A Minimal Reimplementation of AlphaEvolve and FunSearch for Heuristic Discovery

Mehdi Soleimanifar
mehdi@caltech.edu

June 4, 2025

Abstract

In this document, we implement a basic version of advanced search algorithms that leverage frontier AI models for scientific and algorithmic discovery, including AlphaEvolve [NVE+25] and FunSearch [RPBN+24]. Applied to the online bin packing problem, our approach—using only modest compute (e.g., GPT-4.1-nano)—uncovers a novel heuristic that nearly matches the performance of the best heuristic identified in the FunSearch paper, while exhibiting superior generalization to other datasets.

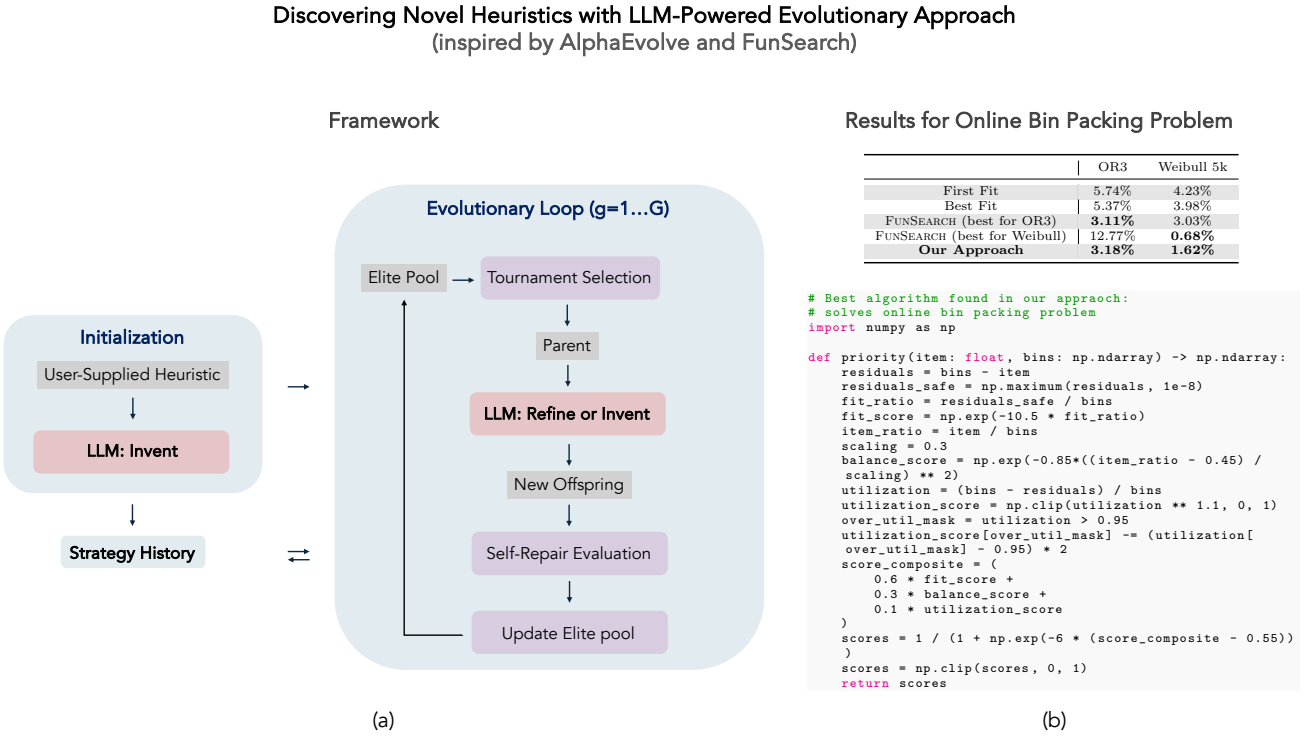

Figure 1: (a) We implement an evolutionary approach for discovering new heuristic algorithms for combinatorial problems. This approach is inspired by AlphaEvolve [NVE+25] and FunSearch [RPBN+24] and involves using LLMs to produce candidate algorithms or heuristics, which are then evaluated and evolved over multiple iterations. (b) Performance of online bin packing methods (top) and heuristic code found with our approach (bottom) using gpt-4.1-nano. Our heuristic achieves similar result to the one found by FunSearch [RPBN+24] in the OR3 (20 instances, 500 items each) and moreover, it exhibits better generalization when applied to Weibull-5k (5 instances, 5000 items each)

1 LLM-Guided Evolutionary Search for Discovery

1.1 Background

FunSearch. FunSearch [RPBN+24] leverages an evolutionary approach, iteratively mutating and recombining high-scoring code to explore the solution space and uncover non-trivial, domain-specific optimizations. These

code snippets are tested in a sandboxed environment, scored for performance, and the best solutions are fed back to the LLM to guide further refinements. This iterative process enabled FunSearch to discover novel heuristics that outperformed traditional first-fit and best-fit algorithms on benchmark bin-packing datasets.

