Neumann, 2013) into a single-objective fitness function which counts the number of chosen vertices (first term), and the number of uncovered edges (second term).

$$f(x) = -\sum_{i=1}^{n} x_i - (n+1) \sum_{\{i,j\} \in E} (1 - x_i)(1 - x_j). \tag{1}$$

Classical EAs applied naively to VERTEXCOVER perform poorly. The (1+1) EA and variants thereof need at least exponential time to achieve an approximation factor better than $(1 - \varepsilon)/\varepsilon$ for $n^{\delta-1} \leq \varepsilon \leq 2$, i.e., almost arbitrarily bad approximation ratio (Friedrich et al., 2010; Oliveto et al., 2008). In contrast, the trivial greedy algorithm obtains a 2-approximation in polynomial time. Using more sophisticated EAs and adding more problem-specific information can improve the performance. Kratsch & Neumann (2013) used the numer of selected vertices as first objective, and the number of uncovered edges as second objective. With this formulation, a multi-objective EA obtains an optimal vertex cover in expected time $O(OPT \cdot n^4 + n \cdot 2^{OPT^2+OPT})$, where OPT refers to the size of the smallest vertex cover. Using the ILP-relaxation as second objective leads to a lower runtime bound. Assuming only problem instances where $OPT \leq k$ for some fixed parameter k yields a polynomial runtime, i.e., the algorithm is a fixed-parameter algorithm. OPT cannot be bounded by a constant in general instances, and the runtime bound is therefore exponential on the entire problem class.

We would like to consider non-trivial combinatorial optimisation problems where the agent cannot rely on published results from the theoretical literature on black-box optimisation algorithms, or on formalisations in Mathlib. We also want to consider problems which we expect to be in P. We therefore chose the problem of computing a minimum vertex cover in well-covered graphs, which we call WELLCOVEREDVC. Recall that a vertex cover is *minimum* if the graph admits no vertex cover with fewer number of vertices. A vertex cover is called *minimal* (not to be confused with minimum) if no strict subsets $V' \subset V$ is a vertex cover. By definition, a minimum vertex cover is always a minimal vertex cover, but the converse is not true for all graphs. Graphs where the minimum and minimal vertex covers coincide are called *well-covered* graphs.

The fitness function in Eq. (1) combined with the choice of graph class gives the problem several structural properties that are helpful for search heuristics, and which are provided to the agent in the problem card. Given two search points with different number of uncovered edges, the one with fewer uncovered edges always has higher fitness (exposed as lemma `wc_F_lt_of_uncovered_lt` in the problem card). Also, if $\{i, j\} \in E$ is an uncovered edge in x, then then flipping bit x_i reduces the number of uncovered edges (exposed as lemma `wc_uncovered_flip_lt` in the problem card). It is also straightforward to see that f contains no local optima. Consider any non-optimal search point x. If x is a not a vertex cover, then there must exists an uncovered edge $\{i, j\}$, and a search point x' obtained by bit i or bit j in x has strictly higher fitness than x. Assume the other case, that x is a vertex cover but not a minimum vertex cover. Then since the graph is well-covered, x is also not a minimal vertex cover. Therefore, there exists a vertex i such that $x_i = 1$ and flipping bit i in x produces a strictly fitter vertex cover x'. The fitness therefore increases.

In a first set of experiments, we provided a problem card containing all the structural information described above. In this case, the agent produced an optimisation algorithm similar to a (1+1) EA, but with regime-changing asymmetric mutation rates. As long as the current search point is not a vertex cover (revealed by te fitness being less than $-n$), the mutation-rates were chosen assymetrically so that flipping 0-bits into 1-bits was twice as likely as the reverse. Once the current search point represented a vertex cover, the mutation-rate was chosen assymetrically in the opposite direction, making bit-flips from 1 into 0 twice as likely as bit-flips from 0 into 1. The overall runtime obtained was $O(n^4)$. The problem card in the first set of experiments revealed all the structural information needed to design an efficient algorithm. In a second set of experiments, we created an anonymised problem card which does not mention the terms vertex cover or well-covered graphs, only providing raw fitness definitions. An experiment with Fable 5.1 failed to obtain an algorithm in this setting. However, Opus 5.5 successfuly generated an (1+1) EA with regime-changing mutation rates. This time, the asymmetry between the rates was more pronounced, with the high mutation rate a factor $\Theta(n)$ larger than the small rate. In overall, the obtained runtime was $O(n^4)$.

The problem BANDWIDTHVC is VERTEXCOVER restricted to graphs with bandwidth at most k. A graph $G = (V, E)$ with vertex set $V = \{0, 1, \dots n-1\}$ is said to have *bandwidth* k if for all edges $\{u, v\} \in E$, it holds $|u - v| \leq k$. A low bandwidth k implies that edges are only allowed between "nearby" vertices with respect to their numbering. In particular, $k = 1$ allows only subgraphs of the path $0, 1, \dots, n-1$, while $k = n-1$ allows every graph.

Since this is an NP-hard problem, we did not expect the agent to be able to produce a good algorithm for general vertex cover. Instead, the agent came up with an algorithm that probes all the $\Theta(n^2)$ potential edges in the graph to learn the problem instance, then without making further queries, computes offline the optimum for that instance, and finally query the optimum. The overall runtime bound was $O(n^3)$. This demonstrates that the agent is able to "cheat" this evaluation framework by exploting a well-known weakness in how algorithms are evaluated in the black-box scenario: Algorithms are only charged for making fitness queries, and not for offline computation. As the agent shows, the black-box complexity of NP-hard problems is polynomial.

Sub-modular minimisation We now turn to the problem of minimising sub-modular functions (SFM), a problem with important applications in machine learning. We represent any subset $S \subseteq [n]$ as a bitstring $x \in \{0,1\}^n$ in the natural way by setting $x_i = 1$ if and only if $i \in S$. We now call a pseudo-Boolean function $f : \{0,1\}^n \to \mathbb{R}$ submodular if and only if for all subsets $S \subseteq T \subseteq [n]$ and $v \notin T$, it holds $f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$. We will assume that f is integer-valued with range in the interval $[0, B]$ where $B := n^c$ for some unspecified constant c. There exists theoretical algorithms that minimise SFMs in $O(n^3 \log\log(n)/\log(n))$ oracle queries (Jiang, 2023), i.e., independent of B. However, the more practically useful Fujishige-Wolfe algorithm has running time $O(n^2B^2)$. We are not aware of runtime analyses of evolutionary algorithms for general SFM minimisation, but results for special cases such as the minimum spanning tree problem are available (Neumann & Wegener, 2007).

We expected this problem to be significantly more challenging to the previous problems. We therefore added to the problem card the advise that this is a challenging problem where a naive elitist hill-climber may not work, and where the agent needs to carefully design the potential function and exploit submodularity. Furthermore, we ran several generations of the pipeline. The algorithmic design and analysis produced in a first generation of the pipeline was seeded into the following generation, etc. Through all generations, the agent produced the same class of algorithm, which can be described as projected subgradient descent on the Lovasz extension. The first generation returned a runtime bound of $12nB^2(2n+4)(n+1) = O(n^{3+2c})$. The second generation improved the constant factor, reducing the runtime to $16n^2B^2(n+2)$. The third generation saw a linear-time speed-up, with a runtime of $32nB^2(n+1) = O(n^2B^2)$. Note that this is asymptotically the same as the Fujishige-Wolfe algorithm. Generation four and five only brought constant time improvements, leading to the approximate upper bound $8.13 \cdot B^2(n+1)^2$.

Ablation experiments To evaluate the impact of the staged pipeline, we developed an unstaged pipeline consisting of a single stage: Given a problem card, the agent is prompted to construct an algorithm for the problem which type-checks with the black-box framework, and which has better runtime than the seeded algorithm. The agent is not given the drift analysis API, the measurability toolkit, the proof cookbook, and most random stream properties. In order to be able to verify the

algorithmic construction in Lean, we provided the agent with Lean-definitions of the black-box optimisation algorithm framework, and definitions of runtime. In order for the algorithm to be able to use the random source, we also provided basic clauses related to sampling from the random stream. Since the ablation only concerned the staged pipeline, we kept the problem cards unchanged. We also provided the agent with the definition of the seeded algorithm (random search).

For CYCLEUNCOVER, Fable 5.1 reproduced the trivial random search algorithm (i.e., it copies the provided `algorithm0` with minor syntactic changes). Note that this failure mode would have been caught by the `NotRandomSearch` guard in the staged pipeline. Instead of relying on drift analysis, the agent applied the tail-sum formula $\mathbb{E}[T] = \sum_{k=1}^{\infty} \Pr(k \leq T)$ to the runtime variable T. The agent recognised that the problem involved a hidden permutation and two global optima, giving a runtime bound of $2^n/2$.

For BANDWIDTHVC, Fable 5.1 constructed the (1+1) EA, and proved all formalisation properties. However, the model failed to produce a runtime bound. The model attempted to apply a fitness-level argument. This is one of the simplest to use techniques for analysing the runtime of the (1+1) EA introduced more than two decades ago (Wegener, 2002). The method is perhaps the simplest to use tool, but has limited applicability for complex algorithms and fitness landscapes.

We then investigated weaker models. Claude Sonnet (row 17) passed Stages 0–10g on CYCLEUNCOVER, but needed between 54 and about 1,000 attempts per stage, and it did not pass the mass bound of Stage 10h, because its Good set is empty at some reachable states (Section E.9). The open-weight models, Goedel-Prover-V2-32B for the formal stages and Gemma 4 26B-A4B for the prose stages (vLLM backend, 256 samples per attempt), did not complete the pipeline on ONEMAX in any of 14 recent runs (Section E.1). Rows 19–21 of Table 1 are representative. Two runs stop at `NotRandomSearch` (Stage 4) after 13 and 24 batches of 256 samples, although the statement is true for their PBIL-like designs. The third passes Stage 4 only through a loophole of an earlier pipeline version, since its algorithm is random search, and then stops at `PotentialFineGrained` (Stage 9d), again on a true statement. Frontier models stop, if at all, at a drift stage, whereas the open-weight models stop much earlier (Section F), and they generate 25–124M output tokens per run, about two orders of magnitude more than the frontier-model runs.

5 CONCLUSION

We have presented a staged pipeline in which an agent designs black-box optimisation algorithms and proves their expected runtime in Lean, with no empirical evaluation. Across 8 problem classes, 12 of 15 runs with frontier models completed the pipeline. The agent designed an $O(n^2)$ algorithm for CYCLEUNCOVER, on which the (1+1) EA needs $e^{\Omega(n)}$, matched the optimal order $O(n^2)$ on LEADINGONES, and rediscovered projected subgradient descent on the Lovász extension for integer-valued submodular minimisation, improving its own bound to $O(n^{2+2c})$ over five generations. Without the staging, or with open-weight models, the pipeline did not produce efficient algorithms. The bounds are not yet state of the art; richer drift theorems, more iterations of the design loop, and continuous search spaces are natural next steps.

Only frontier models complete the pipeline, and because runs are costly, most problems were attempted in a single run. The agent can use only the simplified drift theorem (Corollary 2.1), which rules out multiplicative or variable drift and explains weak bounds such as $O(n^3)$ on ONEMAX; the search space is restricted to bitstrings.

Ethics Statement

This work studies the automated design and formal analysis of optimisation algorithms. It involves no human subjects, personal data or sensitive data sets, and all problem instances are synthetic. The machine-checked proofs make claims about the synthesised algorithms verifiable, which reduces the risk of relying on incorrect AI-generated results; such guarantees hold only relative to the formal specification, which is written by humans and must be checked for correctness. We are not aware of other ethical concerns, and we follow the ICLR Code of Ethics.

Reproducibility Statement

The black-box framework and the runtime definition are given in Algorithm 1, the drift theorems used by the pipeline in Section 2 with proofs in Section A.1, and the formal statement of every pipeline stage in Table 2. Table 1 lists all runs, including the failed ones, and Section E gives the problem cards and, for each run, the synthesised algorithm, potential, drift constants and any deviation from the standard protocol (pre-checked stages, rejected probes, withheld lemmas read). The anonymised supplementary material contains, for every run in Table 1, the final Lean file produced by the pipeline, together with the problem cards, the staged and zero-shot pipeline files, and the Lean library they build on, with the Lean toolchain and Mathlib revision pinned. A README explains how to re-check each run with the Kimina Lean server (Dos Santos et al., 2025) and lists the axioms on which each final theorem depends. The prompt logs, per-stage checkpoints and session transcripts of all runs will be released upon acceptance.

AI Use Statement

This paper addresses the scientific question of whether large language models can generate algorithms together with proofs of their efficiency, so LLMs were used extensively as the object of study. All synthesised algorithms, potential functions, drift certificates and their Lean proofs reported in Table 1 were generated by the models named there (Claude Opus 5, Opus 5.5, Fable 5.1 and Sonnet; Goedel-Prover-V2-32B and Gemma 4), and every proof was checked by Lean. In addition, LLMs were used to help derive the combinatorial drift part of the Lean library used by the pipeline (Section 2), whose results are likewise checked by Lean; to collect pseudo-code, potentials and constants from the generated runs for Section E, which we checked against the Lean artefacts; and to polish the writing. The authors take full responsibility for the content of the paper.

References

Alexander Bentkamp, Jasmin Christian Blanchette, and Dietrich Klakow. A Formal Proof of the Expressiveness of Deep Learning. *Journal of Automated Reasoning*, 63(2):347–368, August 2019. ISSN 0168-7433, 1573-0670. doi: 10.1007/s10817-018-9481-5. URL http://link.springer.com/10.1007/s10817-018-9481-5.

Süntje Böttcher, Benjamin Doerr, and Frank Neumann. Optimal fixed and adaptive mutation probabilities for the LeadingOnes problem. In *Parallel Problem Solving from Nature – PPSN XI*, volume 6238 of *Lecture Notes in Computer Science*, pp. 1–10. Springer, 2010.

Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science*, 288(1):21–43, October 2002. ISSN 0304-3975. doi: 10.1016/S0304-3975(01)00144-X. URL https://www.sciencedirect.com/science/article/pii/S030439750100144X.

Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A benchmark for vericoding: formally verified program synthesis, September 2025. URL https://arxiv.org/abs/2509.22908v1.

Kevin Buzzard. Mathematical reasoning and the computer. *Bulletin of the American Mathematical Society*, 61(2):211–224, February 2024. ISSN 1088-9485, 0273-0979. doi: 10.1090/bull/1833. URL https://www.ams.org/bull/2024-61-02/S0273-0979-2024-01833-3/.

Süntje Böttcher, Benjamin Doerr, and Frank Neumann. Optimal Fixed and Adaptive Mutation Rates for the LeadingOnes Problem. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej, and Günter Rudolph (eds.), *Parallel Problem Solving from Nature, PPSN XI*, pp. 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-15843-8 978-3-642-15844-5. doi: 10.1007/978-3-642-15844-5_1. URL http://link.springer.com/10.1007/978-3-642-15844-5_1.

Nadav Cohen, Cs Huji Ac Il, Or Sharir, Or Sharir, Cs Huji Ac Il, Amnon Shashua, and Cs Huji Ac Il. On the Expressive Power of Deep Learning: A Tensor Analysis. *Journal of Machine Learning Research*, 49:1–31, 2016.

Dogan Corus, Duc-Cuong Dang, Anton V. Eremeev, and Per Kristian Lehre. Level-Based Analysis of Genetic Algorithms and Other Search Processes. *IEEE Transactions on Evolutionary Computation*, 22(5):707–719, October 2018. ISSN 1089-778X, 1089-778X, 1941-0026. doi: 10.1109/TEVC.2017.2753538. URL https://ieeexplore.ieee.org/document/8039236/.

Benjamin Doerr. Probabilistic tools for the analysis of randomized optimization heuristics. In Benjamin Doerr and Frank Neumann (eds.), *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, pp. 1–87. Springer, 2020.

Benjamin Doerr, Daniel Johannsen, and Carola Winzen. Multiplicative Drift Analysis. *Algorithmica*, 64(4):673–697, December 2012. ISSN 1432-0541. doi: 10.1007/s00453-012-9622-x. URL https://doi.org/10.1007/s00453-012-9622-x.

Marco Dos Santos, Hugues de Saxcé, Haiming Wang, Ran Wang, Mantas Baksys, Mert Unsal, Junqi Liu, Zhengying Liu, and Jia Li. Kimina Lean Server: A high-performance Lean server for large-scale verification. arXiv:2504.21230, 2025.

Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1–2):51–81, 2002.

Stefan Droste, Thomas Jansen, and Ingo Wegener. Upper and Lower Bounds for Randomized Search Heuristics in Black-Box Optimization. *Theory of Computing Systems*, 39(4):525–544, July 2006. ISSN 1433-0490. doi: 10.1007/s00224-004-1177-z. URL https://doi.org/10.1007/s00224-004-1177-z.

Manuel Eberl, Max W. Haslbeck, and Tobias Nipkow. Verified Analysis of Random Binary Tree Structures. *Journal of Automated Reasoning*, 64(5):879–910, June 2020. ISSN 1573-0670. doi: 10.1007/s10817-020-09545-0. URL https://doi.org/10.1007/s10817-020-09545-0.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, October 2022. ISSN 1476-4687. doi: 10.1038/s41586-022-05172-4. URL https://www.nature.com/articles/s41586-022-05172-4.

Tobias Friedrich, Jun He, Nils Hebbinghaus, Frank Neumann, and Carsten Witt. Approximating Covering Problems by Randomized Search Heuristics Using Multi-Objective Models. *Evolutionary Computation*, 18(4):617–633, December 2010. ISSN 1063-6560. doi: 10.1162/EVCO_a_00003. URL https://doi.org/10.1162/EVCO_a_00003.

Bruce Hajek. Hitting-Time and Occupation-Time Bounds Implied by Drift Analysis with Applications. *Advances in Applied Probability*, 14(3):502–525, 1982. ISSN 0001-8678. doi: 10.2307/1426671. URL https://www.jstor.org/stable/1426671. Publisher: Applied Probability Trust.

Jun He and Xin Yao. Drift analysis and average time complexity of evolutionary algorithms. *Artificial Intelligence*, 127(1):57–85, March 2001. ISSN 00043702. doi: 10.1016/S0004-3702(01)00058-3. URL http://linkinghub.elsevier.com/retrieve/pii/S0004370201000583.

Haotian Jiang. Minimizing Convex Functions with Rational Minimizers. *Journal of the ACM*, 70 (1):1–27, February 2023. ISSN 0004-5411, 1557-735X. doi: 10.1145/3566050. URL https://dl.acm.org/doi/10.1145/3566050.

Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs, November 2017. URL http://arxiv.org/abs/1601.01001. arXiv:1601.01001 [cs.LO].

Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1997. ISBN 0201896834.

Stefan Kratsch and Frank Neumann. Fixed-Parameter Evolutionary Algorithms and the Vertex Cover Problem. *Algorithmica*, 65(4):754–771, April 2013. ISSN 1432-0541. doi: 10.1007/s00453-012-9660-4. URL https://doi.org/10.1007/s00453-012-9660-4.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, pp. 611–626, 2023.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution, September 2025. URL http://arxiv.org/abs/2509.19349. arXiv:2509.19349 [cs].

Chenyi Li, Ziyu Wang, Wanyi He, Yuxuan Wu, Shengyang Xu, and Zaiwen Wen. Formalization of Complexity Analysis of the First-order Algorithms for Convex Optimization, July 2024. URL http://arxiv.org/abs/2403.11437. arXiv:2403.11437 [math].

Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-Code-Prover: Hierarchical Proof Search for Open State-of-the-Art Code Verification, March 2026. URL http://arxiv.org/abs/2603.19329. arXiv:2603.19329 [cs.SE].

Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. arXiv:2508.03613, 2025.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 32201–32223. PMLR, July 2024. URL https://proceedings.mlr.press/v235/liu24bs.html.

Fei Liu, Yiming Yao, Ping Guo, Zhiyuan Yang, Xi Lin, Zhe Zhao, Xialiang Tong, Kun Mao, Zhichao Lu, Zhenkun Wang, Mingxuan Yuan, and Qingfu Zhang. A Systematic Survey on Large Language Models for Algorithm Design. *ACM Computing Surveys*, 58(8):1–32, June 2026. ISSN 0360-0300, 1557-7341. doi: 10.1145/3787585. URL https://dl.acm.org/doi/10.1145/3787585.

Frank Neumann and Ingo Wegener. Randomized local search, evolutionary algorithms, and the minimum spanning tree problem. *Theoretical Computer Science*, 378(1):32–40, June 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.11.002. URL https://www.sciencedirect.com/science/article/pii/S0304397506008589.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, June 2025. URL http://arxiv.org/abs/2506.13131. arXiv:2506.13131 [cs].

Pietro S. Oliveto, Jun He, and Xin Yao. Analysis of population-based evolutionary algorithms for the vertex cover problem. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pp. 1563–1570, June 2008. doi: 10.1109/CEC.2008.4631000. URL https://ieeexplore.ieee.org/abstract/document/4631000.

Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and Lean Collaboration for Advanced Formal Reasoning, May 2025. URL https://arxiv.org/abs/2505.05758v5.

Henry Robbins, Connor Lawless, Madeleine Udell, and Ellen Vitercik. FLARE: Verifying MILP Reformulations with LLM-Based Theorem Proving, August 2026. URL http://arxiv.org/abs/2608.25220. arXiv:2608.25220 [cs.AI].

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, January 2024. doi: 10.1038/s41586-023-06924-6.

Maximilian Schäffeler and Mohammad Abdulaziz. Formally Verified Approximate Policy Iteration. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(25):26659–26667, April 2025. ISSN 2374-3468. doi: 10.1609/aaai.v39i25.34868. URL https://ojs.aaai.org/index.php/AAAI/article/view/34868. Number: 25.

Nat Sothanaphan. Resolution of Erdős Problem #728: a writeup of Aristotle's Lean proof, January 2026. URL http://arxiv.org/abs/2601.07421. arXiv:2601.07421 [math.NT].

Niki van Stein and Thomas Bäck. LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29 (2):331–345, April 2025. ISSN 1941-0026. doi: 10.1109/TEVC.2024.3497793. URL https://ieeexplore.ieee.org/abstract/document/10752628.

Koundinya Vajjha, Avraham Shinnar, Barry Trager, Vasily Pestun, and Nathan Fulton. CertRL: formalizing convergence proofs for value and policy iteration in Coq. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pp. 18–31, New York, NY, USA, January 2021. Association for Computing Machinery. ISBN 978-1-4503-8299-1. doi: 10.1145/3437992.3439927. URL https://dl.acm.org/doi/10.1145/3437992.3439927.

Sumanth Varambally, Thomas Voice, Yanchao Sun, Zhifeng Chen, Rose Yu, and Ke Ye. Hilbert: Recursively Building Formal Proofs with Informal Reasoning, September 2025. URL https://arxiv.org/abs/2509.22819v2.

Ingo Wegener. Methods for the Analysis of Evolutionary Algorithms on Pseudo-Boolean Functions. In Ruhul Sarker, Masoud Mohammadian, and Xin Yao (eds.), *Evolutionary Optimization*, pp. 349–369. Springer US, Boston, MA, 2002. ISBN 978-0-306-48041-6. doi: 10.1007/0-306-48041-7_14. URL https://doi.org/10.1007/0-306-48041-7_14.

A Detailed Description of Pipeline

We now describe the stages in the pipeline informally. The stages in the pipeline has informal stages where the agent proposes in English an informal design (either of an algorithm or potential function), and formal stages that are either definitions or formal guards. Formal statements are available in the appendix. `StepMeasurability` (1c): We require that the step function is a measurable function. This condition will be required for the drift analysis later in the pipeline. `Completeness1` (2): We require that the algorithm must be able to query any search point in every iteration. More precisely, for every search point $x \in \mathcal{X}$, and for every reachable state $y \in \mathcal{Y}$, there exists a random input $z \in \mathcal{Z}$ and fitness value $f \in \mathbb{R}$, such that if the algorithm receives fitness f for the query associated with state y, it will transition on random input z to a state where x is queried. This condition is satisfied by almost all evolutionary algorithms, e.g., those that involve bit-wise mutation. Note that the completeness condition can be trivially satisfied for an arbitrary algorithm by querying a uniform search point with some non-zero (but perhaps very small probability) in every step. The completeness condition is not always necessary for an efficient algorithmic design. However, the condition makes it straightforward to prove that the algorithm eventually queries the optimal search point (i.e., the algorithm is correct), and that for finite search spaces, the algorithm has a finite expected hitting time which is required by our formulation of the drift theorem. The extra assumption that y is a reachable state and not an arbitrary state was included to avoid imposing the condition on inconsistent states that the algorithm never visits. For example, the state of the UMDA algorithm is a probability vector $p \in [0, 1]^n$ where $p_i, i \in [n]$, is the probability of sampling a 1-bit in position i. The algorithm maintains the invariant $p_i \in [1/n, 1 - 1/n]$ in order not to get stuck in a particular bit-position. `NonAbsorbing` (3): We require that the algorithm uses the random input. More precisely, for every reachable state $y \in \mathcal{Y}$, and any oracle response $f \in \mathbb{R}$, there exists two random inputs $z_1, z_2 \in \mathcal{Z}$ such that the algorithm in the next iteration makes different queries on random input z_1 and z_2. This condition is not necessary to have an efficient algorithm. However, the condition filters out many poor algorithms, such as those that only return a constant value. `NotRandomSearch` (4): We require that the algorithm can reach a state where the query distribution is different from the uniform distribution (random search). Random search has expected running time $2^{\Omega(n)}$, and any efficient algorithm must have a different query distribution than random search. We observed that weaker models applied to the pipeline with random search seeded as algorithm$_0$ tended to produce designs that were essentially random search. In an early version of the pipeline, we did not include the reachable state assumption in the condition. We discovered that this would allow the model to cheat by devising a state space with unreachable witness states $\mathcal{Y}$ which could serve as witnesses for the `NotRandomSearch` and other properties. `Adaptiveness` (5): We require that the state update of the algorithm depends on the fitness responses from the oracle. Clearly, no algorithm which completely ignores the oracle can be an effective optimiser on even simple problem classes such as OneMax. `QueryNonInjectivity` (6): We require that there exists two different states which lead to different queries. This condition is added to filter out degenerate algorithms that always query the same search point. The remaining algorithm construction stages (7–8) are designed to package all the definitions into a Lean structure. This will allow the later proof infrastructure to process the algorithmic design as one mathematical object. These stages also includes measurability requirements, e.g., that all the components of the algorithm are measurable functions. The next main part is the drift analysis. We would like the agent to prove that the three conditions of Corollary 2.1 are satisfied, where X_t corresponds to the potential of the algorithm at iteration t. The pipeline asks the agent to, for every state y, define three subsets of random inputs $\mathrm{Good}(y), \mathrm{Worsen}(y), \mathrm{BigWorsen}(y)$ corresponding to the three conditions respectively, drift parameters ε and p, and a jump to lower bound p_{jump}. `PotentialDesign` and `DefinePotential` (9a–9b): In the informal stage, the agent produces several candidate potential functions along with the parameters above along with an informal "drift story" explaining why the potential function applied to the algorithm has drift towards the optimum. The selected design is then formalised. We require that value of the potential is always in the interval $[0, 1]$. Although not require for drift analysis, this requirement which is not difficult to satisfy through normalisation simplifies the formal drift analysis machinery later in the pipeline. `PotentialOpt` (9c): We require that the potential of a state y is 0 if and only if the algorithm queries an optimal search point in state y. This is a necessary condition for the drift theorem to give a meaningful statement about the runtime of the algorithm. `PotentialFineGrained` (9d): The seeded design uses a potential function with only the values 0 and 1, where 0 corresponds to optimal states and 1 all other states. This is sufficient to prove runtime bound for random search, however it will be insufficient to capture the gradual progress of an efficient algorithm. Experiments with weaker models often reproduced the

