Using LLMs To Construct Adversarial Instances in Combinatorial Optimization

Henri Nikoleit

Born 13th June 2002 in Gardelegen, Germany

5th November 2025

Master's Thesis Mathematics

Advisor: Prof. Dr. Heiko Röglin

Institute of Computer Science

Second Advisor: Prof. Dr. Floris van Doorn

Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

1 Introduction

FunSearch is a method for finding good solutions to optimisation-problems. It works similarly to local-search, but searches for code instead of vectors, and perturbs vectors by querying a large language model (LLM) instead of adding pseudo-random noise (Romera-Paredes et al. 2024). In this work, we use FunSearch to find new adversarial constructions for different combinatorial optimization problems.

In standard local-search, one minimises some function $f : \mathbb{R}^d \to \mathbb{R}$ by starting with a random vector $v \in \mathbb{R}^d$, and iteratively checking some random point v' in a small neighborhood around v. If $f(v') < f(v)$, we replace v by v', otherwise leaving v unchanged. Thus, the values $f(v)$ decrease over time, converging to a local minimum.

While this is useful for practical applications, one can also use it for theoretical purposes: If the worst-case performance of some algorithm is unknown, we could try finding worse inputs by defining $f(v)$ to be the performance of the algorithm on some input-vector $v \in \mathbb{R}^d$. After finding a set of inputs that the algorithm performs badly on, we can try analysing these sets of inputs, hoping to notice a particular structure in them and thus proving stronger results on the worst-case performance of the algorithm. Unfortunately, the sets of inputs found this way usually don't have a lot of structure.

FunSearch mitigates this by not searching for vectors $v \in \mathbb{R}^d$, but by searching for python-code that produces such vectors v. Instead of choosing a random point in the "neighborhood" of the current python-code, we query an LLM, asking for a changed version of the supplied program. This has the advantages that:

- Symmetries are much more readily encoded in python.
- Analysing python-programs is easier than analysing the vector $v \in \mathbb{R}^d$.
- While the random-point-selection in local-search is independent of f, we can add context about the problem in our LLM-call.

We used FunSearch to find worst-case instances for certain problems in combinatorial optimization:

- A variant of the Best-Fit algorithm for the bin-packing problem, which shuffles the order of the input-items beforehand,
- The Nemhauser-Ullmann algorithm for calculating the Pareto-set of knapsack-problems,
- The competitiveness of hierarchical clustering for the k-median objective,
- The iterative-rounding algorithm for the generalised Gasoline-problem.

We attempted using FunSearch for several other problems as well, but were unsuccessful, see Section 4.5 for details.

	Best-Fit	Knapsack	k-median	Gasoline
Previous Best Lower Bound	1.3	2.0	1.0	2.0
Local Search	1.478	1.93	1.36	2.11
FunSearch without Hand-Tuning	1.497	646.93	1.538	3.05
FunSearch with Hand-Tuning	1.5	1.415^n	1.618	4.65
Known Upper Bound	1.5	2.0^n	16.0	None

Table 1: Comparison across different problems of: Previous state of the art, local search (see Section 3.1), FunSearch without hand-tuning (Section 3.2), FunSearch with hand-tuning (Section 3.3), and the best-known upper bounds.

For each problem, we first used FunSearch to find an instance the algorithm performed badly on. Afterwards, we analyzed and tuned the found programs by hand, until we ended up with an instance that was simple and symmetrical enough to lend itself to theoretical analysis, leading to the results in Table 1.

1.1 Structure of This Thesis

In Section 2, we introduce the four optimisation-problems, the specific questions we want to make progress on, and prior results on these questions. The four subsections are independent of each other, they can be read in any order.

In Section 3, we introduce local-search, motivate FunSearch, present ablations for different hyper-parameters, and give implementation-details for each of the four problems: Both local-search and FunSearch require evaluating $f(v)$ many times, which usually requires calculating the optimal-solution to some NP-hard problem.

In Section 4, we show the instances found by FunSearch, how we tuned them into ones that lend themselves to mathematical analysation, and then conduct that mathematical analysation by giving proofs for the above results. For the gasoline-problem, we did not manage to find a proof, but provide empirical evidence instead. These four subsections can also be read in any order. We finally list the problems that we failed to make progress on.