AlphaEvolve. AlphaEvolve [NVE+25] generalizes FunSearch from single-function discovery to the evolution of *entire code bases*. The agent maintains a population of programs and applies two mutation modes:

1. *reinvention*, which generates a fresh implementation from scratch, and
2. *refinement*, which edits an existing parent program.

Programs are evaluated, ranked, and propagated through tournament selection, with elite individuals preserved across generations. AlphaEvolve demonstrated state-of-the-art algorithms for tasks ranging from matrix multiplication (surpassing Strassen) to data-center scheduling.

The key distinction between these approaches lies in their scope and sophistication. FunSearch establishes the proof-of-concept for LLM-driven evolutionary algorithm discovery but remains constrained to single-function optimization. AlphaEvolve extends this paradigm to full program evolution, incorporating more sophisticated mutation strategies and population management.

1.2 Problem Overview

What is bin packing? We address a classic combinatorial optimization problem called *online one-dimensional bin packing*. Items show up one at a time (this is what *online* means), each with a size between 0 and 1. As soon as an item arrives we must place it into a bin of capacity 1, and once placed it cannot be moved. The goal is simple: use as few bins as possible. In general this is an NP-hard problem with no general efficient algorithm. Hence, one aims for heuristic approaches which exhibit good performance for practical instances.

How does FunSearch evaluate the performance? We evaluate heuristic performance by comparing the number of bins used against the L_1 lower bound. For each dataset instance, the L_1 bound is simply computed as total item sizes divided by total bin capacity, representing the minimum bins needed if items could be packed with perfect efficiency. Each heuristic is applied to all instances in a dataset, and we record the average number of bins used across instances. Performance is then measured as the percentage excess above the L_1 bound:

$$\text{excess} = \frac{\text{average bins used} - \text{average } L_1 \text{ bound}}{\text{average } L_1 \text{ bound}} \times 100\%. \tag{1}$$

Lower excess indicates better performance, with 0% representing theoretically optimal packing. We report results on two standard FunSearch datasets: OR3 (20 instances, 500 items each) and Weibull-5k (5 instances, 5000 items each), allowing direct comparison with existing methods and quantifying how closely evolved heuristics approach the theoretical optimum.

Potential Caveats in FunSearch paper. A few of algorithmic choices in the FunSearch paper are, in our view, debatable and initially caused some confusion. Below, we summarize them:

- **Evaluation metric** (1) **is suboptimal.** The aggregation method used in the FunSearch paper averages the number of bins and the lower bounds separately before computing the percentage excess. This approach disproportionately weights larger instances and obscures instance-level performance. A more standard and interpretable alternative could be to compute the percentage excess for each instance individually and *then* average those values, ensuring equal weight for all instances regardless of size. For consistency with published results, we follow the original convention in our comparisons.
- **Total number of items is known.** Online bin packing algorithms can be implemented in two ways: opening new bins only when needed or alternatively, pre-allocating many bins upfront (which assumes knowing the total number of items in advance). While pre-allocation can improve performance by providing more placement options, it is somewhat in conflict with the online constraint where items arrive sequentially without advance knowledge. This implementation choice significantly impacts results—we observed difference in performance on identical datasets using the same heuristic but different bin management approaches.
- **Their optimal algorithms for OR3 and Weibull-5k are different.** The heuristics discovered by FunSearch are highly specialized to the data distribution they were trained on. For instance, the heuristic

yielding near-optimal performance ($< 1\%$ excess) on WEIBULL-5K performs significantly worse on OR3, with excess rising to over 12% in our simulations. Why does this happen? While this limits generalization, it highlights a deeper strength of FUNSEARCH: its ability to discover distribution-specific algorithms that exploit structural regularities in the data. In this sense, the most impressive feature is not the $< 1\%$ number itself, but that FUNSEARCH reliably identifies the *right* heuristic for each regime.

- **Excellent performance on Weibull-5k is partly due to concentration effects.** The WEIBULL-5K dataset exhibits strong concentration properties due to its sharply peaked distribution and large instance size (5 000 items), which enables simple heuristics (like preferring the emptiest feasible bin) to perform remarkably well. Such heuristics, however, fail to adapt to more heterogeneous or adversarial settings like OR3 as mentioned before.

1.3 Our Implementation

Our evolutionary algorithm maintains a fixed-size elite population of p individuals (parameter `pop_size`) and evolves over G generations. The algorithm combines tournament selection with dual mutation strategies to discover effective bin-packing heuristics through iterative refinement and invention.

Hyperparameters. Three hyperparameters govern the search:

- `pop_size` (p): number of elite individuals retained each generation
- `generations` (G): total evolutionary iterations (search horizon).
- `tournament_size` (k): number of elites sampled for parent selection; larger k increases selection pressure.

Initialization (generation $g = 0$). We seed the population with user-provided heuristics. The remaining slots are filled by prompting the LLM (`gpt-4.1-nano` in our experiments) to *invent* fresh heuristics. All p candidates are evaluated, and the best p (lowest mean excess-bin percentage) form the first elite set.