two-valued potential function. We therefore added the formal guard that the potential function must contain a value between 0 and 1. `PotentialMeasurability` (9e): We require that the potential function is measurable. This property is needed for the later drift analysis. The remaining guards in this part of the pipeline involve proving measurability properties of the potential function, and packaging into a Lean structure for easy handling later (`ConstructPotentialStructure`, 9f). `PositiveDrift` (10f): We require that for any reachable, non-optimal state y, if the algorithm receives a random input $z \in \mathrm{Good}(y)$, then the potential decreases by at least ε in the next iteration. `NegativeDriftAndJump` (10g): We require that for any non-optimal reachable state y and any random input $z \in \mathcal{Z}$, if the potential increases in the next iteration, then z belongs to $\mathrm{Worsen}(y)$, and if it increases by more than ε, then z belongs to $\mathrm{BigWorsen}(y)$. Furthermore, for any reachable state y, if the algorithm receives input $z \in \mathrm{Jump}(y)$, then the potential in the next iteration is 0. In other words, the algorithm jump directly to an optimal search point whenever it receives a random input in $\mathrm{Jump}(y)$. `DriftSetMass` (10h): We require that, for every reachable, non-optimal state y, the probabilities that the random input lies in $\mathrm{Good}(y)$, $\mathrm{Worsen}(y)$ and $\mathrm{BigWorsen}(y)$ satisfy conditions (1)–(3) of Corollary 2.1 (Figure 2). `JumpMass` (10i): We require that from all states y, $p_{\text{jump}} \leq \Pr(z \in \mathrm{Jump}(y))$. When satisfied, this condition implies that the runtime of the algorithm is stochastically dominated by a geometrically distributed random variable with success probability p_{jump}. The expected runtime is therefore finite, which implies one of the conditions of the additive drift theorem. The only requirement on p_{jump} is that it is positive. `RuntimeBound` (10j): The final stage is an application of a runtime bound $\mathbb{E}[T_A] \leq 4/(\varepsilon p)$ that follows immediately from the previously collected proof certificates.

Table 2: The stages of the pipeline that generates one algorithm $A^{i+1} = (\mathcal{Y}, \texttt{init}, \texttt{step}, \texttt{query})$ and proves it faster than its predecessor, restated in the $(\mathcal{X}, \mathcal{Y}, \mathcal{Z}, \mathbb{R})$ formalism of Algorithm 1. `[I]` is an informal (prose) stage, `[F⋆]` a formal stage with candidate selection, `[F−]` a formal stage without selection. $f \in \mathcal{F}$ denotes a fixed fitness function (an element of the problem class, e.g. ONEMAX composed with an unknown target bitstring), so that $\texttt{reachableWith}(f, y)$ (as opposed to $\texttt{reachable}(y)$) restricts to states reachable when the *true* fitness fed to `step` is always f. $z, z_1, z_2 \in \mathcal{Z}$ denote random draws; $\mathrm{unif}(z) \in \mathcal{X}$ is the bitstring obtained by sampling each coordinate of z against threshold $1/2$. n is the bitstring length, and a 'prev' subscript refers to the previous algorithm/potential pair in the outer self-improvement loop.

Stage	Name	Formalisation
Algorithm construction (Stages 0–8)		
0	`DesignLock [I]`	Commit to $\mathcal{Y}$, `init`, `step`, `query` and a qualitative account of which component of $y \in \mathcal{Y}$ tracks f; no numeric constants yet.
1	`Construct-AlgorithmCom-ponents [F⋆]`	Define $\mathcal{Y}$, $\texttt{init} : \mathcal{Z} \to \mathcal{Y}$, $\texttt{step} : \mathcal{Z} \times \mathcal{Y} \times \mathbb{R} \to \mathcal{Y}$, $\texttt{query} : \mathcal{Y} \to \mathcal{X}$.
1c	`StepMeasur-ability [F−]`	`step` is measurable.
2	`Completeness1 [F−]`	$\forall x \in \mathcal{X}, \forall y \in \mathcal{Y}$, $\texttt{reachable}(y) \to \exists z \in \mathcal{Z}, \exists f \in \mathbb{R}$, $\texttt{query}(\texttt{step}(z, y, f)) = x$.
3	`NonAbsorbing [F−]`	$\forall y \in \mathcal{Y}$, $\texttt{reachable}(y) \to \forall f \in \mathbb{R}$, $\exists z_1, z_2 \in \mathcal{Z}$, $\texttt{query}(\texttt{step}(z_1, y, f)) \neq \texttt{query}(\texttt{step}(z_2, y, f))$.
4	`NotRandom-Search [F−]`	$\exists y \in \mathcal{Y}$, $\texttt{reachable}(y) \wedge \exists z \in \mathcal{Z}, \exists f \in \mathbb{R}$, $\texttt{query}(\texttt{step}(z, y, f)) \neq \mathrm{unif}(z)$.
5	`Adaptiveness [F−]`	$\exists y \in \mathcal{Y}$, $\texttt{reachable}(y) \wedge \exists z, z' \in \mathcal{Z}, \exists f_1, f_2, f \in \mathbb{R}$, $\texttt{query}(\texttt{step}(z', \texttt{step}(z, y, f_1), f)) \neq \texttt{query}(\texttt{step}(z', \texttt{step}(z, y, f_2), f))$ (a two-step strengthening of the usual one-step adaptiveness property: the fitness fed at the *first* step must still move the query one step later).
6	`QueryNon-Injectivity [F−]`	$\exists y_1, y_2 \in \mathcal{Y},\ y_1 \neq y_2 \wedge \texttt{query}(y_1) = \texttt{query}(y_2)$.
7	`AlgorithmInfra-structure [F−]`	Equip $\mathcal{Y}$ with the five instances (measurable space, topology, pseudo-metrizability, Borel σ-algebra, second-countability) the framework requires.
7a	`InitMeasur-ability [F−]`	`init` is measurable.

(Table 2 continued)

Stage	Name	Formalisation
7b	QueryMeasurability [F-]	`query` is measurable.
8	ConstructAlgorithmStructure [F⋆]	Package $A = (\mathcal{Y}, \texttt{init}, \texttt{step}, \texttt{query})$, together with the Stage 1c/7/7a/7b witnesses, into a single algorithm record.
Potential design (Stages 9a–9f)		
9a	PotentialDesign [I]	Rank candidate potentials $\Phi : \mathcal{F} \times \mathcal{Y} \to [0,1]$ by whether `step` can be shown, even informally, to decrease Φ faster than unif.
9b	DefinePotential [F⋆]	Define $\Phi : \mathcal{F} \times \mathcal{Y} \to \mathbb{R}$ with $\forall f \in \mathcal{F}, \forall y \in \mathcal{Y},\ 0 \le \Phi(f,y) \le 1$.
9c	PotentialOpt [F-]	$\forall f \in \mathcal{F}, \forall y \in \mathcal{Y},\ \Phi(f,y) \le 0 \iff \forall x \in \mathcal{X},\ f(x) \le f(\texttt{query}(y))$ — $\Phi(f,\cdot)$ vanishes exactly at states whose query is f-optimal.
9d	PotentialFineGrained [F-]	$\exists f \in \mathcal{F}, \exists y \in \mathcal{Y},\ 0 < \Phi(f,y) < 1$ — rules out a $\{0,1\}$-valued (indicator) potential.
9e	PotentialMeasurability [F-]	$\forall f \in \mathcal{F},\ \Phi(f,\cdot)$ is measurable.
9f	ConstructPotentialStructure [F-]	Package Φ, together with the Stage 9b–9e witnesses, into a single potential record.
Drift design and runtime bound (Stages 10a–10j)		
10a	DriftDesign [I]	Work out closed forms, against the now-fixed A and Φ, for $\varepsilon, p, p_{\text{jump}} : \mathbb{N} \to \mathbb{R}_{>0}$ and for the four draw sets $\text{Good}, \text{Worsen}, \text{BigWorsen}, \text{Jump} : \mathcal{F} \times \mathcal{Y} \to 2^{\mathcal{Z}}$ below.
10b	DefineDriftConstants [F⋆]	Define $\varepsilon(n), p(n), p_{\text{jump}}(n)$ as closed-form expressions in n.
10c	DriftConstantProperties [F-]	$\varepsilon(n) > 0,\ p(n) > 0,\ p_{\text{jump}}(n) \in (0,1)$, and (the speed-up certificate) $\exists \delta > 0, \exists n_0, \forall n > n_0,$ $(1+\delta)\,\varepsilon_{\text{prev}}(n)\,p_{\text{prev}}(n) < \varepsilon(n)\,p(n)$.
10d	DefineDriftSets [F⋆]	Define $\text{Good}, \text{Worsen}, \text{BigWorsen}, \text{Jump} : \mathcal{F} \times \mathcal{Y} \to 2^{\mathcal{Z}}$ as explicit sets of draws.
10e	DriftSetMeasurability [F-]	For each $S \in \{\text{Good}, \text{Worsen}, \text{BigWorsen}, \text{Jump}\}$: $\{(y,z) \in \mathcal{Y} \times \mathcal{Z} : z \in S(f,y)\}$ is measurable.
10f	PositiveDrift [F-]	$\forall y \in \mathcal{Y}$, $\texttt{reachableWith}(f,y) \wedge \Phi(f,y) > 0 \to \forall z \in \text{Good}(f,y)$, $\Phi\big(f, \texttt{step}(z,y,f(\texttt{query}(y)))\big) \le \Phi(f,y) - \varepsilon(n)$.
10g	NegativeDriftAndJump [F-]	For y with $\texttt{reachableWith}(f,y)$, $\Phi(f,y) > 0$, and every $z \in \mathcal{Z}$: $\Phi(f, \texttt{step}(z,y,f(\texttt{query}(y)))) > \Phi(f,y) \to z \in \text{Worsen}(f,y)$, $\Phi(f, \texttt{step}(z,y,f(\texttt{query}(y)))) > \Phi(f,y) + \varepsilon(n) \to z \in \text{BigWorsen}(f,y)$; and, for every y with $\texttt{reachableWith}(f,y)$: $\forall z \in \text{Jump}(f,y),\ \Phi(f, \texttt{step}(z,y,f(\texttt{query}(y)))) = 0$.
10h	DriftSetMass [F-]	For $\texttt{reachableWith}(f,y)$ with $\Phi(f,y) > 0$: $p(n) \le \Pr(z \in \text{Good}(f,y))$, $\Pr(z \in \text{Worsen}(f,y)) \le p(n)/2$, $\Pr(z \in \text{BigWorsen}(f,y)) \le \varepsilon(n)\,p(n)/4$.
10i	JumpMass [F-]	$\forall y \in \mathcal{Y},\ p_{\text{jump}}(n) \le \Pr(z \in \text{Jump}(f,y))$ — holds at *every* y, unlike Stage 10h, since it underwrites integrability of the hitting time.
10j	RuntimeBound [F-]	$\mathbb{E}[T_A] \le \dfrac{4}{\varepsilon(n)\,p(n)}$.

A.1 Proofs of the drift condition

The additive drift theorem used in Section 2 is the following; a proof is given by Corus et al. (2018).

Theorem A.1 (Additive drift theorem)**.** *Let $(Z_t)_{t\in\mathbb{N}}$ be a discrete-time stochastic process in $[0,\infty)$ adapted to any filtration $(\mathscr{G}_t)_{t\in\mathbb{N}}$. For any $a \ge 0$, define $T_a := \min\{t \in \mathbb{N} \mid Z_t \le a\}$. If for some $\varepsilon > 0$ and all $t \in \mathbb{N}$: (1) $\mathbb{E}[Z_{t+1} - Z_t + \varepsilon\,; t < T_a \mid \mathscr{G}_t] \le 0$, (2) $\mathbb{E}[Z_t] < \infty$, and (3) $\mathbb{E}[T_a] < \infty$, then $\mathbb{E}[T_a \mid \mathscr{G}_0] \le Z_0/\varepsilon$.*

We present the proof of Corollary 2.1. The Lean formalisation of this proof, which was developed as part of the staged pipeline, follows almost excactly this proof.