1.2 Academic Integrity

This work was the result of a collaboration between Ankit Anand from Google Deepmind, Heiko Röglin, Anurag Murty Naredla, and myself. I am deeply grateful for their support and our productive discussions. The result of this collaboration was the paper **Nikoleit, Henri, Ankit Anand, Anurag Murty Naredla, and Heiko Röglin. 2025.** ***Adversarial Examples for Heuristics in Combinatorial Optimization: An LLM Based Approach*****.** , which is currently under review. For writing this thesis, I only copied the bibliography-file containing citation-data and parts that I wrote myself from that paper.

Except where explicitly noted otherwise, no part of this thesis was written by a Large Language Model (LLM). I *did* use generative AI in the following ways:

- LLMs are an integral component of FunSearch, the search-algorithm we used (see Section 3.2).
- I used Gemini Deep Research to find existing literature and references.
- I sparingly used generative AI for writing the implementation (see Section 3.5). This was helpful for repetitive problems with lots of training-data (e.g. the JavaScript code for the website) and unhelpful for critical implementations with fewer training-data (e.g. branch and bound in the clustering-solver).

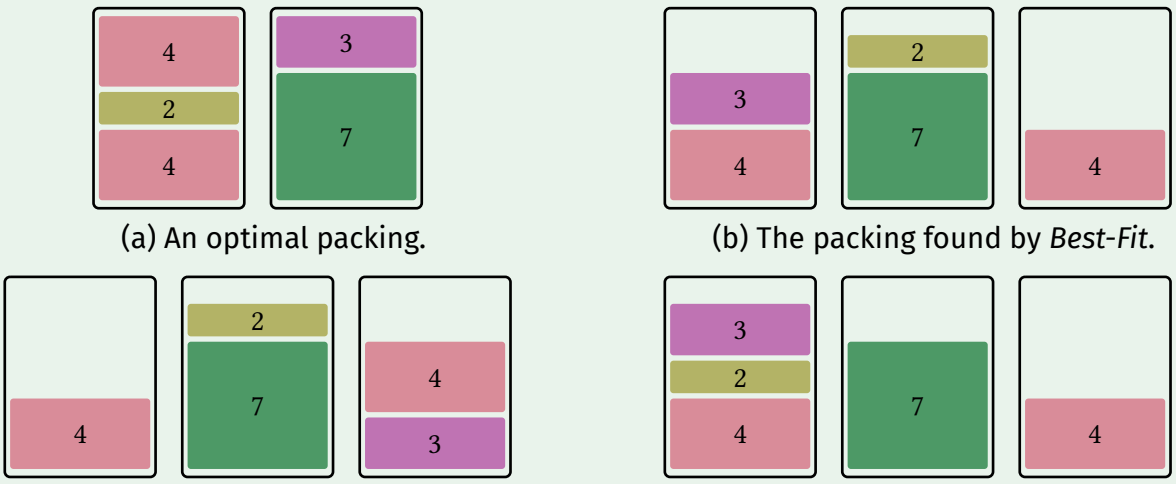
2 Definitions and Prior Work

2.1 Bin-Packing

In the bin-packing problem, we are given a bin-capacity c and a list of n items with weights $w_1, ..., w_n$, each bounded by c. Our task is to find a *packing*, i.e. we must pack all items into bins of capacity c such that each item is in exactly one bin and for all bins, the sum of its contained items must not exceed c. Our objective is to use as few bins as possible. Finding a packing with the minimum number of bins is NP-hard (Garey and Johnson 1979).

Example 2.1.1: We have to assign the following five items to bins with capacity $c = 10$:

$$w_1, ..., w_5 \quad = \quad 4, 7, 2, 3, 4$$



(a) An optimal packing.

(b) The packing found by *Best-Fit*.

(c) The packing found by *Next-Fit*.

(d) The packing found by *First-Fit*.

Figure 1: Different Packings for $w_1, ..., w_5$, with bins of capacity 10.

