For this paper, Google DeepMind posted a short (and easily digestible) description of their results on their blog. We advise to read this first (see first link below) and then perhaps look up especially interesting parts in paper included on the following pages. For the referenced supplementary material, refer to the full publication online (see second link below), but this is probably too much information anyway.

Blog post:
Alhussein Fawzi and Bernardino Romera Paredes.
FunSearch: Making new discoveries in mathematical sciences using Large Language Models.
Google DeepMind Blog, 2023.
https://deepmind.google/discover/blog/funsearch-making-new-discoveries-in-mathematical-sciences-using-large-language-models/

Full publication including supplementary material:
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, Alhussein Fawzi.
Mathematical discoveries from program search with large language models.
Google DeepMind, authors' version of work accepted in Nature, 2023.
https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/funsearch-making-new-discoveries-in-mathematical-sciences-using-large-language-models/Mathematical-discoveries-from-program-search-with-large-language-models.pdf

Mathematical discoveries from program search with large language models

Bernardino Romera-Paredes[1*] Mohammadamin Barekatain[1*]
Alexander Novikov[1*] Matej Balog[1*] M. Pawan Kumar[1*]
Emilien Dupont[1*] Francisco J. R. Ruiz[1*] Jordan S. Ellenberg[2]
Pengming Wang[1] Omar Fawzi[3] Pushmeet Kohli[1] Alhussein Fawzi[1*]

[1]Google DeepMind, London, UK
[2]University of Wisconsin-Madison, Madison, Wisconsin, USA
[3]Université de Lyon (Inria, ENS Lyon, UCBL, LIP), Lyon, France

Abstract

Large Language Models (LLMs) have demonstrated tremendous capabilities in solving complex tasks, from quantitative reasoning to understanding natural language. However, LLMs sometimes suffer from confabulations (or hallucinations) which can result in them making plausible but incorrect statements (Bang et al., 2023; Borji, 2023). This hinders the use of current large models in scientific discovery. Here we introduce *FunSearch* (short for *search*ing in the *fun*ction space), an evolutionary procedure based on pairing a pre-trained LLM with a systematic evaluator. We demonstrate the effectiveness of this approach to surpass the best known results in important problems, pushing the boundary of existing LLM-based approaches (Lehman et al., 2022). Applying *FunSearch* to a central problem in extremal combinatorics — the cap set problem — we discover new constructions of large cap sets going beyond the best known ones, both in finite dimensional and asymptotic cases. This represents the first discoveries made for established open problems using LLMs. We showcase the generality of *FunSearch* by applying it to an algorithmic problem, online bin packing, finding new heuristics that improve upon widely used baselines. In contrast to most computer search approaches, *FunSearch* searches for programs that describe *how* to solve a problem, rather than *what* the solution is. Beyond being an effective and scalable strategy, discovered programs tend to be more interpretable than raw solutions, enabling feedback loops between domain experts and *FunSearch*, and the deployment of such programs in real-world applications.

Many problems in mathematical sciences are "easy to evaluate," despite being typically "hard to solve." For example, in computer science, NP-complete optimization problems admit a polynomial-time evaluation procedure (measuring the quality of the solution), despite the widespread belief that no polynomial-time algorithms to *solve* such problems exist. We focus in this paper on problems admitting an efficient `evaluate` function, which measures the quality of a candidate solution. Prominent examples include the maximum independent set problem and maximum constraint satisfaction problems (such as finding the ground state energy of a Hamiltonian). Our goal is to generate a `solve` program, such that its outputs receive high scores from `evaluate` (when executed on inputs of interest), and ultimately improve over the best known solutions.

*Equal contributors.

While Large Language Models (LLMs) have recently seen dramatic improvements in their coding capabilities [5–9], with applications including debugging [10, 11], solving code competitions [12, 13] and improving code performance [14], synthesizing `solve` programs for *open* problems requires finding *new* ideas that are verifiably correct. This is very hard for LLMs, as they tend to confabulate or ultimately fall short of going beyond existing results. To surpass the "nominal" capabilities of LLMs, recent works [3] have combined them with evolutionary algorithms [15, 16], leading to important improvements on diverse synthetic problems [17], searching for neural network architectures [18–20], and solving puzzles [21]. Our proposed method, *FunSearch*, pushes the boundary of LLM-guided evolutionary procedures to a new level: the discovery of new scientific results for established open problems, and the discovery of new algorithms. Surpassing state-of-the-art results on established open problems provides a clear indication that the discoveries are truly new, as opposed to being retrieved from the LLM's training data.

FunSearch (short for *search*ing in the *fun*ction space) combines a pre-trained (frozen) Large Language Model, whose goal is to provide creative solutions, with an evaluator, which guards against confabulations and incorrect ideas. *FunSearch* iterates over these two components, evolving initial low-scoring programs into high-scoring ones discovering new knowledge. Key to the success of this simple procedure is a combination of multiple essential ingredients. First, we sample best performing programs and feed them back into prompts for the LLM to improve on; we refer to this as best-shot prompting. Second, we start with a program in the form of a *skeleton* (containing boilerplate code and potentially prior structure about the problem), and only evolve the part governing the critical program logic. For example, by setting a greedy program skeleton, we evolve a *priority* function used to make decisions at every step. Third, we maintain a large pool of diverse programs by using an island-based evolutionary method that encourages exploration and avoids local optima. Finally, leveraging the highly parallel nature of *FunSearch*, we scale it asynchronously, considerably broadening the scope of this approach to find new results, while keeping the overall cost of experiments low.

We show the surprising effectiveness of *FunSearch* on several use-cases. We consider a fundamental problem in extremal combinatorics, namely, the cap set problem [22, 23]. *FunSearch* demonstrates the existence of hitherto unknown constructions that go beyond existing ones, including the largest improvement in 20 years to the asymptotic lower bound. To the best of our knowledge, this shows the first scientific discovery — a new piece of verifiable knowledge about a notorious scientific problem — using an LLM. Using *FunSearch*, we also find new algorithms for the online bin packing problem that improve upon traditional ones on well-studied distributions of interest [24, 25], with potential applications to improving job scheduling algorithms.

While most computer search techniques output directly what the solution is (e.g., a list of vectors forming a cap set), *FunSearch* produces programs generating the solution. For structured problems, such programs tend to be more interpretable — facilitating interactions with domain experts — and concise — making it possible to scale to large instances — compared to a mere enumeration of the solution. In addition, decision procedures (such as for bin packing) described by code in a standard programming language are crucially easier to deploy compared to other types of descriptions (e.g., neural networks), which typically require specialized hardware and for which verifying design specifications is notoriously hard.

1 *FunSearch*

An overview of *FunSearch* is shown in Figure 1, and its components are described in more detail below. For more details and ablations showing the importance of each component, see Methods and

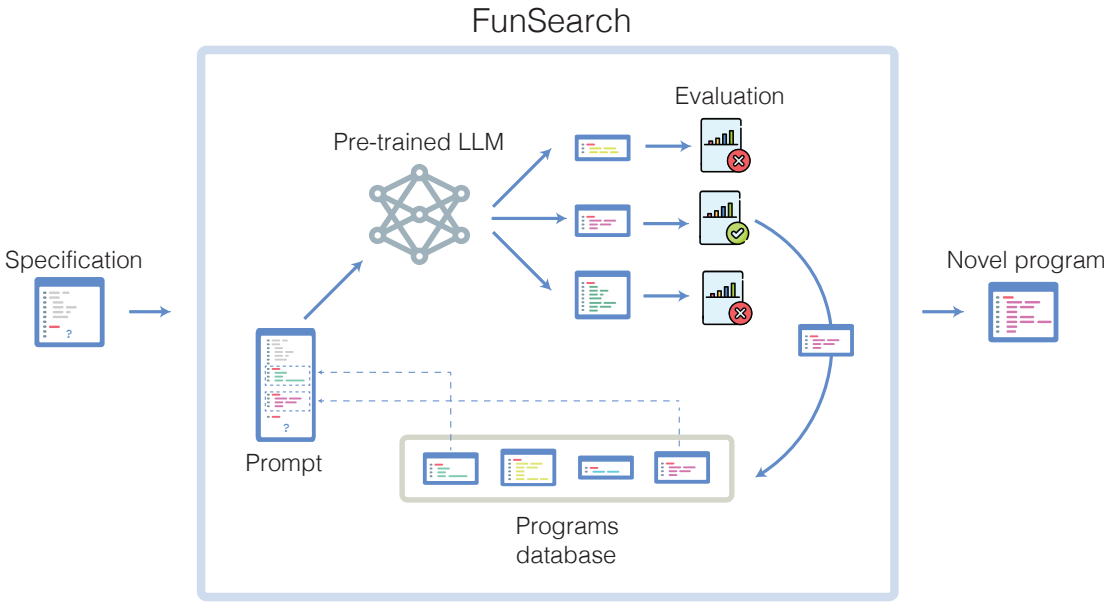


Figure 1: Overview of *FunSearch*. The input to *FunSearch* is a specification of the problem in the form of an `evaluate` function, an initial implementation of the function to evolve, which can be trivial, and potentially a skeleton. At each iteration, *FunSearch* builds a prompt by combining several programs sampled from the programs database (favouring high-scoring ones). The prompt is then fed to the pre-trained LLM, and new programs are created. Newly created programs are then scored and stored in the programs database (if correct), thus closing the loop. The user can at any point retrieve the highest-scoring programs discovered so far.