Proof of Corollary 2.1. Define

$$E_1 := X_{t+1} \leq X_t - \varepsilon$$
$$E_2 := X_t - \varepsilon < X_{t+1} \leq X_t$$
$$E_3 := X_t < X_{t+1} \leq X_t + \varepsilon$$
$$E_4 := X_t + \varepsilon < X_{t+1}$$

Note that these definitions immediately imply

$$\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_1}(X_t - X_{t+1}) \geq \mathbb{1}_{\{t<T\}}\mathbb{1}_{E_1}\varepsilon \tag{2}$$
$$\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_2}(X_t - X_{t+1}) \geq 0 \tag{3}$$
$$\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_3}(X_t - X_{t+1}) \geq -\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_3}\varepsilon \tag{4}$$
$$\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_4}(X_t - X_{t+1}) \geq -\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_4} \tag{5}$$

where the last inequality follows because $X_t \in [0,1]$ for all t. We now have

$$\mathbb{1}_{\{t<T\}}(\varepsilon p/4) = \mathbb{1}_{\{t<T\}}(\varepsilon p - \varepsilon p/2 - \varepsilon p/4)$$

by assumptions 1-3

$$\leq \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_1}\varepsilon \mid \mathscr{G}_t\right] - \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_3\cup E_4}\varepsilon \mid \mathscr{G}_t\right] - \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_4} \mid \mathscr{G}_t\right]$$

since $\mathbb{1}_{E_3\cup E_4} \geq \mathbb{1}_{E_3}$

$$\leq \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_1}\varepsilon \mid \mathscr{G}_t\right] - \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_3}\varepsilon \mid \mathscr{G}_t\right] - \mathbb{E}\left[\mathbb{1}_{\{t<T\}}\mathbb{1}_{E_4} \mid \mathscr{G}_t\right]$$

by Eqs. (2), (4), and (5)

$$\leq \mathbb{E}\left[\mathbb{1}_{\{t<T\}}(\mathbb{1}_{E_1} + \mathbb{1}_{E_3} + \mathbb{1}_{E_4})(X_t - X_{t+1}) \mid \mathscr{G}_t\right]$$

by Eq. (3) and $\mathbb{1}_{E_1} + \mathbb{1}_{E_2} + \mathbb{1}_{E_3} + \mathbb{1}_{E_4} = 1$

$$\leq \mathbb{E}\left[\mathbb{1}_{\{t<T\}}(X_t - X_{t+1}) \mid \mathscr{G}_t\right].$$

The result now follows by rearranging the inequality above. □

Algorithm 2 (1+1) EA

1: $t := 0$
2: Choose an initial bit string $x_0 \in \{0,1\}^n$ uniformly at random
3: **while** forever **do**
4: | Create x' by flipping each bit in x_t independently with probability p (mutation)
5: | $x_{t+1} := x'$ if $f(x') \geq f(x)$, and $x_{t+1} := x_t$ otherwise (selection)
6: | $t := t + 1$
7: **end while**

B IMPLEMENTATION AND PROMPT

Implementation. The pipeline code was initially based on the Goedel-Prover-V2 codebase (Lin et al., 2025). Every candidate declaration is checked by Lean 4 on a Kimina Lean server (Dos Santos et al., 2025), and the Lean errors are returned to the model in the next attempt. The responses to the prompts of each stage come from a *backend*. With the *CLI backend*, the pipeline writes each prompt to standard output and reads the response from standard input, so that the pipeline can be driven by an external agent; rows 1–18 of Table 1 use this backend with Claude models. With the *vLLM backend*, the pipeline serves open-weight models itself with vLLM (Kwon et al., 2023), sampling many candidates per attempt; rows 19–21 use this backend with Goedel-Prover-V2-32B for the formal stages and Gemma 4 for the prose stages. The stages, their Lean statements and the checks are the same for both backends.

The prompt. Every stage prompt starts with the same global block, followed by the accepted declarations so far and the task of the stage. The global block describes the framework: the black-box algorithm structure of Algorithm 1, the problem-class interface and the baseline `algorithm0` (random search), whose definitions the agent can use as a template. It then lists an API of library results, whose signatures are checked against the Lean build. These include samplers that turn the random source into bitstrings, with a fixed or a state-dependent per-bit bias, together with inverse lemmas that construct a random input producing any given bitstring (used for `Completeness1` and `NonAbsorbing`), bit-level lemmas for `getBit`, `flipBit` and exclusive or, and the probability that a sampler returns a given bitstring, which the mass stages 10h and 10i use. A measurability toolkit provides two tactics that decompose tuple-, `if`- and arithmetic-structured step and potential functions into leaves, and gives the idioms that close each kind of leaf. For the drift stages, the block describes the random source, the reachability predicates and the assembly theorem of Stage 10j, and it explains which properties of that theorem constrain the design, e.g. that a single ε and p must serve every reachable non-optimal state. The problem card (Sections C and E) is spliced in as the only problem-specific part. Finally, a proof cookbook collects recurring Lean pitfalls and their fixes: write selection with a positive guard (`if b then`) rather than `if ¬b`, which elaborates to a term that `simp` cannot reduce; never call `decide` on goals involving a symbolic n; how to show that two state tuples differ; and to prove measurability with the toolkit rather than with `aesop`, whose fixed search budget times out on nested `if`s. After a failed attempt the agent sees the failing declaration, the Lean error, its earlier attempts at the stage and a ledger of facts verified in earlier stages.

C THEORETICAL BENCHMARK FUNCTIONS

For $y \in \{0,1\}^n$ write $|y|_1 = \sum_{i=1}^n y_i$ for the number of ones. The three benchmark functions are

$$\textsc{OneMax}(y) = |y|_1 = \sum_{i=1}^{n} y_i,$$

$$\textsc{LeadingOnes}(y) = \sum_{i=1}^{n} \prod_{j=1}^{i} y_j,$$

$$\textsc{Jump}_k(y) = \begin{cases} k + |y|_1 & \text{if } |y|_1 \le n-k \text{ or } |y|_1 = n, \\ n - |y|_1 & \text{otherwise,} \end{cases}$$

so OneMax counts the ones, LeadingOnes counts the ones before the first zero, and $\textsc{Jump}_k$ behaves like OneMax except for a gap of width k below the optimum 1^n, in which the fitness points away from the optimum. For the $(1{+}1)$ EA with mutation rate $1/n$ the expected optimisation times are $\Theta(n \log n)$ on OneMax, $\Theta(n^2)$ on LeadingOnes and $\Theta(n^k)$ on $\textsc{Jump}_k$ for constant $k \ge 2$ (Droste et al., 2002).

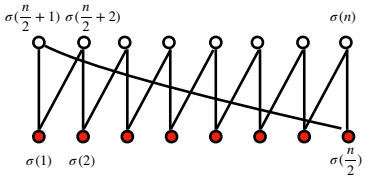
A black-box algorithm must not depend on the instance, so each function is turned into a class of instances by hiding its optimum. With $\oplus$ the bitwise exclusive or, $\bar{z}$ the complement of z and $\mathrm{perm}_\pi(x)_i = x_{\pi(i)}$ for a permutation π of $\{1,\dots,n\}$, the problem cards used in our experiments define

$$\begin{aligned} F_z(x) &= \textsc{OneMax}(x \oplus z), && z \in \{0,1\}^n, \\ F_{z,\pi}(x) &= \textsc{LeadingOnes}\big(\mathrm{perm}_\pi(x \oplus z)\big), && z \in \{0,1\}^n,\ \pi \text{ a permutation}, \\ F_z(x) &= \textsc{Jump}_k(x \oplus z), && z \in \{0,1\}^n. \end{aligned}$$

Each instance has the unique optimum $\bar{z}$. For $\textsc{Jump}_k$ the gap width $k \ge 2$ is a constant of the class that the agent does not know, so every proof must be uniform in k. Algorithms whose mutation treats all bits alike, such as the $(1{+}1)$ EA, have the same expected optimisation time on every instance of a class as on the underlying function.

D CYCLEUNCOVER: DEFINITION AND RUNTIME OF THE (1+1) EA

This section proves Theorem 4.1. The objective `eval`, the count `count` and the feasibility condition `guard` are those of Figure 3: `guard`(x) holds exactly when the chosen vertices form an independent



Given a permutation $\sigma : [n] \to [n]$, define

$$\texttt{count}(x) := \sum_{j=1}^{n/2} x_{\sigma(j)} + x_{\sigma(n/2+j)}$$

$$\texttt{guard}(x) := \forall j \in [n/2],\ x_{\sigma(j)} + x_{\sigma(n/2+j)} \leq 1 \\ \wedge\ x_{\sigma(j)} + x_{\sigma(n/2+j+1 \bmod n/2)} \leq 1$$

$$\texttt{eval}(x) := \begin{cases} n + \texttt{count}(x) & \text{if } \texttt{guard}(x) \\ \texttt{count}(x) & \text{otherwise} \end{cases}$$

Figure 3: Right: Problem definition in human readable form. Left: A visualisation of the problem instance (not shown to the agent). The problem is independent set on cycle graphs where an optimal selection of vertices are indicated in red (not revealed to the agent).

set of the hidden cycle, and $\texttt{eval}(x) = n + \texttt{count}(x)$ if $\texttt{guard}(x)$ and $\texttt{eval}(x) = \texttt{count}(x)$ otherwise.

Proof of Theorem 4.1. Without loss of generality, we assume that σ is the identity-permutation.

We condition on the event that both endpoints u and v of the first edge are 1-bits in the initial search point, and that it contains at least $9n/20$ 1-bits. By a union bound and a Chernoff bound (Doerr, 2020), the probability of this event is at least $1/4 + (1 - e^{-\Omega(n)}) - 1 \geq 1/5$. If this event does not happen, we lower bound the runtime by 0.

We consider two phases, the first phase consisting of the first $t = 3en$ iterations, and the second phase consisting of the following e^n iterations. We consider phase 1 a failure (from the perspective of the analysis) if bit u or v is flipped during this phase. The probability of this failure event is upper bounded by $1 - (1 - 1/n)^{2t} \leq 1 - e^{-2t/(n-1)}$. If phase 1 is not a failure, then $\texttt{guard}(x)$ is never satisfied, and the fitness is $\texttt{count}(x)$, i.e., the number of 1-bits is non-decreasing.

We call an iteration in the first phase successful if either the current search point has at least $5n/6$ 1-bits, or if mutation increases the number of 1-bits. The probability of a successful iteration is at least

$$p \geq \frac{n}{6}\left(1 - \frac{1}{n}\right)^{n-1}\frac{1}{n} \geq \frac{1}{6e}.$$

The number of successful iterations in phase 1 dominates stochastically a binomially distributed random variable with parameters t and p. By a Chernoff bound, the probability that there are less than

$$tp(1 - 1/6) \geq (n/2)(1 - 1/6) = 5n/12$$

successes is $e^{-\Omega(n)}$, and otherwise we call phase 1 a failure.

Assuming no failure in phase 1, the current search point x at the start of phase 2 contains $|x| \geq \min(5n/6, 9n/20 + 5n/12) = 5n/6$ 1-bits. We now prove inductively that in each of the following e^n iterations, the number of 1-bits in the current search point is at least $5n/6$. If the current point is infeasible with at least $5n/6$ 1-bits, an offspring is accepted only if it is feasible or has at least as many 1-bits. In order to satisfy $\texttt{guard}(x)$ i.e. the solution is an independent set, it is necessary to have at most $n/2$ 1-bits, i.e., at least $n/3$ bits must be flipped simultaneously. By a union bound, this type of failure has probability at most $e^n n^{-\Omega(n)} = n^{-\Omega(n)}$.

Condition on the good starting event and no failures, the running time of the algorithm is at least e^n. By a union bound, the probability of a failure in phase 1 or 2 is at most $1 - e^{-2t/(n-1)} + e^{-\Omega(n)} + n^{-\Omega(n)} < c$ for some constant $c \in (0, 1)$. The unconditional expected runtime is therefore at least $(1/5)(1 - c)e^n = e^{\Omega(n)}$. □

E SYNTHESISED ALGORITHMS, POTENTIALS AND RUNTIME BOUNDS OF ALL RUNS

This appendix collects, for every run summarised in Table 1, the synthesised algorithm, the potential function and the runtime bound. The pseudocode is a human-readable transcription of the frozen Lean definitions of `init` and `step`; in every run `query` returns the first component of the state. We write $\mathrm{bits}_p(u) \in \{0,1\}^n$ for the per-bit sampler that sets bit i iff $u_i \le p_i$ (so $\Pr[\mathrm{bits}_p(u)_i = 1] = p_i$; a scalar p means $p_i = p$ for all i), $u^{+n} := (u_n, u_{n+1}, \dots)$ for the coordinates of the same draw from index n on, $\oplus$ for bitwise XOR, $|x|_1$ for the number of ones, e_v for the v-th unit vector, $\bar{x}$ for the complement and $[P] \in \{0,1\}$ for the indicator of P. Unless stated otherwise, f is the fitness of the pending query q passed to `step`, and a potential $\Phi_z(s)$ is 0 if and only if the query of s is fitness-maximal. For staged runs the runtime theorem is $\mathbb{E}[T_A] \le 4/(\varepsilon_1 p_1)$ for all $n \ge 3$ and every instance z, with the drift constants ε_1 and p_1 (the Good-set mass) given; "machine-checked" means the file compiles with no `sorry`.

Problem cards. ONEMAX: $F_z(x) = \textsc{OneMax}(x \oplus z)$, unique optimum $\bar{z}$. LEADINGONES (card `lo`): an instance is a pair (z, π) of a string z and a permutation π of the positions, and $F_{z,\pi}(x) = \mathrm{LO}\big(\mathrm{perm}_\pi(x \oplus z)\big)$ with $\mathrm{perm}_\pi(x)_i = x_{\pi(i)}$ and $\mathrm{LO}(y) = \sum_{i=1}^{n} \prod_{j \le i} y_j$; the unique optimum is $\bar{z}$. The second version of the card also supplies the inequality $e^{-m/(r-1)} \le (1 - 1/r)^m$ for $r > 1$. $\textsc{Jump}_k$ (card `jump_k`): $F_z(x) = \textsc{Jump}_k(x \oplus z)$, where $\textsc{Jump}_k(y) = k + |y|_1$ if $|y|_1 \le n - k$ or $|y|_1 = n$, and $\textsc{Jump}_k(y) = n - |y|_1$ otherwise; the unique optimum is $\bar{z}$. The gap width $k \ge 2$ is an opaque constant, so every proof must be uniform in k. SUBMODULARMIN, integer-valued (card `submodular_minimisation_integer_valued_additional_advice`): an instance is a submodular function $f : \{0,1\}^n \to \{0, 1, \dots, B\}$ with $B = n^c$ for an opaque class constant $c \ge 1$ and $\min_x f(x) = 0$, and $F_f(x) = -f(x)$. Besides the definitions, the card supplies only facts about the integer range (every non-optimal point has $F_f \le -1$, and distinct values differ by at least 1) and the advice quoted in Section E.16; it supplies no lemmas about submodular functions. CYCLEUNCOVER (card `laminate`): the objective `eval` of Figure 3, with $h = \lfloor n/2 \rfloor$, optimum value $3h$. VERTEXCOVER, *flat fitness* (cards `vertex_cover`, early `bandwidth_vertex_cover`): $F_z(x) = -|x|_1$ if x is a cover of G_z and $|x|_1 - 2n$ otherwise. VERTEXCOVER, *graded fitness* (later `bandwidth_vertex_cover`, `wcvc`, `well_covered_vertex_cover`): $F_z(x) = -\big((n+1)\,\mathrm{unc}_z(x) + |x|_1\big)$, where $\mathrm{unc}_z(x)$ is the number of edges with no chosen endpoint; covers are exactly the points with $F_z \ge -n$. For the bandwidth cards every edge uv satisfies $|u - v| \le k$, and k is known to the algorithm; the well-covered cards restrict to graphs whose minimal covers are all minimum. The `wcvc` card is an anonymised copy of the well-covered card with all names and derived facts removed.