Evolution (generations $g = 1, \ldots, G$). At every generation we run a classic elitist $(\mu + \lambda)$ strategy with $\mu = \lambda = p$.

1. **Parent choice via tournament selection.** The loop runs p times. At each iteration we draw a *fresh* tournament of k distinct elites, pick the individual with the lowest excess-bin percentage as the parent, and (implicitly) return the sampled elites to the pool. Consequently the same high-performer can win several tournaments while some elites may spawn no offspring, producing a variable reproduction rate.
2. **Variation.** Given the parent, the LLM is prompted either to *refine* it (prob. 0.5) using the parent's code and score, or to *invent* a brand-new heuristic (prob. 0.5). The prompt is augmented with the accumulated `strategy_history` to minimise duplicate ideas.
3. **Self-repair evaluation.** The candidate heuristic is executed on every instance. If execution fails, the error trace is fed back to the LLM for up to three repair attempts; persistent failure incurs a penalty fitness of 10^6. Successful candidates are scored by the mean percentage excess over the analytic L_1 bound.

After all $\lambda = p$ offspring are evaluated, we concatenate them with the $\mu = p$ previous elites, sort by fitness, and retain the best p individuals for the next generation—an elitist $(\mu + \lambda)$ replacement that guarantees the best-so-far solution quality never degrades.

We note that changing the outer loop to run $\lambda \neq p$ iterations would give a general $(\mu + \lambda)$ strategy with independent control over the offspring count while leaving parent survival unchanged.

1.4 Results

Optimal heuristic found. To assess the effectiveness of our approach, we evaluated the performance of the generated heuristic. Figure 1 summarizes the comparative results alongside existing baselines, including traditional heuristics (First Fit, Best Fit) and the state-of-the-art FUNSEARCH system [RPBN$^+$24]. Using modest compute and using a minimal LLM gpt-4.1-nano, our method almost matches or outperforms FUNSEARCH on OR3, while demonstrating significantly better generalization to the more challenging WEIBULL-5K benchmark. The Python snippet below the table displays one of the best heuristics evolved by our system, illustrating the kind of interpretable, parameterized logic it discovers through guided search.

Key Evolutionary Dynamics. A closer analysis of the evolutionary process reveals several key dynamics that contributed to the discovery of high-performing heuristics. The most successful heuristic, which achieved a 3.18% excess bin, was not a spontaneous discovery but the result of iterative refinement over multiple generations. Its lineage began with a `seed` heuristic from `(gen=0, round=3)` with a score of 4.27%, which was improved by a `refine` mutation to score 3.88% (`(gen=2, round=7)`), and finally refined again to achieve the best score (`(gen=6, round=9)`). This pathway, `(gen=0, round=3) [4.27%]` → `(gen=2, round=7) [3.88%]` → `(gen=6, round=9) [3.18%]`, is a powerful illustration of the algorithm's ability to build upon and incrementally improve promising solutions.

The `refine` operator was the primary driver of high performance; in fact, every top-tier heuristic with a score below 4.0% was generated through refinement. Interestingly, once the champion heuristic (`(gen=6, round=9)`) was discovered, it became a prolific parent for many subsequent strong contenders due to the elitism mechanism. However, these "child" heuristics did not always outperform their parent (e.g., `(gen=8, round=4)` scored 3.75%), highlighting that refinement acts as a targeted exploration around an elite solution, which, when combined with elitism, creates a robust search that can explore variations without losing the best solution found so far.

1.5 Comparison to the original frameworks

Similarities. Our code preserves the two key AlphaEvolve ideas: (i) explicit reinvention vs. refinement prompts and (ii) population-based evolution with elitism. Like FunSearch, evaluation is done by direct execution and scoring of candidate code.

Simplifications.

(1) *No code-base graph.* The full AlphaEvolve represents a program as a dependency graph of multiple files and permits fine-grained edits; we treat the heuristic as a single Python function.

(2) *Greedy credit assignment.* Fitness is the *sole* evaluation signal; we do not incorporate the auxiliary *novelty* or *critic* rewards used by AlphaEvolve to encourage exploration.

(3) *Fixed prompt engineering.* We rely on the initial system prompt and error feedback only, whereas AlphaEvolve dynamically chains thoughts or uses a reflection LLM to suggest edits.

(4) *Small population and horizon.* FunSearch typically runs for hundreds of iterations and AlphaEvolve for thousands; our toy run uses $(G, P) = (10, 14)$ to stay within local compute budgets.

Take-aways. The experiment demonstrates that *even a few dozen* LLM calls, a simple tournament EA, and lightweight runtime checks suffice to reproduce the core *invent/refine* dynamics of AlphaEvolve on a standard FunSearch task. Scaling population size, generations, and the richness of mutation operators are promising next steps towards closing the performance gap with the original DeepMind systems.

References

[NVE+25] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[RPBN+24] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.