Appendix A in Supplementary Information.

Specification. The input to *FunSearch* is a specification of the problem in the form of an `evaluate` function, which scores candidate solutions. In addition, we provide an initial program (which can be trivial) to evolve. While in principle these are the minimum requirements, we found that performance tends to improve significantly if we write the initial `solve` program in the form of a skeleton (containing boilerplate code and prior knowledge of the problem in the form of a program structure), and only use *FunSearch* to evolve the critical part that governs its logic. Figure 2 (a) shows an example where the skeleton takes the form of a simple greedy algorithm, and the crucial part to evolve by *FunSearch* is the `priority` function that is used to make the greedy decision at every step. This delegates to *FunSearch* precisely the part that is usually the hardest to come up with. While a fixed skeleton may constrain the space of programs that can be discovered, we find it improves overall results because it focuses the LLM resources on evolving the critical part only, instead of also using the LLM to recreate already known program structures (with more opportunities for mistakes that would render the entire program incorrect). If available, the user can optionally provide additional known information about the problem at hand, in the form of docstrings, relevant primitive functions, or import packages, which *FunSearch* may use.

```python
"""Finds large cap sets."""
import numpy as np
import utils_capset

# Function to be executed by FunSearch.
def main(n):
  """Runs `solve` on `n`-dimensional cap set and
  ↪  evaluates the output."""
  solution = solve(n)
  return evaluate(solution, n)

def evaluate(candidate_set, n):
  """Returns size of candidate_set if it is a cap
  ↪  set, None otherwise."""
  if utils_capset.is_capset(candidate_set, n):
    return len(candidate_set)
  else:
    return None

def solve(n):
  """Builds a cap set of dimension `n` using
  ↪  `priority` function."""
  # Precompute all priority scores.
  elements = utils_capset.get_all_elements(n)
  scores = [priority(el, n) for el in elements]
  # Sort elements according to the scores.
  elements = elements[np.argsort(scores,
  ↪  kind='stable')[::-1]]

  # Build `capset` greedily, using scores for
  ↪  prioritization.
  capset = []
  for element in elements:
    if utils_capset.can_be_added(element, capset):
      capset.append(element)
  return capset

# Function to be evolved by FunSearch.
def priority(element, n):
  """Returns the priority with which we want to add
  ↪  `element` to the cap set."""
  return 0.0
```

(a) Cap set.

```python
"""Finds good assignment for online 1d bin
↪  packing."""
import numpy as np
import utils_packing

# Function to be executed by FunSearch.
def main(problem):
  """Runs `solve` on online 1d bin packing instance,
  ↪  and evaluates the output."""
  bins = problem.bins
  # Packs `problem.items` into `bins` online.
  for item in problem.items:
    # Extract bins that have space to fit item.
    valid_bin_indices =
    ↪  utils_packing.get_valid_bin_indices(item,
    ↪  bins)
    best_index = solve(item,
    ↪  bins[valid_bin_indices])
    # Add item to the selected bin.
    bins[valid_bin_indices[best_index]] -= item
  return evaluate(bins, problem)

def evaluate(bins, problem):
  """Returns the negative of the number of bins
  ↪  required to pack items in `problem`."""
  if utils_packing.is_valid_packing(bins, problem):
      return -utils_packing.count_used_bins(bins,
      ↪  problem)
  else:
    return None

def solve(item, bins):
  """Selects the bin with the highest value according
  ↪  to `heuristic`."""
    scores = heuristic(item, bins)
    return np.argmax(scores)

# Function to be evolved by FunSearch.
def heuristic(item, bins):
  """Returns priority with which we want to add
  ↪  `item` to each bin."""
  return -(bins - item)
```

(b) Online bin packing.

Figure 2: Examples of *FunSearch* specifications for two problems. The `evaluate` function takes as input a candidate solution to the problem, and returns a score assessing it. The `solve` function contains the algorithm skeleton, which calls the function to evolve that contains the crucial logic. For (a), the function to evolve is called `priority`, and for (b) it is called `heuristic`. The `main` function implements the evaluation procedure by connecting the pieces together. Specifically, it uses the `solve` function to solve the problem, and then scores the resulting solutions using `evaluate`. In simplest cases, `main` just executes `solve` once and uses `evaluate` to score the output, e.g., see (a). In specific settings such as online algorithms, the `main` function implements some additional logic, e.g., see (b).

Pre-trained LLM. The LLM is the creative core of *FunSearch*, in charge of coming up with improvements to the functions presented in the prompt and sending these for evaluation. Perhaps surprisingly, we obtain our results with a pre-trained model, i.e., without any fine-tuning on our problems. We use Codey, an LLM built on top of the PaLM2 model family [26], which has been finetuned on a large corpus of code and is publicly accessible through its API [27]. Because *FunSearch* relies on sampling from an LLM extensively, an important performance-defining tradeoff is between

the quality of the samples and the inference speed of the LLM. In practice, we have chosen to work with a fast-inference model (rather than slower-inference, higher-quality), and the results in the paper are obtained using a total number of samples on the order of 10^6. Beyond this tradeoff, we have empirically observed that the results obtained in this paper are not too sensitive to the exact choice of LLM, as long as it has been trained on a large enough corpus of code. See Appendix A in Supplementary Information for a comparison to StarCoder [7], a state-of-the-art open-source LLM for code.

Evaluation. Programs generated by the LLM are evaluated and scored on a set of inputs. For example, in the cap set problem (Section 2.1) the inputs are the values of the dimensionality n that we are interested in, and in combinatorial optimization (Section 2.2), the inputs correspond to different bin packing instances. The scores across different inputs are then combined into an overall score of the program using an aggregation function, such as the mean. The scored programs are then sent to the programs database. Programs that were incorrect (did not execute within the imposed time and memory limits, or produced invalid outputs) are discarded, and the remaining scored programs are then sent to the programs database.

Programs database. The programs database keeps a population of correct programs, which are then sampled to create prompts. Preserving and encouraging diversity of programs in the database is crucial to enable exploration and avoid being stuck in local optima. To encourage diversity we adopt an islands model, also known as multiple population and multiple-deme model [28, 29], a genetic algorithm approach. A number of islands, or subpopulations, are created and evolved independently. To sample from the program database, we first sample an island and then sample a program within that island, favoring higher-scoring and shorter programs (see Methods for the exact mechanism). Crucially, we let information flow between the islands by periodically discarding the programs in the worst half of the islands (corresponding to the ones whose best individuals have the lowest scores). We replace the programs in those islands with a new population, initialized by cloning one of the best individuals from the surviving islands.

Prompt. New prompts are created by "best-shot prompting" from the programs database, and are then fed to the LLM to generate a new program. We first sample k programs from a single island in the programs database, according to the procedure described above. Sampled programs are then sorted according to their score, and a version is assigned to each (`v0` for the lowest scoring program, `v1` for the second lowest scoring, etc.). These programs are then combined into a single prompt — with the version appended as a suffix to the function name; e.g., in the case of Figure 2 (a), this would be `priority_v0, priority_v1, ...` — and the header of the function we wish to generate (e.g., `priority_vk`) is added to the end of the prompt. In practice, we set $k = 2$, as two functions lead to better results compared to just one, with diminishing returns beyond that. Constructing a prompt by combining *several* programs (as opposed to only one) enables the LLM to spot patterns across the different programs and generalize those. Related approaches to prompt building have been recently considered; e.g., [17], and were shown to perform well on different domains.

Distributed approach. We implement *FunSearch* as a distributed system that has three types of workers: a *programs database*, *samplers*, and *evaluators*, which communicate *asynchronously*. The programs database stores and serves programs, samplers generate new functions using the pre-trained LLM, while evaluators assess programs, as shown in Figure F.26 in Supplementary Information. In the example of Figure 2 (a), the programs database stores `priority` functions, samplers generate

new implementations of `priority`, while evaluators score the proposals by executing the `main` function on user-specified inputs. Our distributed system offers several advantages: first, it naturally leverages parallelism across different tasks, e.g., LLM sampling and evaluation are performed concurrently. Second, it enables scaling to more than one sampler and evaluator, which would be a very limiting setup, considering that evaluation can take minutes for many problems of interest. Running evaluators in parallel considerably broadens the scope of this approach to such problems. The distributed setting enables running many evaluator nodes on inexpensive CPU hardware, while few samplers run on machines with accelerators for fast LLM inference; this keeps the overall cost and energy usage of experiments low. In our experiments, we typically use 15 samplers and 150 CPU evaluators (can be served on 5 CPU servers each running 32 evaluators in parallel). See Appendix A in Supplementary Information for more details. Also, due to the randomness of LLM sampling and of the evolutionary procedure, for some problems we run several experiments to get the best reported results. See Methods and Appendix A.3 in Supplementary Information for a full statistical analysis.