E.1 ONEMAX, OPEN-WEIGHT MODELS (STAGED; INCOMPLETE)

These runs used the vLLM backend, which serves the models itself, instead of the CLI backend used for the other runs (Section B). The formal stages were generated by Goedel-Prover-V2-32B with 256 samples per attempt at temperature 1, and the prose stages (0, 9a, 10a) by Gemma 4 26B-A4B, both served with vLLM on 3 NVIDIA L40 GPUs (Sulis) or 4 NVIDIA GH200 GPUs (Isambard-AI). All runs are on ONEMAX with earlier versions of the staged pipeline that have the same 28 stages as Table 2. The token usage of the formal model is not logged by this backend, and the Lean code of these runs is not included.

Table 3 lists the 14 runs made with the three most recent of these pipeline versions (September 2026). No run completed: each ended after 21–46 h of wall-clock time. Ten runs got stuck at Stage 4 (`NotRandomSearch`), three at Stage 1c or 2, and one reached Stage 9d (`PotentialFineGrained`). Rows 19–21 of Table 1 are, in order, the most recent run on each cluster with the most recent pipeline used there, and the run that got furthest.

All samples of the runs in rows 19–21 are logged, so the output tokens of Goedel-Prover-V2-32B can be counted with its own tokeniser over the logged responses; the prose stages add at most 7×10^4 tokens per run. Lean lines are counted as in Table 1, at the last passed stage.

Row 19 (`generous-dragon-779`, Sulis, `optimisation`). The frozen design samples each bit with its own bias, initialised to $1/2$; after an improvement, every bias moves by 0.1 towards the

corresponding bit of the query, clamped to $[0.1, 0.9]$. This is a PBIL-like estimation-of-distribution algorithm, not the reference-point-plus-mask design described in the Stage 0 proposal. Stages 0–3 passed. Stage 4 is true for this design (the biases leave $1/2$ after one improving step), but all 13×256 samples for it failed: 38% contained no extractable Lean code, 35% left unsolved goals, 10% did not parse and 6% used unknown identifiers. The pipeline backtracked to Stages 3, 2 and 1c, but never to the design stages. 5,896 samples, 25.1M output tokens, 23.8 h.

Row 20 (`avid-waterfall-734`, Isambard-AI, `split_v9`). Both designs, the first and the one proposed after a backtrack to Stage 0, are PBIL-like: per-bit biases, learning rate $1/n$, clamped to $[0.05, 0.95]$ and then $[0.1, 0.9]$. The update moves the biases towards the newly drawn query rather than the evaluated one, so the fitness only decides whether they move. Stage 4 is again true, and all 24×256 samples for it failed: 53% left unsolved goals, 17% used unknown identifiers, 11% did not parse and 8% contained no extractable code. 10,776 samples, 73.1M output tokens, 23.6 h.

Row 21 (`golden-yogurt-711`, Isambard-AI, `split_v8`). The biases are initialised to $1/2$ and never updated, so the algorithm is uniform random search with an incumbent. Stage 4 nevertheless passed (2 of 256 samples), because in this pipeline version the `NotRandomSearch` obligation did not require the witness state to be reachable: the accepted proof uses a state with all biases equal to 1, which the algorithm never reaches. This is the unreachable-witness loophole discussed with the `NotRandomSearch` guard; from `split_v9` onwards the obligation requires a reachable state (Table 2). The run went on to define the potential $1 - \textsc{OneMax}(q \oplus z)/n$ and passed Stage 9c three times, but all 15×256 samples for Stage 9d failed, although the statement is true for this potential: 31% omitted the required lemma, 25% left unsolved goals, 11% did not parse, 10% used unknown identifiers, and 16% recorded no Lean error. 22,584 samples, 124.3M output tokens, 36.8 h.

E.2 OneMax, Claude Opus 5 (staged; complete)

Algorithm 3 (1+1)-style hill-climber with rate $1/(n+1)$ (OneMax, Claude Opus 5).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$ ▷ *accept (always at the first step)*
  end if
  $p_i \leftarrow n/(n+1)$ if $c_i = 1$, else $1/(n+1)$, for all i ▷ *flip each bit w.p.* $\frac{1}{n+1}$
  return $(\mathrm{bits}_p(u),\ \textsc{true},\ c,\ c_f)$
end procedure

Potential. With $M(s) = \max\{F_z(q), F_z(x^*)\}$ if b, else $M(s) = F_z(q)$,

$$\Phi_z(s) = [q \neq \bar{z}] \cdot \Big(\tfrac{1}{n+1} + \tfrac{n}{n+1} \big(1 - \tfrac{M(s)}{n}\big) \Big).$$

Runtime. $\varepsilon_1 = \frac{1}{n+1}$, $p_1 = \frac{2}{(n+1)^2}$: $\mathbb{E}[T_A] \leq 2(n+1)^3 = O(n^3)$. Machine-checked; speed-up certificate $\delta = 1$, $n_0 = 11$.

E.3 LEADINGONES, CLAUDE OPUS 5.5 (STAGED; COMPLETE)

Algorithm 4 Fitness-dependent mutation around an elitist reference point (LEADINGONES, Claude Opus 5.5).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \text{FALSE},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$ ▷ *accept (always at the first step)*
  end if
  $r \leftarrow 1/(\max\{c_f, 0\} + 2)$ ▷ *rate read from the learned fitness*
  return $(c \oplus \mathrm{bits}_r(u),\ \text{TRUE},\ c,\ c_f)$
end procedure

Potential. With $M(s) = \max\{F(q), F(x^*)\}$ if b, else $M(s) = F(q)$,

$$\Phi(s) = [q \neq \bar{z}] \cdot \frac{n + 1 - M(s)}{n + 1}.$$

Drift sets. $\mathrm{Good}_1(s)$ and $\mathrm{Jump}_1(s)$ are the draws after which the potential drops by at least ε_1, respectively reaches 0; $\mathrm{Worsen}_1 = \mathrm{BigWorsen}_1 = \emptyset$ by elitism and the invariant $f^* = F(x^*)$ on reachable states. The mass bound uses the draws that keep the correct prefix and flip the first wrong position, and lower-bounds $(1 - r)^c$ by Bernoulli's inequality. *Runtime.* $\varepsilon_1 = \frac{1}{n+1}$, $p_1 = \frac{1}{(n+2)^2}$, $p_{\text{jump}} = (n + 2)^{-n}$: $\mathbb{E}[T_A] \leq 4(n + 1)(n + 2)^2 = O(n^3)$. Machine-checked; $\delta = 1$, $n_0 = 12$. The rate is, up to its offset, the fitness-dependent rate $1/(\mathrm{LO} + 1)$ of Böttcher et al. (2010). The operator's true runtime is $\Theta(n^2)$, and the factor n is lost in the Bernoulli step.

E.4 LEADINGONES WITH THE e^{-1} LEMMA, CLAUDE OPUS 5.5 (STAGED; COMPLETE)

This session used the second version of the card and reached the same design as Section E.3 independently: `init`, `step` and the potential are those of Algorithm 4, except that the mask is drawn with the per-bit sampler and a constant rate vector, which gives the same distribution. The first design used the scalar sampler, and was resubmitted in this form after two backtracks caused by its measurability proof.

Drift sets. With $s'_u = (q'_u, \text{TRUE}, c, c_f)$ the successor state, $\mathrm{Jump}_1(s) = \{u : q'_u = \bar{z}\}$ and $\mathrm{Good}_1(s) = \mathrm{Jump}_1(s) \cup \{u : F(c) + 1 \leq F(q'_u)\}$; $\mathrm{Worsen}_1 = \mathrm{BigWorsen}_1 = \emptyset$ as before. The mass bound uses the same draws as Section E.3, but lower-bounds $(1 - r)^c$ by e^{-1} using the inequality supplied by the card. *Runtime.* $\varepsilon_1 = \frac{1}{n+1}$, $p_1 = \frac{e^{-1}}{n+2}$, $p_{\text{jump}} = (n + 2)^{-n}$: $\mathbb{E}[T_A] \leq 4e(n + 1)(n + 2) = O(n^2)$, the true order of the operator. Machine-checked; $\delta = 1$, $n_0 = 9$. The last three stages were pre-checked against the Lean server before submission. The agent also read the card's withheld lemmas at Stage 10h; none of them is used, but one rewrite chain in the proof of the prefix-count lemma was copied from the proof of one of them.

E.5 JUMP$_k$, CLAUDE OPUS 5.5 (STAGED; COMPLETE)

Algorithm 5 (1+1)-style hill-climber with rate $1/(n+1)$ (JUMP$_k$, Claude Opus 5.5).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$ ▷ *accept (always at the first step)*
  end if
  return $(c \oplus \mathrm{bits}_{1/(n+1)}(u),\ \textsc{true},\ c,\ c_f)$
end procedure

In substance this is the (1+1) EA with mutation rate $1/(n+1)$: the Stage 0 request for a structural change against the baseline was not met.

Potential. With $N = n + k$, $M(s) = \max\{F_z(q), f^*\}$ if b, else $M(s) = F_z(q)$, and $M(s)$ clamped to $[0, N]$,

$$\Phi_z(s) = \begin{cases} 0 & \text{if } F_z(q) = F_z(\bar{z}), \\ \dfrac{N + 1 - \min\{\max\{M(s), 0\}, N\}}{N+1} & \text{otherwise.} \end{cases}$$

Drift sets. $\mathrm{Good}_1(s)$ and $\mathrm{Jump}_1(s)$ are the draws after which the potential drops by at least ε_1, respectively reaches 0. The stored fitness after a step equals the old level, so the potential never increases and $\mathrm{Worsen}_1 = \mathrm{BigWorsen}_1 = \emptyset$ without any invariant. The mass bound uses one explicit mask with at most k ones: if the reference point c lies on the slope ($|c \oplus z|_1 < n - k$), the mask flips one bit on which c agrees with z; otherwise it is $c \oplus \bar{z}$, which jumps straight to the optimum. Its probability is at least $p^k(1-p)^n \geq p^{k+1}$ with $p = \frac{1}{n+1}$, by Bernoulli's inequality. The deceptive branch of the objective is never evaluated. ***Runtime.*** $\varepsilon_1 = \frac{1}{n+k+1}$, $p_1 = (n+1)^{-(k+1)}$, $p_{\text{jump}} = (n+1)^{-n}$: $\mathbb{E}[T_A] \leq 4(n+k+1)(n+1)^{k+1} = O(n^{k+2})$ for every $n \geq 3$, uniformly in k. Machine-checked; $\delta = 1$, and n_0 depends on k and is obtained non-constructively. The (1+1) EA is known to need $\Theta(n^k)$ expected time on JUMP$_k$ (Droste et al., 2002), so the certified bound is a factor n^2 above the true order. All submissions were blind. One rejected submission at Stage 10h was a deliberate probe whose unsolved goals exposed the definitions of ONEMAX, `getBit` and `flipBit`.

E.6 JUMP$_k$ WITH THE e^{-1} LEMMA, CLAUDE OPUS 5.5 (STAGED, 3 GENERATIONS; COMPLETE)

One CLI-backend session with blind submissions (wandb run `celestial-lake-792`) on the card `jump_k_exp`, which is the JUMP$_k$ card of Section E.5 plus the inequality $e^{-m/(r-1)} \leq (1 - 1/r)^m$ (Section C). The pipeline was run for three generations, each seeded with the algorithm and bound of the previous one.

Generation 1. Strictly elitist reference point; the mask is drawn with a learned per-bit rate, reset to $1/(n+2)$ on a strict improvement, multiplied by $1 + 1/n$ otherwise and clamped to $[1/(n+2), 2/(n+4)]$. The certificate uses only the clamp: $\varepsilon_1 = \frac{1}{n+k+1}$, $p_1 = (n+2)^{-k}e^{-2}$, so $\mathbb{E}[T_A] \leq 4e^2(n+k+1)(n+2)^k = O(n^{k+1})$. This is one factor n better than Section E.5; the factor is saved exactly by the supplied inequality, which replaces a Bernoulli bound on $(1-p)^n$ by e^{-2}. *Generation 2.* A fixed rate $2/(n+2)$: $\varepsilon_2 = \frac{1}{2(n+k+1)}$, $p_2 = \left(\frac{2}{n+2}\right)^k e^{-2}$, so $\mathbb{E}[T_A] \leq 2^{3-k}e^2(n+k+1)(n+2)^k$. *Generation 3* (row 5 of Table 1). Rate cycling with a learned phase bit: the phase is reset on a strict improvement and toggled on every failure, and the next mask rate is $3/(n+2)$ if the phase is set and $2/(n+2)$ otherwise. $\varepsilon_3 = \frac{1}{n+k+1}$ and $p_3 = \min\left\{(\frac{2}{n+2})^k e^{-2},\ (\frac{3}{n+2})^k e^{-n/((n+2)/3-1)}\right\}$, so $\mathbb{E}[T_A] \leq 4(n+k+1)/p_3 = O(n^{k+1})$, with a certified speed-up of $\delta = 2/5$ over generation 2 for

$n > 60$. All three generations are machine-checked with no `sorry`; generation 3 has 1,335 lines of Lean, and the three generations used 396k output tokens.