2 Results

We now describe some of the new discoveries made by *FunSearch* in two different fields: pure mathematics and applied computer science. Additional discoveries on other problems (namely, corners problem and Shannon capacity of cycle graphs) are presented in Appendix B in Supplementary Information. Full discovered programs are available in Appendix C in Supplementary Information.

2.1 Extremal combinatorics

We apply *FunSearch* to two related problems in extremal combinatorics — a branch of mathematics that studies the maximal (or minimal) possible sizes of sets satisfying certain properties.

Cap sets. The cap set problem [22], once described by Terence Tao as "perhaps my favourite open question" [30], refers to the task of finding the largest possible set of vectors in $\mathbb{Z}_3^n$ (known as a cap set) such that no three vectors sum to zero. Geometrically, no three points of a cap set lie on a line (see Figure 3 for an example with $n = 2$).

The problem has drawn much interest for a variety of reasons. For one, it is an analogue of the classical number theory problem of finding large subsets of primes in which no three are in arithmetic progression. For another, it differs from many problems in combinatorics in that there is no consensus among mathematicians regarding what the right answer should be. Finally, the problem serves as a model for the many other problems involving "three-way interactions." For instance, progress towards improved upper bounds for the cap set problem [31, 32] immediately led to a series of other combinatorial results, e.g., on the Erdös-Radio sunflower problem [33].

The exact size of the largest possible cap set in n dimensions is known only for $n \leq 6$. A brute force approach is not practical as the search space quickly becomes enormous with growing n, e.g., around 3^{1600} for $n = 8$. Previous methods impose potentially suboptimal restrictions on the search space [34, 35]. In contrast, we search the full space via an algorithm skeleton that utilises a function $\texttt{priority} : \mathbb{Z}_3^n \to \mathbb{R}$. Intuitively, this function provides a priority with which each $x \in \mathbb{Z}_3^n$ should be included in the cap set. Our algorithm starts with an empty set and iteratively adds the vector $x \in \mathbb{Z}_3^n$ with the highest priority that does not violate the cap set constraint; see Figure 2 (a). Starting from a trivial constant function, we evolve the crucial `priority` component of our approach to result in large cap sets.

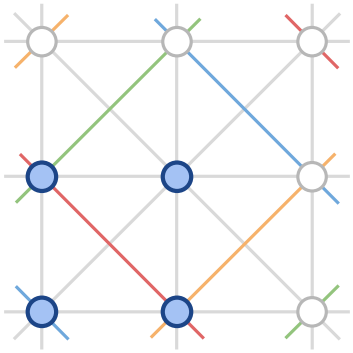


Figure 3: Diagram of a cap set of size 4 in $\mathbb{Z}_3^2$. The circles are the elements of $\mathbb{Z}_3^2$ with the ones belonging to the cap set shown in blue. The possible lines in $\mathbb{Z}_3^2$ are also shown (with colors indicating lines that wrap around in arithmetic modulo 3). No three elements of the cap set are in a line.

Using this approach we discovered cap sets of sizes shown in Figure 4 (a). Notably, in dimension $n = 8$, *FunSearch* found a larger cap set than what was previously known, thus illustrating the power of *FunSearch* to discover novel constructions. This also shows the scalability of *FunSearch* to larger dimensions, where the previously best known construction relied on a complex combination of cap sets in lower dimensions [34, 35]. In contrast, *FunSearch* discovered a larger cap set from scratch, without having to be explicitly taught any way of combining cap sets. Moreover, we do not just discover the set of 512 8-dimensional vectors in itself, but a program that generates it: we show this program in Figure 4 (b). Through inspecting the code, we obtain a degree of *understanding* of what this set is: specifically, manual simplification of Figure 4 (b) provides the construction in Figure 4 (c). Some properties of this construction are strikingly similar to the construction of the Hill cap [36, 37], which results in the optimal 112-cap in $\mathbb{Z}_3^6$.

Admissible sets. Beyond finding the size of the largest cap set c_n in dimension n, a fundamental problem in additive combinatorics [23] is determining the capacity $C = \sup_n c_n^{1/n}$. The breakthrough result of [32] established an upper bound of $C \leq 2.756$. In this work, we are interested in lower bounds on C. To this end, we use the framework of *constant weight admissible sets* (or admissible sets for short) [35], which has established the current state-of-the-art.

Formally, admissible sets $\mathcal{A}(n, w)$ are collections of vectors in $\{0, 1, 2\}^n$ satisfying two properties: i) each vector has the same number w of non-zero elements but a unique support (thereby implying $|A| \leq \binom{n}{w}$); ii) for any three distinct vectors there is a coordinate in which their three respective values are $\{0, 1, 2\}$, $\{0, 0, 1\}$, or $\{0, 0, 2\}$. Informally, an admissible set describes how to combine cap sets in smaller dimensions into large cap sets in higher dimensions [35]. We denote the set of full-size admissible sets (with $|A| = \binom{n}{w}$) as $\mathcal{I}(n, w)$. The current state-of-the-art [39] has relied on SAT solvers to construct large admissible sets.

As before, we evolve a function `priority` : $\{0, 1, 2\}^n \to \mathbb{R}$, which is used to iteratively grow admissible sets. Starting from a trivial constant function, we discover one that provides us with an $\mathcal{I}(12, 7)$ admissible set; the discovered program is shown in Figure 5 (b). This discovery alone already improves the lower bound on the cap set capacity from 2.2180 [39] to 2.2184. Yet, interpreting the program found by *FunSearch* (Figure 5 b) helps us significantly push the boundaries of what admissible sets we can construct. Specifically, we notice that the discovered `priority` function treats the n coordinates in a highly symmetric way, and indeed it turns out that the admissible set

n	3	4	5	6	7	8
Best known	9	20	45	112	236	496
FunSearch	9	20	45	112	236	**512**

(a)

```python
def priority(el: tuple[int, ...],
↪  n: int) -> float:
  score = n
  in_el = 0
  el_count = el.count(0)

  if el_count == 0:
    score += n ** 2
    if el[1] == el[-1]:
      score *= 1.5
    if el[2] == el[-2]:
      score *= 1.5
    if el[3] == el[-3]:
      score *= 1.5
  else:
    if el[1] == el[-1]:
      score *= 0.5
    if el[2] == el[-2]:
      score *= 0.5

  for e in el:
    if e == 0:
      if in_el == 0:
        score *= n * 0.5
      elif in_el == el_count - 1:
        score *= 0.5
      else:
        score *= n * 0.5 ** in_el
      in_el += 1
    else:
      score += 1

  if el[1] == el[-1]:
    score *= 1.5
  if el[2] == el[-2]:
    score *= 1.5

  return score
```

(b)

```python
def build_512_cap() -> list[tuple[int, ...]]:
  """Returns a cap set of size 512 in `n=8` dimensions."""
  n = 8
  V = np.array(list(itertools.product(range(3), repeat=n)), dtype=np.int32)
  support = lambda v: tuple(i for i in range(n) if v[i] != 0)
  reflections = lambda v: sum(1 for i in range(1, n // 2) if v[i] == v[-i])

  # Add all 128 weight-8 vectors that have >= 2 reflections.
  weight8_vectors = [v for v in V
                     if np.count_nonzero(v) == 8  # Weight is 8.
                     and reflections(v) >= 2]  # At least 2 reflections.

  # Add all 128 weight-4 vectors that have specific support.
  supports_16 = [(0, 1, 2, 3), (0, 1, 2, 5), (0, 3, 6, 7), (0, 5, 6, 7),
                 (1, 3, 4, 6), (1, 4, 5, 6), (2, 3, 4, 7), (2, 4, 5, 7)]
  weight4_vectors = [v for v in V
                     if support(v) in supports_16]

  # Add all 128 weight-4 vectors with specific support and 1 reflection.
  supports_8 = [(0, 1, 2, 7), (0, 1, 2, 6), (0, 1, 3, 7), (0, 1, 6, 7),
                (0, 1, 5, 7), (0, 2, 3, 6), (0, 2, 6, 7), (0, 2, 5, 6),
                (1, 2, 4, 7), (1, 2, 4, 6), (1, 3, 4, 7), (1, 4, 6, 7),
                (1, 4, 5, 7), (2, 3, 4, 6), (2, 4, 6, 7), (2, 4, 5, 6)]
  weight4_vectors_2 = [v for v in V
                       if support(v) in supports_8
                       and reflections(v) == 1]  # Exactly 1 reflection.

  # Add 128 weight-5 vectors with <= 1 reflections and one more condition.
  allowed_zeros = [(0, 4, 7), (0, 2, 4), (0, 1, 4), (0, 4, 6),
                   (1, 2, 6), (2, 6, 7), (1, 2, 7), (1, 6, 7)]
  weight5_vectors = [
      v for v in V
      if tuple(i for i in range(n) if v[i] == 0) in allowed_zeros
      and reflections(v) <= 1  # At most 1 reflection.
      and (v[1] * v[7]) % 3 != 1 and (v[2] * v[6]) % 3 != 1]

  return weight8_vectors + weight4_vectors + weight4_vectors_2 +
  ↪  weight5_vectors
```

(c)

Figure 4: Result of applying *FunSearch* to the cap set problem. **(a)** Size of the largest cap set in $\mathbb{Z}_3^n$ for different dimensions n. **(b)** The function `priority` : $\mathbb{Z}_3^n \to \mathbb{R}$ discovered by *FunSearch* that results in a cap set of size 512 in $n = 8$ dimensions. One feature to note is that the priority is affected by whether the same entry appears in positions `i` and `-i` (`-i` denotes the `i`-th position counting from the end). This motivates the notion of `reflections`, used in (c). **(c)** An explicit construction of this new 512-cap, which we were able to manually construct thanks to having discovered the cap set by searching in function space. See Appendix E.2 in Supplementary Information for more details and for relation to Hill cap.

it constructs is preserved under independent cyclic permutations of coordinates within four disjoint groups of coordinate triples. Hereinafter we call such admissible sets *symmetric* (see Appendix D in Supplementary Information for a formal definition).

Bound on C	Admissible set ingredient	Source
2.2101	$\mathcal{I}(90, 89)$	(Calderbank and Fishburn, 1994)
2.2173	$\mathcal{I}(10, 5)$	(Edel, 2004)
2.2180	$\mathcal{I}(11, 7)$	(Tyrrell, 2022)
2.2184	$\mathcal{I}(12, 7)$	*FunSearch*
2.2194	$\mathcal{I}(15, 10)$	*FunSearch*
2.2202	$\mathcal{A}(24, 17)$	*FunSearch*

(a)

```python
def priority(el: tuple[int, ...], n: int, w: int) -> float:
  score = 0.0
  for i in range(n):
    if el[i] == 1:
      score -= 0.9 ** ( i % 4 )
    if el[i] == 2:
      score -= 0.98 ** (30 - ( i % 4 ))
    if el[i] == 1 and el[i - 4] == 1:
      score -= 0.98 ** (30 - ( i % 4 ))
    if el[i] == 2 and el[i - 4] != 0:
      score -= 0.98 ** (30 - ( i % 4 ))
    if el[i] == 2 and el[i - 4] == 1 and el[i - 8] == 2:
      score -= 0.98 ** (30 - ( i % 4 ))
      score -= 6.3
    if el[i] == 2 and el[i - 4] == 2 and el[i - 8] == 1:
      score -= 0.98 ** (30 - ( i % 4 ))
    if el[i] == 2 and el[i - 4] == 1 and el[i - 8] == 1:
      score -= 6.3
    if el[i] == 2 and el[i - 4] == 0 and el[i - 8] == 2:
      score -= 6.3
    if el[i] == 1 and el[i - 4] == 1 and el[i - 8] == 0:
      score -= 2.2
  return score
```

(b)

Figure 5: Results on the cap set problem via admissible sets. **(a)** Summary of lower bounds on the cap set capacity C. **(b)** The priority function $\{0, 1, 2\}^n \to \mathbb{R}$ discovered by *FunSearch* that results in an $\mathcal{I}(12, 7)$ admissible set. The source code reveals that when $n = 12$, the function treats the four triples of coordinates $\{0, 4, 8\}$, $\{1, 5, 9\}$, $\{2, 6, 10\}$, and $\{3, 7, 11\}$ together. We then checked that the admissible set is in fact symmetric under independent cyclic permutations of coordinates within each of these four triples. See Appendix D and Appendix E.3 in Supplementary Information for more details.

We now use *FunSearch* to directly search for symmetric admissible sets. Note that this is a more restricted but also much smaller search space, which allows for significantly higher dimensions and weights than were previously possible. This led us to discovering a full-size $\mathcal{I}(15, 10)$ admissible set (implying $C \geq 2.219486$) and a partial admissible set in $\mathcal{A}(24, 17)$ of size 237 984, which implies a new lower bound on the cap set capacity of 2.2202 (see Figure 5 a). While this is the largest improvement to the lower bound in the last 20 years, we note it is still far from the upper bound, and we hope our results inspire future work on this problem.

Not only does *FunSearch* scale to much larger instances than traditional combinatorial solvers (see Appendix A.4 in Supplementary Information), it is a unique feature of searching in function space that we were able to inspect the code discovered by *FunSearch* and infer a new insight into the problem, in the form of a new symmetry. The procedure we followed in this section is a concrete example of how LLM-based approaches can be used in mathematical sciences: *FunSearch* suggests a solution, which is examined by researchers, who may note features of interest. These features are used to refine the search, leading to better solutions. This process can be iterated, with both human and search consistently in the loop.

2.2 Bin packing

Combinatorial optimization is a subfield of mathematics which plays an important role across a wide range of areas, from theoretical computer science to practical problems in logistics and scheduling.

	OR1	OR2	OR3	OR4	Weibull 5k	Weibull 10k	Weibull 100k
First Fit	6.42%	6.45%	5.74%	5.23%	4.23%	4.20%	4.00%
Best Fit	5.81%	6.06%	5.37%	4.94%	3.98%	3.90%	3.79%
FunSearch	**5.30%**	**4.19%**	**3.11%**	**2.47%**	**0.68%**	**0.32%**	**0.03%**

Table 1: Fraction of excess bins (lower is better) for various bin packing heuristics on the OR and Weibull datasets. *FunSearch* outperforms first fit and best fit across problems and instance sizes.

While many combinatorial optimization problems are provably hard to solve for large instances, it is typically possible to achieve strong performance using *heuristics* to guide the search algorithm. The choice of a heuristic is crucial for obtaining strong performance, but designing a good heuristic is difficult in practice. In this section, we show that *FunSearch* can be used to discover effective heuristics for one of the central problems in combinatorial optimization: bin packing [4].

The goal of bin packing is to pack a set of items of various sizes into the smallest number of fixed-sized bins. Bin packing finds applications in many areas, from cutting materials to scheduling jobs on compute clusters. We focus on the *online* setting where we pack an item as soon as it is received (as opposed to the offline setting where we have access to all items in advance). Solving online bin packing problems then requires designing a heuristic for deciding which bin to assign an incoming item to.

Heuristics for online bin packing are well studied and several variants exist with strong worst case performance [40–45]. However, they often exhibit poor performance in practice [4]. Instead, the most commonly used heuristics for bin packing are *first fit* and *best fit*. First fit places the incoming item in the first bin with enough available space, while best fit places the item in the bin with least available space where the item still fits. Here, we show that *FunSearch* discovers better heuristics than first fit and best fit on simulated data.

To achieve this, we define a heuristic as a program that takes as input an item and an array of bins (containing the remaining capacity of each bin) and returns a priority score for each bin. The `solve` function picks the bin with the highest score according to the heuristic (see Figure 2 b). *FunSearch* is then used to evolve this heuristic, starting from best fit.

We first evaluate *FunSearch* on the well-known OR-Library bin packing benchmarks [24], consisting of four datasets, OR1 to OR4, containing bin packing instances with an increasing number of items (see Appendix E.4 in Supplementary Information for details). We evolve our heuristic on a training set of generated bin packing instances with the same number of items as those in OR1 and, after the evolutionary process is concluded, test it on the OR1 to OR4 datasets. We measure performance as the fraction of excess bins used over the L_2 lower bound [46] of the optimal offline packing solution (which is generally not achievable in the online setting).

As can be seen in Table 1, *FunSearch* outperforms both first fit and best fit across all datasets. Further, the learned heuristic generalizes: even though it has only seen instances of the same size as OR1 during training, it generalizes across problem sizes, performing even better on large instances and widening the gap to best fit. In addition to the OR benchmarks, we also use *FunSearch* to evolve heuristics on bin packing instances sampled from a Weibull distribution, as these closely follow many real-world scheduling problems [25, 47] (see Appendix E.4 in Supplementary Information for details). As shown in Table 1, the performance of *FunSearch* is very strong on this dataset, significantly outperforming first fit and best fit across instances, as well as scaling gracefully to large instances (being only 0.03% off the lower bound on the optimum for 100 000 items). In addition, *FunSearch* is

```python
def heuristic(item: float, bins: np.ndarray) -> np.ndarray:
  """Online bin packing heuristic discovered with FunSearch."""
  score = 1000 * np.ones(bins.shape)
  # Penalize bins with large capacities.
  score -= bins * (bins - item)
  # Extract index of bin with best fit.
  index = np.argmin(bins)
  # Scale score of best fit bin by item size.
  score[index] *= item
  # Penalize best fit bin if fit is not tight.
  score[index] -= (bins[index] - item)**4
  return score
```

Figure 6: Example of a short online bin packing heuristic discovered by *FunSearch* for the OR dataset. This example illustrates frequently observed behavior: instead of always packing items into the best fit bin, the heuristic encourages packing the item only if the fit is tight (line 11). Comments in the code were manually added. See Appendix C in Supplementary Information for more discovered heuristics.

robust and consistently outperforms these baselines as shown in the statistical analysis in Appendix A.3 in Supplementary Information.