E.7 CycleUncover, Claude Fable 5.1 (staged; complete)

Write $H = 2\lfloor n/2 \rfloor$ and $\rho = 1/(n+1)$. The state (q, A, B, r, a, k) holds the query, the indicator strings A, B of the vertices labelled with the two colours so far, the frontier (last labelled vertex) r, its colour a, and the number k of labelled vertices.

Algorithm 6 Cycle walk (CycleUncover, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_\rho(u),\ 0^n,\ 0^n,\ 0^n,\ \textsc{false},\ 0)$
end procedure
procedure `step`$(u,\ (q, A, B, r, a, k),\ f)$
  $\pi \leftarrow A$ if $k = H$, else r; $T \leftarrow \pi \oplus q$ ▷ *the mask just used*
  if $f = 2$ **and** $0 < k < H$ **and** $|T|_1 = 1$ **and** $T \wedge (A \oplus B) = 0^n$ **then**
    if a **then**
      $B \leftarrow B \oplus T$
    else
      $A \leftarrow A \oplus T$
    end if
    $r \leftarrow T;\ a \leftarrow \neg a;\ k \leftarrow k + 1$ ▷ *T is adjacent to the frontier*
  else if $f = H + 1$ **and** $k = 0$ **and** $|T|_1 = 1$ **then**
    $A \leftarrow T;\ r \leftarrow T;\ a \leftarrow \textsc{true};\ k \leftarrow 1$ ▷ *T is a read coordinate*
  end if
  $\pi' \leftarrow A$ if $k = H$, else r
  return $(\pi' \oplus \mathrm{bits}_\rho(u),\ A,\ B,\ r,\ a,\ k)$
end procedure

Potential. With $k^+(s)$ the label count after one `step` from s fed the true fitness (independent of the draw) and truncated subtraction $\dot{-}$, $\Phi_z(s) = 0$ if $F_\sigma(q) = 3h$, else $\Phi_z(s) = \big(1 + (H \dot{-} k^+(s))\big)/(H + 2)$. *Runtime.* $\varepsilon_1 = \frac{1}{H+2}$, $p_1 = \frac{1}{3(n+1)}$: $\mathbb{E}[T_A] \le 12(n+1)(2\lfloor n/2 \rfloor + 2) \le 12(n+1)(n+2) = O(n^2)$. Machine-checked; $\delta = 1$, $n_0 = 10$. This is a learn-then-solve design that uses exact fitness values as an adjacency oracle.

E.8 CycleUncover, Claude Fable 5.1 (zero-shot ablation; complete)

Algorithm 7 Random search with a recorded incumbent (CycleUncover, zero-shot, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$
  end if
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{true},\ c,\ c_f)$ ▷ *the query ignores the state*
end procedure

Potential. None (the ablation has no drift framework). *Runtime.* Two optima (the colour classes of the hidden cycle) are derived from the definition, so each query succeeds with probability 2^{1-n} and a tail-sum argument gives $\mathbb{E}[T_A] \le 2^n/2 = \Theta(2^n)$. Machine-checked; the bound beats the 2^n bar for all n.

E.9 CycleUncover, Claude Sonnet (staged; incomplete)

Five designs were tried; all are elitist bit-flip searches with state (q, b, f_b) and the mutation $p_i = (n-1)/n$ if $b_i = 1$, else $1/n$. The last design with Lean artefacts is Design 4:

Algorithm 8 Design 4 (CycleUncover, Claude Sonnet).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ 0^n,\ 2h)$ ▷ *0^n has fitness $2h$*
end procedure
procedure `step`$(u,\ (q, b, f_b),\ f)$
  if $f > f_b$ **then**
    $(b, f_b) \leftarrow (q, f)$
  end if ▷ *Design 5: also accept ties if $f_b > 2h$*
  $p_i \leftarrow (n-1)/n$ if $b_i = 1$, else $1/n$, for all i
  return $(\mathrm{bits}_p(u),\ b,\ f_b)$
end procedure

Potential. Design 4: $\Phi_z(s) = \big(3h - F_\sigma(q)\big)/(3h)$. Design 5: the normalised Hamming distance of the read bits of q to the nearer of the two colour classes, $\Phi_z(s) = \frac{1}{2h}\min\{d_H(q, \mathrm{class}_A), d_H(q, \mathrm{class}_B)\}$. *Runtime.* None proved. Design 4 reached the Worsen/Jump stage with $\varepsilon_1 = \frac{1}{3h}$, $p_1 = e^{-1}/n$, but the Good set is empty at reachable maximal-but-not-maximum independent sets, so its mass bound (and hence the would-be $\Theta(n^2)$ bound $12e\,hn$) is false.

E.10 VertexCover, Claude Fable 5.1 (staged; complete)

Flat fitness; the treewidth parameter of the card is unused. Let $N = n^2$, $\rho = 1/(n+1)$, $u_j = \lfloor j/n \rfloor \bmod n$, $v_j = j \bmod n$, let $\mathrm{probe}(j)$ be 1^n with bits u_j, v_j cleared, let $\mathrm{opt}(A)$ be a minimum-size string covering every edge recorded in the table A, and $\mathrm{target}(k, A) = \mathrm{probe}(k)$ if $k < N$, else $\mathrm{opt}(A)$.

Algorithm 9 Edge probing, then solve (VertexCover, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{probe}(0) \oplus \mathrm{bits}_\rho(u),\ 0,\ A_\emptyset)$ ▷ *empty edge table*
end procedure
procedure `step`$(u,\ (q, k, A),\ f)$
  if $q = \mathrm{target}(k, A)$ **then** ▷ *hit: the probe was queried exactly*
    $A[u_k, v_k] \leftarrow [f < -n]$ ▷ *probe is a non-cover iff $u_k v_k$ is an edge*
    if $k < N$ **then**
      $k \leftarrow k + 1$
    end if
  end if
  return $(\mathrm{target}(k, A) \oplus \mathrm{bits}_\rho(u),\ k,\ A)$
end procedure

Potential. $\Phi_z(s) = \frac{1}{2} + U(s)/\big(2(2N+1)\big)$ off the optimum, with $U(s) = 2\big(N - \min(k, N)\big) + [q \neq \mathrm{target}(k, A)]$. *Runtime.* $\varepsilon_1 = \frac{1}{2(2n^2+1)}$, $p_1 = \frac{1}{n+1}$: $\mathbb{E}[T_A] \le 8(2n^2+1)(n+1) = O(n^3)$. Machine-checked; $\delta = 1$, $n_0 = 18$. Learn-then-solve: $\mathrm{opt}(A)$ is an exact (exponential-time) solver inside `step`; the bound counts queries only.

E.11 BandwidthVC, Claude Fable 5.1 (staged, flat fitness; complete)

Let $K = k + 1$, $L = 2nK$, $a_p = \lfloor p/K \rfloor$, $b_p = a_p + 1 + (p \bmod K)$, let $\mathrm{probe}(p)$ be 1^n with bits a_p, b_p cleared, $\mathrm{opt}(A)$ as above, $\mathrm{plan}(t, A) = \mathrm{probe}(t/2)$ if t is even and $t < L$, else $\mathrm{opt}(A)$, and $\beta(y)_i = n/(n+1)$ if $y_i = 1$, else $1/(n+1)$.

Algorithm 10 Interleaved window probing and solving (BANDWIDTHVC, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_{\beta(\mathrm{plan}(0,A_\emptyset))}(u),\ 0,\ A_\emptyset)$
end procedure
procedure `step`$(u,\ (q,t,A),\ f)$
  if $q = \mathrm{plan}(t,A)$ **and** $t < L$ **then**
    if t even **then**
      $A[a_{t/2}, b_{t/2}] \leftarrow [f < -n]$
    end if
    $t \leftarrow t+1$
  end if
  return $(\mathrm{bits}_{\beta(\mathrm{plan}(t,A))}(u),\ t,\ A)$
end procedure

Potential. $\Phi_z(s) = \big(2(L \dot{-} t) + [q \neq \mathrm{plan}(t,A)] + 1\big)/(2L+2)$ off the optimum. *Runtime.* $\varepsilon_1 = \frac{1}{2L+2}$, $p_1 = \frac{1}{4}$: $\mathbb{E}[T_A] \le 16(2L+2) = 64n(k+1)+32 = O(kn)$. Machine-checked; $\delta = 1$, $n_0 = 16(k+1)+6$. Learn-then-solve, as in Section E.10, but using the bandwidth window.

E.12 BANDWIDTHVC, CLAUDE FABLE 5.1 (STAGED, GRADED FITNESS; INCOMPLETE)

Clamp $\kappa(\rho) = \min\{\frac{2}{n+4}, \max\{\frac{1}{2(n+4)}, \rho\}\}$.

Algorithm 11 Direction-adaptive asymmetric mutation (BANDWIDTHVC, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0,\ \frac{1}{n+4},\ \frac{1}{n+4})$
end procedure
procedure `step`$(u,\ (q,b,a,f_a,\rho_\downarrow,\rho_\uparrow),\ f)$
  if not b **then**
    $(a, f_a) \leftarrow (q, f)$
  else if $f \ge f_a$ **then**
    if $|q|_1 < |a|_1$ **then**
      $\rho_\downarrow \leftarrow \kappa(2\rho_\downarrow);\ \rho_\uparrow \leftarrow \kappa(\rho_\uparrow/2)$
    else
      $\rho_\downarrow \leftarrow \kappa(\rho_\downarrow/2);\ \rho_\uparrow \leftarrow \kappa(2\rho_\uparrow)$
    end if
    $(a, f_a) \leftarrow (q, f)$
  end if
  $p_i \leftarrow \kappa(\rho_\downarrow)$ if $a_i = 1$, else $\kappa(\rho_\uparrow)$, for all i
  return $(a \oplus \mathrm{bits}_p(u),\ \textsc{true},\ a,\ f_a,\ \rho_\downarrow,\ \rho_\uparrow)$
end procedure

Potential (final, effective anchor). With $e(s) = q$ if $\neg b$ or $F_z(q) \ge F_z(a)$, else a: off the optimum $\Phi_z(s) = \frac{1}{2} + \frac{-F_z(e(s))}{4n}$ if $e(s)$ is a cover and $\Phi_z(s) = \frac{3}{4} + \frac{\mathrm{unc}_z(e(s))}{4(kn+1)}$ otherwise. *Runtime.* None proved. $\varepsilon_1 = \frac{1}{4((k+1)n+1)}$, $p_1 = \frac{1}{16(n+4)}$ and the speed-up certificate ($\delta = 1$, $n_0 = 640(k+2)$) are machine-checked, and all stages up to the Worsen/Jump stage passed. The Good-set mass bound is false: on an odd path the even vertices form a reachable minimal but non-minimum cover with no improving single flip. The would-be bound $256((k+1)n+1)(n+4) = O(kn^2)$ is therefore not certified.

E.13 BANDWIDTHVC, CLAUDE FABLE 5.1 (ZERO-SHOT ABLATION; INCOMPLETE)

Algorithm 12 $(1{+}1)$ EA with rate $1/n$ (BANDWIDTHVC, zero-shot, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$
  end if
  return $(c \oplus \mathrm{bits}_{1/n}(u),\ \textsc{true},\ c,\ c_f)$
end procedure

Potential. None. *Runtime.* The declared bound $\mathbb{E}[T_A] \le 2^n/n$ was left as `sorry`; only the (vacuous) comparison $2^n/n < 2^n$ is machine-checked.

E.14 BANDWIDTHVC, CLAUDE FABLE 5.1, SECOND SESSION (STAGED; INCOMPLETE)

Algorithm 13 Fill-then-prune mutation (BANDWIDTHVC, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$
  end if
  if $c_f < -n$ **then** ▷ *incumbent is not a cover: fill*
    $p_i \leftarrow 1/(n+1)^2$ if $c_i = 1$, else $1/2$, for all i
  else ▷ *incumbent is a cover: prune*
    $p_i \leftarrow 1/(n+1)$ for all i
  end if
  return $(c \oplus \mathrm{bits}_p(u),\ \textsc{true},\ c,\ c_f)$
end procedure

Potential. Off the optimum, $\Phi_z(s) = \big(1 + [\neg b] + (-F_z(x^*))/((n+1)(kn+1))\big)/3$. *Runtime.* None proved. The honest constants $\varepsilon_1 = \frac{1}{3(n+1)(kn+1)}$, $p_1 = (n+1)^{-n}$ were transcribed, and the speed-up certificate was reported false (odd-path parity trap: the only improving point of the even-vertex cover is its complement).

E.15 BANDWIDTHVC, CLAUDE OPUS 5.5 (STAGED; COMPLETE)

Two designs were tried in one run. *Attempt 1* (elitist anchor XOR an i.i.d. mask whose rate r resets to $\frac{1}{n+1}$ on a strict improvement and otherwise doubles, wrapping at $\frac{n}{n+1}$) passed all stages up to the drift constants; its speed-up certificate is false for the same parity trap, since the improving probability there is $(n+1)^{-n} < 2^{-n}$. After the stall-driven restart, *attempt 2* passed all stages. Let $L = 2k+1$ and $\ell = \min\{L, n\}$.

Algorithm 14 Window-repair random search (BANDWIDTHVC, Claude Opus 5.5, attempt 2).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$
  end if
  $y \leftarrow \mathrm{bits}_{1/2}(u);\ m \leftarrow \mathrm{bits}_{1/3}(u^{+n})$
  if $y_i = 1$ for all $i < \ell$ **then** ▷ *an optimum never chooses all of* $0, \ldots, 2k$
    $y_i \leftarrow c_i \oplus m_i$ for all $i < \ell$ ▷ *redraw the window near the incumbent*
  end if
  return $(y,\ \textsc{true},\ c,\ c_f)$
end procedure

Potential. With $F_z^{\mathrm{opt}} = \max_y F_z(y)$, $e(s) = q$ if $\neg b$ or $F_z(q) \geq F_z(x^*)$, else x^*, $\eta = 1/(4 \cdot 3^L)$ and $R = (n+1)kn + n$: off the optimum $\Phi_z(s) = (1-\eta) + \eta \min\{1, (F_z^{\mathrm{opt}} - F_z(e(s)))/(R+1)\}$. *Runtime.* $\varepsilon_1 = 1 - \eta$, $p_1 = 2^{-n}(1 + 3^{-L})$: $\mathbb{E}[T_A] \leq 4 \cdot 2^n / \big((1 - a/4)(1 + a)\big)$ with $a = 3^{-(2k+1)}$, i.e. $\Theta(2^n)$ — a constant-factor improvement over random search. Machine-checked; $\delta = a/4$, $n_0 = 1$.

E.16 SUBMODULARMIN, INTEGER-VALUED, CLAUDE OPUS 5.5 (STAGED; COMPLETE)

One CLI-backend session, with no pre-checking and no intervention (wandb run `breezy-sun-789`). In addition to the facts listed with the problem cards, the prompt for this card contains the following advice (verbatim apart from a corrected typo): "Note that this is a challenging problem domain where naive algorithmic designs, such as elitist hill-climbing with a single individual, are unlikely to succeed. Current efforts in designing efficient evolutionary algorithms for this problem are ongoing, so your design must be innovative and well-justified. You need to carefully consider the properties of submodular functions and how to exploit them in your algorithm design. Note that the minimisation [of] sub-modular functions is a well-studied problem in combinatorial optimisation which allows for polynomial time algorithms."

The state $(q, b, \lambda, k, g, f_{\mathrm{prev}}, t, \phi)$ holds the query q, the initialisation flag b, a point $\lambda \in [0,1]^n$, the position k in the current chain, the accumulated subgradient $g \in \mathbb{R}^n$, the previous chain value f_{prev}, the intended query t and a flag ϕ recording that an optimum has been seen. $S_k(\lambda)$ is the set of the k coordinates with the largest values of λ (ties broken by index), $\eta = 1/(nB^2)$, $m = \mathrm{bits}_{1/(n+1)}(u)$, and Π clamps each coordinate to $[0,1]$.

Algorithm 15 Projected subgradient descent on the Lovász extension (SUBMODULARMIN, Claude Opus 5.5).

procedure `init`(u)
  return $(S_0 \oplus m,\ \text{FALSE},\ (\frac{1}{2})^n,\ 0,\ 0^n,\ 0,\ S_0,\ \text{FALSE})$
end procedure
procedure `step`$(u,\ (q, b, \lambda, k, g, f_{\text{prev}}, t, \phi),\ f)$
  if not b **then**
    $t \leftarrow S_0(\lambda)$; $k \leftarrow 0$ ▷ *start the first chain*
  else if ϕ **then**
    keep querying t
  else if $f \geq 0$ **then**
    $t \leftarrow q$; $\phi \leftarrow$ TRUE ▷ *an optimum was queried*
  else if $q = t$ **then** ▷ *the intended level set was queried exactly*
    $g_i \leftarrow f_{\text{prev}} - f$ for the coordinate i added at position k ▷ *marginal value*
    if $k < n$ **then**
      $k \leftarrow k + 1$; $f_{\text{prev}} \leftarrow f$; $t \leftarrow S_k(\lambda)$
    else
      $\lambda \leftarrow \Pi(\lambda - \eta g)$; $k \leftarrow 0$; $g \leftarrow 0^n$; $t \leftarrow S_0(\lambda)$ ▷ *subgradient step*
    end if
  end if ▷ *otherwise the query was perturbed: repeat t*
  return $(t \oplus m,\ \text{TRUE},\ \lambda,\ k,\ g,\ f_{\text{prev}},\ t,\ \phi)$
end procedure

Potential. With y^* a fixed minimiser of f, $D(s) = \|\lambda - \mathbf{1}_{y^*}\|^2$ and $c = \eta/(2n+4)$: $\Phi(s) = 0$ if $F_f(q) \geq 0$; otherwise $\Phi(s) = c/(n+1)$ if ϕ, and $\Phi(s) = \big(\min\{D(s), n\} + c\,(2(n+1-k) - [q = t])\big)/(n+1)$ if not ($c(2n+3)$ replaces the second term before initialisation). The first term is the distance of λ to the minimiser, which drops by at least η per completed chain: if no level set of the chain is optimal, then by integrality the Lovász extension at λ is at least 1, and $\|g\|^2 \leq nB^2$. The second term counts the queries left in the current chain. ***Drift sets.*** $\mathrm{Good}_1(s)$ is the draws with an all-zero mask, so that the intended query is made exactly; $\mathrm{Jump}_1(s)$ is the draws whose mask turns the next query into y^*; $\mathrm{Worsen}_1 = \mathrm{BigWorsen}_1 = \emptyset$. ***Runtime.*** $\varepsilon_1 = 1/\big(nB^2(2n+4)(n+1)\big)$, $p_1 = \frac{1}{3}$, $p_{\text{jump}} = (n+1)^{-n}$: $\mathbb{E}\,[T_A] \leq 12\,nB^2(2n+4)(n+1) = O(n^3B^2) = O(n^{3+2c})$ for all $n \geq 3$ and every instance. Machine-checked. The algorithm is the classical projected subgradient method on the Lovász extension; the bound is polynomial for every fixed c and pseudo-polynomial in B, and it is weaker than the best known query bounds for submodular minimisation. 2,389 lines of Lean, 340k output tokens, 2.2 h. An earlier session on the real-valued card without the integrality facts (`submodular_minimisation`) did not get past Stage 10c.

Further generations (row 15 of Table 1). The same session continued the loop for four more generations, each seeded with the previous algorithm and bound. With $B = n^c$ the certified bounds are $12\,nB^2(2n+4)(n+1)$ (generation 1), $16\,n^2B^2(n+2)$ (2), $32\,nB^2(n+1)$ (3), $\frac{128}{15}B^2(n+1)^2$ (4) and $\frac{512}{63}B^2(n+1)^2$ (5), so the order improves from $O(n^{3+2c})$ to $O(n^{2+2c})$ at generation 3 and later generations improve the constant. All five generations are machine-checked with no `sorry` and only the standard axioms. Generation 5 has 2,901 lines of Lean, and the five generations used 1.1M output tokens in total, including the write-ups made between generations.

E.17 WELLCOVEREDVC, ANONYMISED CARD, CLAUDE FABLE 5.1 (STAGED; INCOMPLETE)

Clamp $\kappa(r) = \min\{\frac{1}{2n}, \max\{\frac{1}{4n}, r\}\}$ (applied only when sampling).

Algorithm 16 Polarity-adaptive asymmetric mutation (`wcvc`, Claude Fable 5.1).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0,\ \frac{1}{2n},\ \frac{1}{2n})$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*, r_\uparrow, r_\downarrow),\ f)$
  if not b **then**
    $(c, c_f) \leftarrow (q, f)$
  else if $f \geq f^*$ **then**
    if $|q|_1 > |x^*|_1$ **then**
      $(r_\uparrow, r_\downarrow) \leftarrow (2r_\uparrow, r_\downarrow/2)$
    else if $|q|_1 < |x^*|_1$ **then**
      $(r_\uparrow, r_\downarrow) \leftarrow (r_\uparrow/2, 2r_\downarrow)$
    end if
    $(c, c_f) \leftarrow (q, f)$
  else
    $(c, c_f) \leftarrow (x^*, f^*)$
  end if
  $p_i \leftarrow \kappa(r_\downarrow)$ if $c_i = 1$, else $\kappa(r_\uparrow)$, for all i
  return $(c \oplus \mathrm{bits}_p(u),\ \textsc{true},\ c,\ c_f,\ r_\uparrow,\ r_\downarrow)$
end procedure

Potential. With $F_z^{\mathrm{eff}}(s) = \max\{F_z(x^*), F_z(q)\}$ if b, else $F_z(q)$, and $R = (n+1)(n^2+1)+1$: off the optimum $\Phi_z(s) = \min\{1, (1 - F_z^{\mathrm{eff}}(s))/R\}$. ***Runtime.*** None proved. All stages up to the Worsen/Jump stage passed, with $\varepsilon_1 = 1/R$, $p_1 = \frac{1}{8n}$ and a machine-checked certificate ($\delta = 1$, $n_0 = 21$). The Good set had been frozen with an arbitrary fitter target rather than a single flip, so its mass bound could not be proved; the bound $32n\big((n+1)(n^2+1)+1\big) = O(n^4)$ claimed for a corrected Good set is unverified. A later resume re-verified only the draw-set measurability stage.

E.18 WellCoveredVC, anonymised card, Claude Opus 5.5 (staged; complete)

Algorithm 17 Regime-switching asymmetric mutation (`wcvc`, Claude Opus 5.5).

procedure `init`(u)
  return $(\mathrm{bits}_{1/2}(u),\ \textsc{false},\ 0^n,\ 0)$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$
  end if
  if $c_f \geq -n$ **then** ▷ *incumbent is a cover: shrink*
    $p_i \leftarrow \frac{1}{n+1}$ if $c_i = 1$, else $\frac{1}{(n+1)^2}$, for all i
  else ▷ *uncovered edges remain: grow*
    $p_i \leftarrow \frac{1}{(n+1)^2}$ if $c_i = 1$, else $\frac{1}{n+1}$, for all i
  end if
  return $(c \oplus \mathrm{bits}_p(u),\ \textsc{true},\ c,\ c_f)$
end procedure

Potential. With $c(s) = q$ if $\neg b$ or $F_z(q) \geq f^*$, else x^*, and $N = (n+1)\binom{n+1}{2} + n + 2$: off the optimum $\Phi_z(s) = (1 - F_z(c(s)))/N$. ***Runtime.*** $\varepsilon_1 = \frac{1}{N}$, $p_1 = \frac{1}{n+1}\big(\frac{n}{n+1}\big)^{n-1}$: $\mathbb{E}[T_A] \leq 4N(n+1)\big(\frac{n+1}{n}\big)^{n-1} \leq 4e(n+1)N = O(n^4)$ (leading term $\approx 2e\,n^4$). Machine-checked; $\delta = 1$, $n_0 = 23$. From the improvement stage on, candidates were pre-checked directly against the Lean server.

E.19 WELLCOVEREDVC, NAMED CARD, CLAUDE FABLE 5.1 (STAGED; COMPLETE)

Algorithm 18 Fitness-directed asymmetric mutation (WELLCOVEREDVC, Claude Fable 5.1).

procedure `init`(u)
  $q \leftarrow \mathrm{bits}_{1/2}(u)$; **return** $(q,\ \textsc{false},\ q,\ 0,\ \textsc{true})$
end procedure
procedure `step`$(u,\ (q, b, x^*, f^*, \gamma),\ f)$
  if b **and** $f < f^*$ **then**
    $(c, c_f) \leftarrow (x^*, f^*)$
  else
    $(c, c_f) \leftarrow (q, f)$
  end if
  $\gamma' \leftarrow [c_f < -n]$ ▷ *grow while the incumbent is not a cover*
  if γ' **then**
    $p_i \leftarrow \frac{1}{2(n+1)}$ if $c_i = 1$, else $\frac{1}{n+1}$, for all i
  else
    $p_i \leftarrow \frac{1}{n+1}$ if $c_i = 1$, else $\frac{1}{2(n+1)}$, for all i
  end if
  return $(c \oplus \mathrm{bits}_p(u),\ \textsc{true},\ c,\ c_f,\ \gamma')$
end procedure

Potential. Off the optimum $\Phi_z(s) = \big(1 + \min\{\ell_z(q), \ell_z(x^*)\}\big)/(n^2 + n + 2)$, where $\ell_z(x) = |x|_1 - m_z^*$ for a cover x and $\ell_z(x) = n + 1 + \mathrm{unc}_z(x)$ otherwise, with m_z^* the minimum cover size. *Runtime.* $\varepsilon_1 = \frac{1}{n^2+n+2}$, $p_1 = \frac{1}{(n+1)^2}$: $\mathbb{E}[T_A] \le 4(n^2+n+2)(n+1)^2 = O(n^4)$. Machine-checked; $\delta = 1$, $n_0 = 20$.

F PIPELINE PROGRESS AND PER-STAGE STATISTICS

Figure 4 shows how each run of Table 1 progressed through the stages of the pipeline, and Figure 5 where the runs spend their effort.