We observed that several heuristics discovered by *FunSearch* use the same general strategy for bin packing (see Figure 6 for an example). Instead of packing items into bins with the least capacity (like best fit), the *FunSearch* heuristics assign items to least capacity bins only if the fit is very tight after placing the item. Otherwise, the item is typically placed in another bin which would leave more space after the item is placed. This strategy avoids leaving small gaps in bins that are unlikely to ever be filled (see Appendix E.5 in Supplementary Information for example visualizations of such packings).

As this example demonstrates, the benefits of *FunSearch* extend beyond theoretical and mathematical results to practical problems like bin packing. Indeed, bin packing, and related combinatorial optimization problems, are ubiquitous and find applications across a range of industries. We are optimistic that *FunSearch* could be applied to several such use-cases with potential for real-world impact.

3 Discussion

The effectiveness of *FunSearch* in discovering new knowledge for hard problems might seem intriguing. We believe that the LLM used within *FunSearch* does not use much context about the problem; the LLM should instead be seen as a source of diverse (syntactically correct) programs with occasionally interesting ideas. When further constrained to operate on the crucial part of the algorithm with a program skeleton, the LLM provides suggestions that marginally improve over existing ones in the population, which ultimately results in discovering new knowledge on open problems when combined with the evolutionary algorithm. Another crucial component of the effectiveness of *FunSearch* is that it operates in the space of programs: rather than directly searching for constructions (which is typically an enormous list of numbers), *FunSearch* searches for *programs* generating those constructions. Because most problems we care about are structured (highly non-random), we hypothesize that solutions are described more concisely with a computer program, compared to other representations. For example, the trivial representation of the admissible set $\mathcal{A}(24, 17)$ consists of more than 200 000 vectors, but the program generating this set consists only of a few lines of code. Because *FunSearch* implicitly encourages *concise* programs, it scales to much larger instances com-

pared to traditional search approaches in structured problems. In a loose sense, *FunSearch* attempts to find solutions that have low Kolmogorov complexity [48–50] (which is the length of the shortest computer program that produces a given object as output), while traditional search procedures have a very different inductive bias. We believe that such Kolmogorov-compressed inductive bias is key to *FunSearch* scaling up to the large instances in our use-cases. In addition to scale, we have empirically observed that *FunSearch* outputs programs that tend to be interpretable — that is, they are clearly easier to read and understand compared to a list of numbers. For example, by scrutinizing *FunSearch*'s output for the admissible set problem, we found a new symmetry, which was then subsequently used to improve the results even further. Despite the rarity of symmetric solutions, we observe that *FunSearch* preferred symmetric ones, as these are more parsimonious (that is, they require less information to specify), in addition to the natural bias of LLMs (trained on human-produced code) in outputting code with similar traits to human code. This is in contrast to traditional genetic programming which do not have this bias (and in addition require hand-tuning the mutation operators [51]).

We note that *FunSearch* currently works best for problems having the following characteristics: a) availability of an efficient evaluator; b) a "rich" scoring feedback quantifying the improvements (as opposed to a binary signal); c) ability to provide a skeleton with an isolated part to be evolved. For example, the problem of generating proofs for theorems [52–54] falls outside this scope, since it is unclear how to provide a rich enough scoring signal. In contrast, for MAX-SAT, the number of satisfied clauses can be used as a scoring signal. In this paper, we have explicitly striven for simplicity and we are confident that *FunSearch* can be further extended to improve its performance and be applicable to more classes of problems. In addition, the rapid development of LLMs is likely to result in samples of far superior quality at a fraction of the cost, making *FunSearch* more effective at tackling a broad range of problems. As a result, we envision that automatically-tailored algorithms will soon become common practice and deployed in real-world applications.

References

[1] Y. Bang, S. Cahyawijaya, N. Lee, W. Dai, D. Su, B. Wilie, H. Lovenia, Z. Ji, T. Yu, W. Chung, et al., A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity, arXiv preprint arXiv:2302.04023 (2023).

[2] A. Borji, A categorical archive of ChatGPT failures, arXiv preprint arXiv:2302.03494 (2023).

[3] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, K. O. Stanley, Evolution through large models, arXiv preprint arXiv:2206.08896 (2022).

[4] E. G. Coffman, M. R. Garey, D. S. Johnson, Approximation algorithms for bin-packing—an updated survey, Algorithm design for computer system design (1984) 49–106.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., Evaluating large language models trained on code, arXiv preprint arXiv:2107.03374 (2021).

[6] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., Program synthesis with large language models, arXiv preprint arXiv:2108.07732 (2021).

[7] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al., StarCoder: may the source be with you!, arXiv preprint arXiv:2305.06161 (2023).

[8] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, M. Lewis, Incoder: A generative model for code infilling and synthesis, in: International Conference on Learning Representations, 2022.

[9] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, C. Xiong, CodeGen: An open large language model for code with multi-turn program synthesis, in: International Conference on Learning Representations, 2022.

[10] X. Chen, M. Lin, N. Schärli, D. Zhou, Teaching large language models to self-debug, arXiv preprint arXiv:2304.05128 (2023).

[11] V. Liventsev, A. Grishina, A. Härmä, L. Moonen, Fully autonomous programming with large language models, arXiv preprint arXiv:2304.10423 (2023).

[12] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al., Competition-level code generation with alphacode, Science 378 (2022) 1092–1097.

[13] E. Zelikman, Q. Huang, G. Poesia, N. D. Goodman, N. Haber, Parsel: A (de-) compositional framework for algorithmic reasoning with language models, arXiv preprint arXiv:2212.10561 (2023).

[14] A. Madaan, A. Shypula, U. Alon, M. Hashemi, P. Ranganathan, Y. Yang, G. Neubig, A. Yazdanbakhsh, Learning performance-improving code edits, arXiv preprint arXiv:2302.07867 (2023).

[15] D. E. Goldberg, Optimization and machine learning, 1989.

[16] J. R. Koza, Genetic programming as a means for programming computers by natural selection, Statistics and computing 4 (1994) 87–112.

[17] E. Meyerson, M. J. Nelson, H. Bradley, A. Moradi, A. K. Hoover, J. Lehman, Language model crossover: Variation through few-shot prompting, arXiv preprint arXiv:2302.12170 (2023).

[18] A. Chen, D. M. Dohan, D. R. So, EvoPrompting: Language models for code-level neural architecture search, arXiv preprint arXiv:2302.14838 (2023).

[19] M. Zheng, X. Su, S. You, F. Wang, C. Qian, C. Xu, S. Albanie, Can GPT-4 perform neural architecture search?, arXiv preprint arXiv:2304.10970 (2023).

[20] M. U. Nasir, S. Earle, J. Togelius, S. James, C. Cleghorn, LLMatic: Neural architecture search via large language models and quality-diversity optimization, arXiv preprint arXiv:2306.01102 (2023).

[21] P. Haluptzok, M. Bowers, A. T. Kalai, Language models can teach themselves to program better (2022).

[22] J. Grochow, New applications of the polynomial method: the cap set conjecture and beyond, Bulletin of the American Mathematical Society 56 (2019) 29–64.

[23] T. Tao, V. H. Vu, Additive combinatorics, volume 105, Cambridge University Press, 2006.

[24] J. E. Beasley, Or-library: distributing test problems by electronic mail, Journal of the operational research society 41 (1990) 1069–1072.

[25] I. Castiñeiras, M. De Cauwer, B. O'Sullivan, Weibull-based benchmarks for bin packing, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2012, pp. 207–222.

[26] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen, et al., Palm 2 technical report, arXiv preprint arXiv:2305.10403 (2023).

[27] Code models overview, https://cloud.google.com/vertex-ai/docs/generative-ai/code/code-models-overview, 2023. [Online; accessed July-2023].

[28] R. Tanese, Distributed genetic algorithms for function optimization, University of Michigan, 1989.

[29] E. Cantú-Paz, A survey of parallel genetic algorithms, Calculateurs paralleles, reseaux et systems repartis 10 (1998) 141–171.

[30] T. Tao, Open question: best bounds for cap sets, https://terrytao.wordpress.com/2007/02/23/open-question-best-bounds-for-cap-sets/, 2009.

[31] E. Croot, V. F. Lev, P. P. Pach, Progression-free sets in are exponentially small, Annals of Mathematics (2017) 331–337.

[32] J. S. Ellenberg, D. Gijswijt, On large subsets of F_q^n with no three-term arithmetic progression, Annals of Mathematics (2017) 339–343.

[33] E. Naslund, W. Sawin, Upper bounds for sunflower-free sets, in: Forum of Mathematics, Sigma, volume 5, Cambridge University Press, 2017, p. e15.

[34] Y. Edel, J. Bierbrauer, Large caps in small spaces, Designs, Codes and Cryptography 23 (2001) 197–212.

[35] Y. Edel, Extensions of generalized product caps, Designs, Codes and Cryptography 31 (2004) 5–14.

[36] R. Hill, On the largest size of cap in $S_{5,3}$, Atti della Accademia Nazionale dei Lincei. Classe di Scienze Fisiche, Matematiche e Naturali. Rendiconti 54 (1973) 378–384.