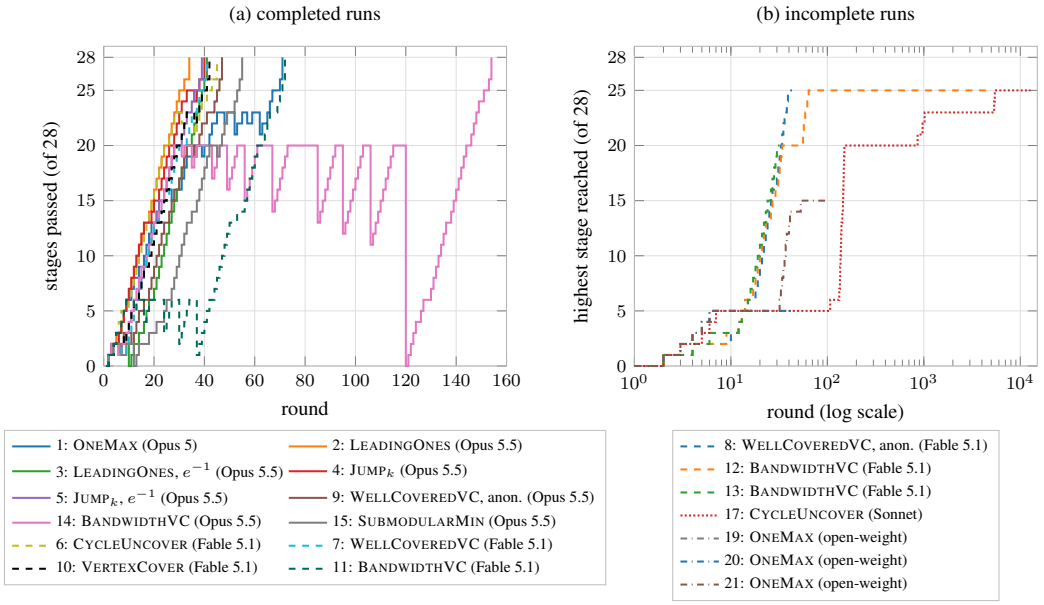


Figure 4: Progress through the 28 pipeline stages for the runs of Table 1, numbered as in the table, one point per round. (a) Runs that completed the pipeline: the number of stages passed, including the drops caused by backtracking; the final point at 28 marks completion. Row 14 includes a first attempt that stalled at Stage 10c before the pipeline restarted. (b) Runs that did not complete: the highest stage reached so far, on a logarithmic round axis (rows 12 and 17 continue for thousands of automatic rounds after their best level). The zero-shot runs (rows 16 and 18) have a single stage and are omitted.

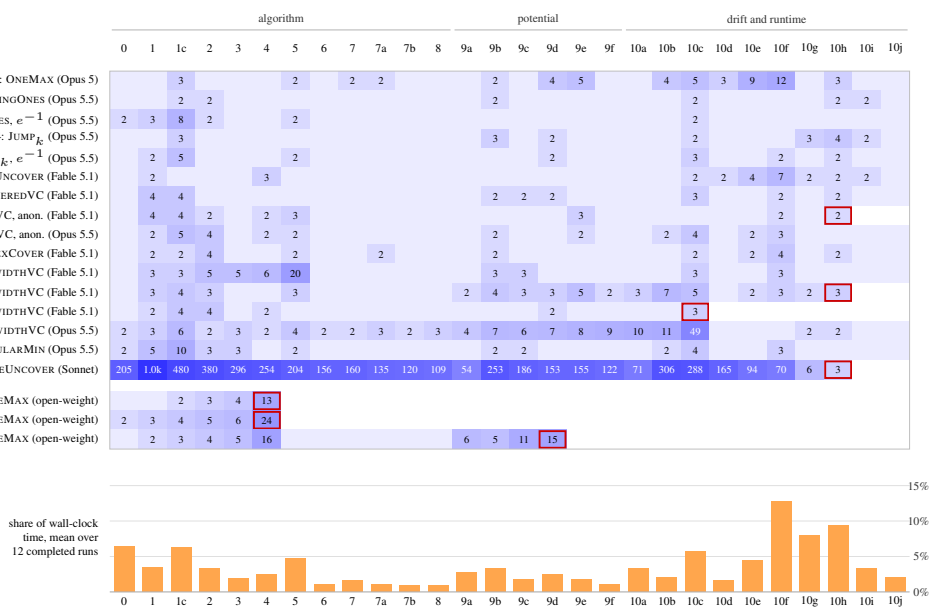


Figure 5: Per-stage statistics of the runs of Table 1, numbered as in the table. Top: number of attempts per stage (colour on a logarithmic scale; the number is shown when a stage needed more than one attempt); a red frame marks the stage at which an incomplete run stopped. For rows 19–21 (vLLM backend) one attempt is a batch of 256 samples. Bottom: share of wall-clock time spent in each stage, averaged over the 12 completed frontier-model runs. Each run is counted up to its first completion or, if it did not complete, up to its last round at its highest stage. The zero-shot runs (rows 16 and 18) have a single stage and are omitted.

Figure 5 shows where the runs spend their effort. Over the 12 completed frontier-model runs, the drift and runtime stages 10a–10j account for 53% of the wall-clock time, algorithm construction (Stages 0–8) for 34% and the potential (Stages 9a–9f) for 13%. The most expensive stages are the pointwise drift and mass proofs: `PositiveDrift` (10f, 13%), `DriftSetMass` (10h, 9%) and `NegativeDriftAndJump` (10g, 8%). Two stages fail at the first attempt in almost every run for technical rather than mathematical reasons: `DriftConstantProperties` (10c) failed first in all 12 completed runs (81 attempts and 13 backtracks in total), mostly on the normal form of the baseline constants, and `StepMeasurability` (1c) in 11 of them. The incomplete frontier-model runs stop at a drift stage: rows 8, 12 and 17 at the mass bound (10h), which is false for their designs, and row 13 at 10c. The open-weight models stop much earlier, at `NotRandomSearch` (Stage 4, rows 19 and 20) and `PotentialFineGrained` (Stage 9d, row 21), after 13–24 batches of 256 samples at that stage. The Sonnet run (row 17) needed between 54 and about 1,000 attempts at every stage it passed before Stage 10g.

G Positioning relative to related work

Table 1: All runs. Each row is one pipeline session driven by the named model. Rows 1–18 use the CLI backend, through which the model exchanges prompts and responses with the pipeline over stdin/stdout; rows 19–21 use the vLLM backend (Section B). "Staged" is the full pipeline of Table 2; "zero-shot" is the one-stage ablation of Section 4. Rows 16–21 are ablations: the zero-shot pipeline, or a weaker model; rows 19–21 use open-weight models run on HPC clusters (Section E.1). *Complete* means that every stage was machine-checked with no `sorry`; otherwise the last passed stage is given. *Order* is the order of the machine-checked bound on $\mathbb{E}[T_A]$, which holds for all $n \geq 3$ and every instance (for staged runs it is $4/(\varepsilon_1 p_1)$). *Tokens* is the number of output tokens (including reasoning) generated by the model that drove the run, from launch to the last accepted stage; n/r means not recorded. *Lean* is the number of lines of Lean code the model contributed to the accepted proof, excluding blank lines, comments, the prompt, the API and the problem card. LO is the fitness of the elitist reference point; the LEADINGONES card marked "e^{-1} lemma" additionally supplies the inequality $e^{-m/(r-1)} \leq (1-1/r)^m$. k is the bandwidth for BANDWIDTHVC and the gap width for JUMP$_k$, and n^c bounds the values of a SUBMODULARMIN instance. *#gen* is the number of generations of the pipeline used to obtain the result; for a run with several generations, the row reports the last generation only (Tokens and Lean are those of that generation). Pseudocode, potentials and constants of every run are given in Section E.

#	Problem	Model	Pipeline	#gen	Algorithm	Complete	Order	Tokens	Lean	App.
1	ONEMAX	Opus 5	staged	1	(1+1) EA, rate $\frac{1}{n+1}$	yes	$O(n^3)$	n/r	2,036	E.2
2	LEADINGONES	Opus 5.5	staged	1	fitness-dependent rate $\frac{1}{\mathrm{LO}+2}$	yes	$O(n^3)$	160k	1,262	E.3
3	LEADINGONES, e^{-1} lemma	Opus 5.5[§]	staged	1	fitness-dependent rate $\frac{1}{\mathrm{LO}+2}$	yes	$O(n^2)$	191k	1,553	E.4
4	JUMP$_k$	Opus 5.5	staged	1	(1+1) EA, rate $\frac{1}{n+1}$	yes	$O(n^{k+2})$	144k	1,086	E.5
5	JUMP$_k$, e^{-1} lemma	Opus 5.5	staged	3	rate cycling with a learned phase bit	yes	$O(n^{k+1})$	396k	1,335	E.6
6	CYCLEUNCOVER	Fable 5.1	staged	1	cycle walk (learn-then-solve)	yes	$O(n^2)$	538k	2,373	E.7
7	WELLCOVEREDVC	Fable 5.1	staged	1	fitness-directed asym. mutation	yes	$O(n^4)$	366k	1,756	E.19
8	WELLCOVEREDVC, anonymised card	Fable 5.1	staged	1	polarity-adaptive asym. mutation	no (10g)	–	275k	1,282	E.17
9	WELLCOVEREDVC, anonymised card	Opus 5.5[‡]	staged	1	regime-switching asym. mutation	yes	$O(n^4)$	245k	1,530	E.18
10	VERTEXCOVER, flat fitness	Fable 5.1	staged	1	edge probing + exact solve	yes	$O(n^3)$	400k	1,889	E.10
11	BANDWIDTHVC, flat fitness	Fable 5.1	staged	1	window probing + exact solve	yes	$O(kn)$	460k	1,658	E.11
12	BANDWIDTHVC	Fable 5.1	staged	1	direction-adaptive asym. mutation	no (10g)	–	402k	1,202	E.12
13	BANDWIDTHVC	Fable 5.1	staged	1	fill-then-prune mutation	no (10b)	–	192k	510	E.14
14	BANDWIDTHVC	Opus 5.5	staged	1	window-repair random search	yes	$\Theta(2^n)$	283k	1,784	E.15
15	SUBMODULARMIN, integer-valued	Opus 5.5	staged	5	Lovász subgradient descent	yes	$O(n^{2+2c})$	1.1M	2,901	E.16
16	CYCLEUNCOVER	Fable 5.1	zero-shot	1	random search	yes	$\Theta(2^n)$	41k	404	E.8
17	CYCLEUNCOVER	Sonnet[†]	staged	1	elitist bit-flip (5 designs)	no (10g)	–	n/r	1,146	E.9
18	BANDWIDTHVC	Fable 5.1	zero-shot	1	(1+1) EA, rate $\frac{1}{n}$	no	–	172k	112	E.13
19	ONEMAX	Goedel-Prover-V2-32B + Gemma 4[¶]	staged	1	per-bit learned bias (PBIL-like)	no (3)	–	25.1M	375	E.1
20	ONEMAX	Goedel-Prover-V2-32B + Gemma 4[¶]	staged	1	per-bit learned bias (PBIL-like)	no (3)	–	73.1M	316	E.1
21	ONEMAX	Goedel-Prover-V2-32B + Gemma 4[¶]	staged	1	random search (fixed biases $\frac{1}{2}$)	no (9c)[‖]	–	124.3M	725	E.1

[†] Model version not recorded in the artefacts. [‡] Candidates for the last five stages were pre-checked directly against the Lean server. [§] Candidates for the last three stages were pre-checked in the same way. [¶] Goedel-Prover-V2-32B for the formal stages (pass@256), Gemma 4 26B-A4B for the prose stages; earlier versions of the staged pipeline with the same 28 stages; tokens are those of all logged samples of the formal model. [‖] Passed Stage 4 only through a loophole in that pipeline version (see Section E.1). All models are Claude models.

Table 3: Open-weight runs on ONEMAX with the most recent pipeline versions. "Last passed" is the last stage that was machine-checked.

Run	Cluster	Started	Pipeline version	Hours	Last passed
`generous-dragon-779`	Sulis	2026-09-21	`optimisation`	23.8	3
`driven-blaze-773`	Sulis	2026-09-18	`optimisation`	23.8	1
`denim-paper-771`	Sulis	2026-09-17	`split_v9`	21.6	3
`super-moon-761`	Sulis	2026-09-15	`split_v9`	23.3	3
`bumbling-plant-742`	Sulis	2026-09-13	`split_v9`	23.2	3
`avid-waterfall-734`	Isambard-AI	2026-09-12	`split_v9`	23.6	3
`mild-dream-724`	Sulis	2026-09-08	`split_v9`	23.6	3
`absurd-puddle-723`	Isambard-AI	2026-09-08	`split_v9`	23.6	3
`peach-armadillo-718`	Isambard-AI	2026-09-06	`split_v9`	23.7	3
`serene-voice-715`	Sulis	2026-09-06	`split_v9`	23.8	1c
`visionary-snow-713`	Sulis	2026-09-05	`split_v8`	20.7	3
`pleasant-elevator-712`	Isambard-AI	2026-09-04	`split_v8`	23.7	1c
`golden-yogurt-711`	Isambard-AI	2026-09-02	`split_v8`	36.8	9c
`royal-snowflake-710`	Sulis	2026-09-01	`split_v8`	46.2	3

Table 4: Positioning relative to three adjacent literatures. The first group machine-checks analyses of *human-designed* algorithms. The second group produces machine-checked proofs, but of a theorem or specification *given* to it. The third group *discovers* algorithms, but validates them only empirically. This work lies at the intersection: the algorithm is proposed by an LLM, its analysis is produced by an LLM, and the property machine-checked is the expected runtime of a stochastic process.

Work	**Algorithmic artefact**	**Process**	**Designed by**	**Analysed by**	**Machine-checked property**
Li et al. (2024)	first-order convex methods	deterministic	human	human	convergence rate
Vajjha et al. (2021)	value / policy iteration	stochastic (MDP)	human	human	convergence
Schäffeler & Abdulaziz (2025)	approximate policy iteration	stochastic (MDP)	human	human	convergence
Ospanov et al. (2025); Varambally et al. (2025); Li et al. (2026)	— (proof / code search)	n/a	n/a	LLM	given theorem or code specification
Romera-Paredes et al. (2024); Novikov et al. (2025); Lange et al. (2025)	heuristics, constructions	—	LLM	—	none (empirical validation)
This work	black-box optimisation algorithm	**stochastic**	**LLM**	**LLM**	**correctness & expected runtime**