[37] P. J. Cameron, J. H. Van Lint, Designs, graphs, codes and their links, volume 3, Cambridge University Press Cambridge, 1991.

[38] A. R. Calderbank, P. C. Fishburn, Maximal three-independent subsets of {0, 1, 2} n, Designs, Codes and Cryptography 4 (1994) 203–211.

[39] F. Tyrrell, New lower bounds for cap sets, arXiv preprint arXiv:2209.10045 (2022).

[40] C. C. Lee, D. T. Lee, A simple on-line bin-packing algorithm, Journal of the ACM (JACM) 32 (1985) 562–572.

[41] P. Ramanan, D. J. Brown, C.-C. Lee, D.-T. Lee, On-line bin packing in linear time, Journal of Algorithms 10 (1989) 305–326.

[42] S. S. Seiden, On the online bin packing problem, Journal of the ACM (JACM) 49 (2002) 640–671.

[43] J. Balogh, J. Békési, G. Dósa, J. Sgall, R. v. Stee, The optimal absolute ratio for online bin packing, in: Proceedings of the twenty-sixth annual ACM-SIAM symposium on discrete algorithms, SIAM, 2014, pp. 1425–1438.

[44] J. Balogh, J. Békési, G. Dósa, L. Epstein, A. Levin, A new and improved algorithm for online bin packing, in: 26th Annual European Symposium on Algorithms (ESA 2018), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 5:1–5:14.

[45] E. G. Coffman, J. Csirik, G. Galambos, S. Martello, D. Vigo, Bin packing approximation algorithms: survey and classification, Handbook of combinatorial optimization (2013) 455–531.

[46] S. Martello, P. Toth, Lower bounds and reduction procedures for the bin packing problem, Discrete applied mathematics 28 (1990) 59–70.

[47] S. Angelopoulos, S. Kamali, K. Shadkami, Online bin packing with predictions 36 (2022) 4574–4580.

[48] G. J. Chaitin, On the length of programs for computing finite binary sequences, Journal of the ACM (JACM) 13 (1966) 547–569.

[49] M. Li, P. Vitányi, et al., An introduction to Kolmogorov complexity and its applications, volume 3, Springer, 2008.

[50] R. J. Solomonoff, A formal theory of inductive inference. part i, Information and control 7 (1964) 1–22.

[51] M. O'Neill, L. Vanneschi, S. Gustafson, W. Banzhaf, Open issues in genetic programming, Genetic Programming and Evolvable Machines 11 (2010) 339–363.

[52] S. Polu, I. Sutskever, Generative language modeling for automated theorem proving, arXiv preprint arXiv:2009.03393 (2020).

[53] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, I. Sutskever, Formal mathematics statement curriculum learning, arXiv preprint arXiv:2202.01344 (2022).

[54] A. Q. Jiang, W. Li, S. Tworkowski, K. Czechowski, T. Odrzygóźdź, P. Miłoś, Y. Wu, M. Jamnik, Thor: Wielding hammers to integrate language models and automated theorem provers, Advances in Neural Information Processing Systems 35 (2022) 8360–8373.

A Methods

A.1 Implementation details of *FunSearch*

Distributed system. We implement *FunSearch* as a distributed system that has three types of workers: a *programs database*, *samplers*, and *evaluators*. The programs database stores the initial user-provided program, as well as all programs received from the evaluators. The samplers are in charge of performing the LLM inference step; to do so they repeatedly query the programs database for prompts. To achieve higher sampling throughput, samplers generate multiple samples from each prompt. The samples from the LLM (i.e., the generated programs) are sent to the evaluators, which score programs by executing them on inputs of interest and assessing the outputs using `evaluate`. Programs that are correct are sent to the programs database to be stored. Each of the three *FunSearch* components is provided as both Python code and pseudocode (Appendix F in Supplementary Information).

Prompt building. When queried for a prompt, the programs database samples k programs to encourage the LLM to merge ideas from them (we typically set $k = 2$; see Appendix E.1 in Supplementary Information). Programs are sorted according to their score in increasing order, starting from "version 0" (`v0`). Using these k programs, the prompt is built as explained next.

For the sake of clarity, we use here the problem specification from Figure 2 (a) to precisely describe the prompting mechanism. The overall structure of the prompt mimics the structure of the program skeleton, with the following differences: (i) The `priority` function is stripped out, and replaced with the $k = 2$ programs sampled, first `priority_v0` and then `priority_v1`. (ii) After that, a `priority_v2` function with no body is appended — the LLM will be in charge of completing the body of that function. (iii) All other functions that appear before `priority_v0` are removed. See Extended Data Figure 1 for an example of the structure of a prompt.

Evolutionary method and program selection. Another key feature of *FunSearch* is the method used for evolution of the population of programs from the programs database, as well as for program selection, i.e., how the programs database samples programs when queried for a prompt. For this, we use the islands model, a parallel genetic algorithm [28, 29]. Specifically, we split the population into m separate groups, or islands. Each island is initialized with a copy of the user-provided initial program and is evolved separately. That is, whenever a prompt is required, we first uniformly sample an island and then sample $k = 2$ programs from that island to build the prompt. The programs generated from the LLM based on that prompt will later be stored in the same island. Every four hours, we discard all the programs from the $m/2$ islands whose best instances have the lowest score. Each of these islands is then seeded with a single program, obtained by first choosing one of the surviving $m/2$ islands uniformly at random, and then retrieving the highest-scoring program from that island (breaking ties in favour of older programs). The evolutionary process is then restarted from this state, in which the reset islands contain one high-performing program each (see Extended Data Figure 2).

This method has several advantages. First, drawing the analogy where an island corresponds to an experiment, this approach effectively allows us to run several smaller experiments in parallel, instead of a single large experiment. This is beneficial because single experiments can get stuck in local minima, where the majority of programs in the population are not easily mutated and combined into stronger programs. The multiple island approach allows us to bypass this and effectively kill off such experiments to make space for new ones starting from more promising programs. Second,

promising experiments are run for longer, since the islands that survive a reset are the ones with higher scores.

Within each island, we further cluster programs according to their *signature*. We define the signature of a program as the tuple containing the program's scores on each of the inputs (e.g., the cap set size for each input n). Programs with the same signature are clustered together. When sampling a program within an island, we first sample an island's cluster, and then a program within that cluster (see Extended Data Figure 3). This approach, which aims at preserving diversity [55, 56], is related to Lexicase [57] in that both approaches consider a set of test cases for scoring an individual, and it is related to fitness uniform optimization [58], which also clusters individuals based on their fitness value, however we sample the clusters based on their score instead of uniformly, as detailed next.

When sampling a cluster, we favor those with larger score values. Specifically, let s_i denote the score of the i-th cluster, defined as an aggregation (e.g., mean) of all the scores in the signature that characterizes that cluster. The probability p_i of choosing cluster i is

$$p_i = \frac{\exp\left(s_i / T_{\text{cluster}}\right)}{\sum_{i'} \exp\left(s_{i'} / T_{\text{cluster}}\right)}, \quad T_{\text{cluster}} = T_0 \cdot \left(1 - \frac{n \bmod N}{N}\right), \tag{1}$$

where T_{cluster} is the temperature parameter, n is the current number of programs in the island, and T_0 and N are hyperparameters (given in Appendix E.1 in Supplementary Information). This approach is sometimes referred to as the Boltzmann selection procedure [59].

When sampling a program within a cluster, we favor shorter programs. In particular, let ℓ_i denote the negative length of the i-th program within the chosen cluster (measured as the number of characters), and let $\widetilde{\ell}_i = \frac{\ell_i - \min_{i'} \ell_{i'}}{\max_{i'} \ell_{i'} + 10^{-6}}$. We set the probability of each program proportional to $\exp(\widetilde{\ell}_i / T_{\text{program}})$, where T_{program} is a temperature hyperparameter.

Robustness. Due to randomness in LLM sampling and in the evolutionary procedure, repeating an experiment can lead to different results. For some problems (e.g. cap set through the admissible set problem, and online bin packing) every single run of *FunSearch* surpasses the baseline, with only some variation in the magnitude of the difference. For example, all experiments on admissible sets improve upon the previous best capacity lower bound, with 60% of experiments on $\mathcal{I}(12, 7)$ finding a full-size admissible set. For other problems, multiple independent repetitions of an experiment may be necessary to improve upon prior best results. In particular, the case of cap set by direct construction in $n = 8$ dimensions is particularly challenging, with only 4 out of 140 experiments discovering a cap set of size 512. See Appendix A.3 in Supplementary Information for more details.

A.2 Related work

Large Language Models. The rise of powerful LLMs such as [60] has been followed by systems in which an LLM core is enveloped by a "programmatic scaffold" [61], and multiple LLM calls are connected together in some way to accomplish larger and more intricate tasks beyond what would be possible using a single prompt and the raw LLM, possibly using external tools or external memory streams [62–66]. LLMs have also been paired with evaluators; for example, [21, 67] fine-tune an LLM on data that has been previously generated by the LLM itself (respectively on puzzle problems and solutions, and on justifications/explanations for answers to questions), and use an evaluator to assess the correctness of this data, ensuring that the fine-tuning dataset contains correct solutions/explanations only. More related to our approach is the use of LLMs as a mutation operator on code. [3] was the first work to show that coupling an LLM with a programatic way of scoring a

solution can lead to a self-improvement loop. In [17–20], the LLM is used as a crossover operator rather than a mutation one, i.e., the LLM prompts are composed of several functions, similarly to *FunSearch*. In [3, 17], the task is to improve code that generates bidimensional virtual robots that can move as far as possible in a given simulated terrain ([17] additionally considers the tasks of symbolic regression, natural language sentences, and image generation), in [18–20] the task is to find neural network architectures (described with Python code), and in [68] the task is continuous exploration in the game of Minecraft. In contrast, in this paper we tackle open problems in mathematics and algorithm design, and we surpass human-designed constructions. We achieve that by combining multiple ingredients together: a distributed system with multiple samplers and evaluators that communicate asynchronously, a user-provided program specification and skeleton, as well as an evolutionary mechanism based on islands that preserves the diversity of programs. *FunSearch* achieves that using an off-the-shelf LLM without fine-tuning.

More broadly, LLMs have been used for program synthesis as one of its main applications [5–9]. There are many use cases being explored, such as automatically editing code to improve performance [14], automatically debugging code [10, 11], generating code from natural language descriptions [69–71], and doing so to solve problems in code competitions [12, 13]. Unlike the above approaches which provide tools to increase the productivity of software engineers, we combine in this paper the creativity of LLMs with the power of evolutionary procedures to push the boundaries of human knowledge through solving open hard problems. Another line of research uses LLMs to guide the search for formal proofs for automatic theorem proving [52–54]. While this approach has the potential of eventually finding new knowledge, the achievements of these methods still lag behind the frontier of human knowledge.

Genetic programming. Genetic programming (GP) is a subfield of computer science concerned with automatically generating or discovering computer programs using evolutionary methods [16, 72, 73] and is employed for symbolic regression applications [74, 75] and discovery of optimization algorithms [76] among others. In this broad sense, combining LLMs with evolution can be seen as an instance of GP with the LLM acting as a mutation and crossover operator. However, using an LLM mitigates several issues in traditional GP [51], as shown in Appendix A in Supplementary Information and discussed in [3]. Indeed, GP methods require defining a number of parameters, chief among them the set of allowed mutation operations (or primitives) [16]. Designing such a set of operations is non-trivial and problem-specific, requiring domain knowledge about the problem at hand or its plausible solution [51]. While research has been done to mitigate this limitation, through for example the reuse of subprograms [77] or modeling the distribution of high-performing programs [78], designing effective and general code mutation operators remains difficult. In contrast, LLMs have been trained on vast amounts of code and as such have learned about common patterns and routines from human-designed code. The LLM can leverage this, as well as the context given in the prompt, to generate more effective suggestions than the random ones typically used in GP.

Related to GP, the field of hyper-heuristics [79, 80] seeks to design learning methods for generating heuristics applied to combinatorial optimization problems. In practice, these heuristics are often programs discovered through GP, typically by evolving a heuristic on a set of instances of a given combinatorial optimization problem, such as bin packing [81]. Indeed, like *FunSearch*, hyper-heuristics have also been applied to online bin packing, with the learned heuristics able to match the performance of first fit [82] and best fit [83] on a set of generated bin packing instances. Augmenting the heuristics with memory of previously seen items can even lead to heuristics outperforming best fit [84]. In addition, these evolved heuristics can sometimes generalize to larger instances than the ones they were trained on [85], similar to the learned *FunSearch* heuristics. However, as is the case with GP, one of the fundamental limitations of hyper-heuristics is that the components of the evolved

heuristic must be manually defined by the user and often need to be tailored to a specific problem to be effective. The LLM in *FunSearch* allows us to bypass this limitation and learn heuristics for bin packing and job scheduling as well as discovering novel mathematical constructions, all within a single pipeline without problem specific tuning.

Program superoptimization and software engineering. Searching for the best way of modifying source code is a task that appears in multiple branches of computer science and software development. These occurrences can be broadly classified into two groups: first, where the goal is to find semantic-preserving modifications (this arises in *program optimization* and *superoptimization*, where the aim is to modify the program so that it executes faster while maintaining its input-output behaviour), and second, where the goal is to find programs with different semantics (this arises, e.g., in *automatic program repair* and *mutation testing*). With some exceptions discussed below, most of these areas use relatively simple and hard-coded mutation operators on either the source code directly (such as deleting or swapping lines) or on the abstract syntax tree (AST).

Machine learning approaches have been used for program superoptimization. For example, [86] used reinforcement learning to learn the sampling probabilities used within a hierarchical probabilistic model of simple program edits introduced by STOKE [87]. Neural networks have also been proposed as a mutation operator for program optimization in [88]. These works operated on code written in Assembly (perhaps because designing meaningful and rich edit distributions on programs in higher-level languages is challenging). More recently, [14] used LLMs to find performance-improving edits to code written in C++ or Python. We also note that reinforcement learning has recently been applied to discover new faster algorithms for fundamental operations such as matrix multiplication [89] and sorting [90].

In this paper, we have not explicitly explored semantic-preserving applications such as discovering performance-improving code edits, but we believe that *FunSearch* could be an effective method for that setting too. In both use cases presented in Section 2, the goal is to evolve programs with new semantics, but the application is different from program repair or mutation testing: in Section 2.1 we used *FunSearch* to discover a program that constructs a previously unknown mathematical object, and in Section 2.2 we used *FunSearch* to discover a program that corresponds to a more efficient heuristic for online bin packing.

Data availability. The experiments carried out in this paper do not require any data corpus other than the publicly available OR-Library bin packing benchmarks [24]. The output functions of interest produced by *FunSearch* are shown across the main paper and in text files in the Supplementary Information.

Code availability. The discovered functions as well as the evolutionary algorithm, code manipulation routines, and a single-threaded implementation of the *FunSearch* pipeline are available as Python code in the Supplementary information and at `https://github.com/google-deepmind/funsearch`. Additionally, the software library launchpad [91], and a sandbox for safely executing generated code on our internal distributed system were used. No training or fine-tuning of a large language model is required; API access for inference is sufficient. We used Codey [27], which is available through its API, and StarCoder [7], which is open source.

References

[55] J.-B. Mouret, S. Doncieux, Overcoming the bootstrap problem in evolutionary robotics using behavioral diversity, in: 2009 IEEE Congress on Evolutionary Computation, 2009, pp. 1161–1168.

[56] J. K. Pugh, L. B. Soros, K. O. Stanley, Quality diversity: A new frontier for evolutionary computation, Frontiers in Robotics and AI 3 (2016) 40.

[57] T. Helmuth, L. Spector, J. Matheson, Solving uncompromising problems with lexicase selection, IEEE Transactions on Evolutionary Computation 19 (2015) 630–643.

[58] M. Hutter, S. Legg, Fitness uniform optimization, IEEE Transactions on Evolutionary Computation 10 (2006) 568–589.

[59] M. de la Maza, An analysis of selection procedures with particular attention paid to proportional and boltzmann selection, in: Proceedings of the fifth international conference on genetic algorithms, 1993, Morgan Kaufmann, 1993.

[60] OpenAI, GPT-4 technical report, 2023. arXiv:2303.08774.

[61] B. Millidge, Scaffolded LLMs as natural language computers, https://www.beren.io/2023-04-11-Scaffolded-LLMs-natural-language-computers, 2023. [Online; accessed July-2023].

[62] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, T. Scialom, Toolformer: Language models can teach themselves to use tools, arXiv preprint arXiv:2302.04761 (2023).

[63] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, M. S. Bernstein, Generative agents: Interactive simulacra of human behavior, arXiv preprint arXiv:2304.03442 (2023).

[64] J. Wu, L. Ouyang, D. M. Ziegler, N. Stiennon, R. Lowe, J. Leike, P. Christiano, Recursively summarizing books with human feedback, arXiv preprint arXiv:2109.10862 (2021).

[65] M. Nye, A. J. Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, A. Odena, Show your work: Scratchpads for intermediate computation with language models, arXiv preprint arXiv:2112.00114 (2021).

[66] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, Y. Cao, ReAct: Synergizing reasoning and acting in language models, in: International Conference on Learning Representations, 2023.

[67] E. Zelikman, Y. Wu, J. Mu, N. Goodman, Star: Bootstrapping reasoning with reasoning, Advances in Neural Information Processing Systems 35 (2022) 15476–15488.

[68] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, A. Anandkumar, Voyager: An open-ended embodied agent with large language models, arXiv preprint arXiv:2305.16291 (2023).

[69] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski, et al., Natural language to code generation in interactive data science notebooks, arXiv preprint arXiv:2212.09248 (2022).

[70] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, X. V. Lin, Lever: Learning to verify language-to-code generation with execution, in: International Conference on Machine Learning, PMLR, 2023, pp. 26106–26128.

[71] S. Zhou, U. Alon, F. F. Xu, Z. Jiang, G. Neubig, Docprompting: Generating code by retrieving the docs, in: International Conference on Learning Representations, 2022.

[72] W. Banzhaf, P. Nordin, R. E. Keller, F. D. Francone, Genetic programming: an introduction: on the automatic evolution of computer programs and its applications, Morgan Kaufmann Publishers Inc., 1998.

[73] W. B. Langdon, R. Poli, Foundations of genetic programming, Springer Science & Business Media, 2013.

[74] H. Ma, A. Narayanaswamy, P. Riley, L. Li, Evolving symbolic density functionals, Science Advances 8 (2022).

[75] M. Schmidt, H. Lipson, Distilling free-form natural laws from experimental data, science 324 (2009) 81–85.

[76] X. Chen, C. Liang, D. Huang, E. Real, K. Wang, Y. Liu, H. Pham, X. Dong, T. Luong, C.-J. Hsieh, et al., Symbolic discovery of optimization algorithms, arXiv preprint arXiv:2302.06675 (2023).

[77] J. R. Koza, Genetic programming II: automatic discovery of reusable programs, MIT press, 1994.

[78] R. Salustowicz, J. Schmidhuber, Probabilistic incremental program evolution, Evolutionary computation 5 (1997) 123–141.

[79] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, S. Schulenburg, Hyper-heuristics: An emerging direction in modern search technology, Handbook of metaheuristics (2003) 457–474.

[80] P. Ross, Hyper-heuristics, Search methodologies: introductory tutorials in optimization and decision support techniques (2005) 529–556.

[81] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, R. Qu, Hyper-heuristics: A survey of the state of the art, Journal of the Operational Research Society 64 (2013) 1695–1724.

[82] E. K. Burke, M. R. Hyde, G. Kendall, Evolving bin packing heuristics with genetic programming, in: International Conference on Parallel Problem Solving from Nature, Springer, 2006, pp. 860–869.

[83] E. K. Burke, M. R. Hyde, G. Kendall, J. Woodward, Automatic heuristic generation with genetic programming: evolving a jack-of-all-trades or a master of one, in: Proceedings of the 9th annual conference on Genetic and evolutionary computation, 2007, pp. 1559–1565.

[84] E. K. Burke, M. R. Hyde, G. Kendall, Providing a memory mechanism to enhance the evolutionary design of heuristics, in: IEEE Congress on Evolutionary Computation, IEEE, 2010, pp. 1–8.

[85] E. K. Burke, M. Hyde, G. Kendall, J. R. Woodward, The scalability of evolved on line bin packing heuristics, in: 2007 IEEE Congress on Evolutionary Computation, IEEE, 2007, pp. 2530–2537.

[86] R. Bunel, A. Desmaison, P. Kohli, P. H. Torr, M. P. Kumar, Learning to superoptimize programs, in: International Conference on Learning Representations, 2017.

[87] E. Schkufza, R. Sharma, A. Aiken, Stochastic superoptimization, ACM SIGARCH Computer Architecture News 41 (2013) 305–316.

[88] A. Shypula, P. Yin, J. Lacomis, C. L. Goues, E. Schwartz, G. Neubig, Learning to superoptimize real-world programs, in: Deep Learning for Code Workshop (ICLR 2022 Workshop), 2022.

[89] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R Ruiz, J. Schrittwieser, G. Swirszcz, et al., Discovering faster matrix multiplication algorithms with reinforcement learning, Nature 610 (2022) 47–53.

[90] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern, et al., Faster sorting algorithms discovered using deep reinforcement learning, Nature 618 (2023) 257–263.

[91] F. Yang, G. Barth-Maron, P. Stańczyk, M. Hoffman, S. Liu, M. Kroiss, A. Pope, A. Rrustemi, Launchpad: a programming model for distributed machine learning research, arXiv preprint arXiv:2106.04516 (2021).

Acknowledgments. We would like to thank Rohan Anil, Vlad Feinberg, Emanuel Taropa, Thomas Hubert, Julian Schrittwieser, Sebastian Nowozin for their LLM support; Tom Schaul, Chrisantha Fernando, Andre Barreto, Prateek Gupta for discussions on evolutionary algorithms; Michael Figurnov and Taylan Cemgil for reviewing the paper; Federico Piccinini, Sultan Kenjeyev for their support on job scheduling, Sam Blackwell for technical support; Olaf Ronneberger, Felix Gimeno, Blanca Huergo, Abbas Mehrabian and Ankit Anand for useful advice; George Holland for program management support.

Contributions. BRP conceived the project with help from AF and PK. AF scoped problems and developed project vision. BRP and AN developed the initial FunSearch codebase. AN, BRP, M. Balog, FR, M. Barekatain, ED, AF implemented and refined the different components of the system. M. Barekatain, AN imported and experimented with LLMs. M. Barekatain, AN, M. Balog worked on evaluating, debugging, and improving the efficiency of experiments. M. Balog, M. Barekatain, BRP, AN, AF, OF, JE contributed to the cap set problem. MPK, M. Balog, JE researched and analyzed results about the admissible sets problem. ED, M. Barekatain, PW contributed to the online bin packing problem. FR, OF researched and did experiments on other problems (Shannon capacity and corners problem), PK contributed technical advice and ideas. AF, BRP, ED, FR, MPK, M. Balog, AN, JE, M. Barekatain wrote the paper. These authors contributed equally: BRP, M. Barekatain, AN, M. Balog, MPK, ED, FR, AF.

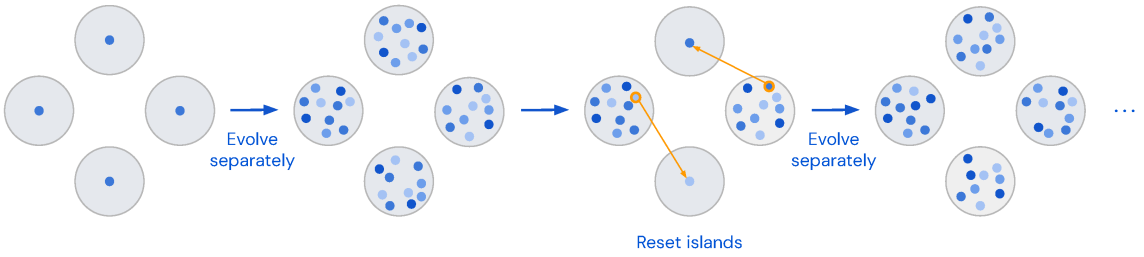
Corresponding authors. Correspondence to Bernardino Romera-Paredes, Pushmeet Kohli or Alhussein Fawzi.

Competing interests. The authors of the paper are planning to file a patent application relating to subject matter contained in this paper in the name of Google DeepMind.

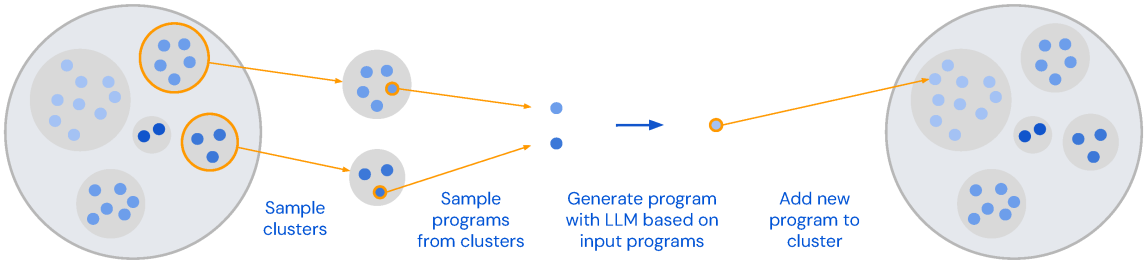
Additional information. Supplementary Information is available for this paper.

```python
"""Finds large cap sets."""
import numpy as np
import utils_capset

def priority_v0(element, n):
  """Returns the priority with which we want to add `element` to the cap set."""
  #######
  # Code from lowest-scoring sampled program.
  return ...
  #######

def priority_v1(element, n):
  """Improved version of `priority_v0`."""
  #######
  # Code from highest-scoring sampled program.
  return ...
  #######

def priority_v2(element, n):
  """Improved version of `priority_v1`."""
```

Extended Data Figure 1: Example of best-shot prompting, based on the skeleton from Figure 2 (a). The prompt includes $k = 2$ implementations sampled from the programs database, with higher-scoring implementations being more likely to be included.



Extended Data Figure 2: Evolutionary method. The initial programs are separated into islands and each of them are evolved separately. After a number of iterations, the islands with the worst score are wiped and the best program from the islands with the best score are placed in the empty islands. Evolution then proceeds separately again until the next reset. This process is repeated until termination.



Extended Data Figure 3: Program clusters within islands. Within each island, programs are grouped into clusters based on their signature (i.e., their scores on several inputs). We first sample clusters, favoring the ones with higher score. Within the chosen clusters, we sample a program, favoring shorter programs. The sampled programs are used to prompt the LLM which generates a new program. If the new program is correct, it is added to the island, either in an existing cluster or a new one if its signature was not yet present.