In practice, heuristics are used, which do not attempt to find the best possible packing, but quickly find a packing that nevertheless uses few bins (Albers et al. 2021; Rieck 2021). All of the following heuristics are *online*: The items w_i arrive in sequence and the heuristic has to assign w_i permanently to a bin. Once the item w_i has been processed, its assignment can not be changed.

- *Best-Fit*: When item w_i arrives, pack it into a bin which has the least remaining space among the bins that can contain w_i. If no such bin exists, open a new one.
- *Next-Fit*: When item w_i arrives, pack it into the bin that w_{i-1} was assigned to, or open a new bin if this is not possible.
- *First-Fit*: Order the bins by the time in which they were opened, and pack w_i into the oldest bin in which it fits. If no such bin exists, open a new one.

These heuristics will usually not output an optimal solution, i.e. a packing that uses the fewest number of bins (see Example 2.1.1). To compare the performance of different heuristics, we can use the following definition:

Definition 2.1.2: Let $\mathcal{I}$ be the set of all (nonempty) bin-packing instances. For some instance $I \in \mathcal{I}$, let $\mathrm{Opt}(I)$ be the number of bins in an optimal packing, and $\mathcal{A}(I)$ be the number of bins in the packing found by a bin-packing algorithm $\mathcal{A}$. The **(absolute) approximation-ratio of** $\mathcal{A}$ is

$$\rho_{\mathscr{A}} \quad := \quad \sup_{I \in \mathcal{I}} \frac{\mathscr{A}(I)}{\mathrm{Opt}(I)}.$$

The approximation-ratio of an algorithm captures its worst-case performance. For instance, $\rho_{\mathrm{BestFit}} = 1.7$ (proven by Dósa and Sgall (2014)), meaning that:

- For every instance, the packing found by Best-Fit will never use more than 1.7 times as many bins as an optimal packing, and
- There is a sequence of instances $I_1, I_2, \ldots$ such that $\frac{\mathrm{BestFit}(I_j)}{\mathrm{Opt}(I_j)}$ converges to 1.7.

Dósa and Sgall (2013) proved that $\rho_{\mathrm{FirstFit}} = 1.7$ as well, and Boyar et al. (2012) showed $\rho_{\mathrm{NextFit}} = 2$.

Comparing algorithms by their absolute approximation-ratios can be a bit pessimistic: In practice, if we are in a position where we must use an online-algorithm, it might not be the case that an adversary can choose *the entire input* I including the order of its items. Hence, we consider a less pessimistic measure for the performance of an algorithm:

Definition 2.1.3: Let S_n be the set of permutations on n elements, i.e. the symmetric group. The **absolute random-order-ratio of** $\mathscr{A}$ is

$$\mathrm{RR}_{\mathscr{A}} \quad := \quad \sup_{I \in \mathcal{I}} \mathbb{E}_{\pi \in S_{|I|}} \left[\frac{\mathscr{A}(\pi(I))}{\mathrm{Opt}(I)} \right].$$

That is to say: We still assume an adversary can choose the *items* of the instance, but their *order* is randomized before being passed on to algorithm $\mathscr{A}$. Note that $\mathrm{Opt}(I)$ does not depend on the order of the items ($\mathrm{Opt}(I) = \mathrm{Opt}(\pi(I))$ for all $\pi \in S_{|I|}$). Kenyon (1996) showed that $1.08 \leq \mathrm{RR}_{\mathrm{BestFit}} \leq 1.5$, and Albers et al. (2021) improved the lower bound to 1.3.

Example 2.1.4: This example is (one instantiation of) the lower-bound construction by Albers et al. (2021) showing $1.3 \leq \mathrm{RR}_{\mathrm{BestFit}}$. Consider bins of capacity $c = 3000$ and the instance:

$$I \quad := \quad [1004, 1004, \ 1016, 1016, \ 992].$$

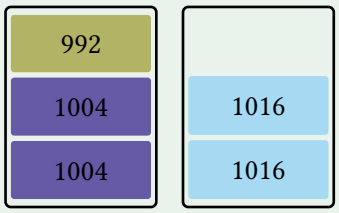


Figure 2: An optimal packing of I.

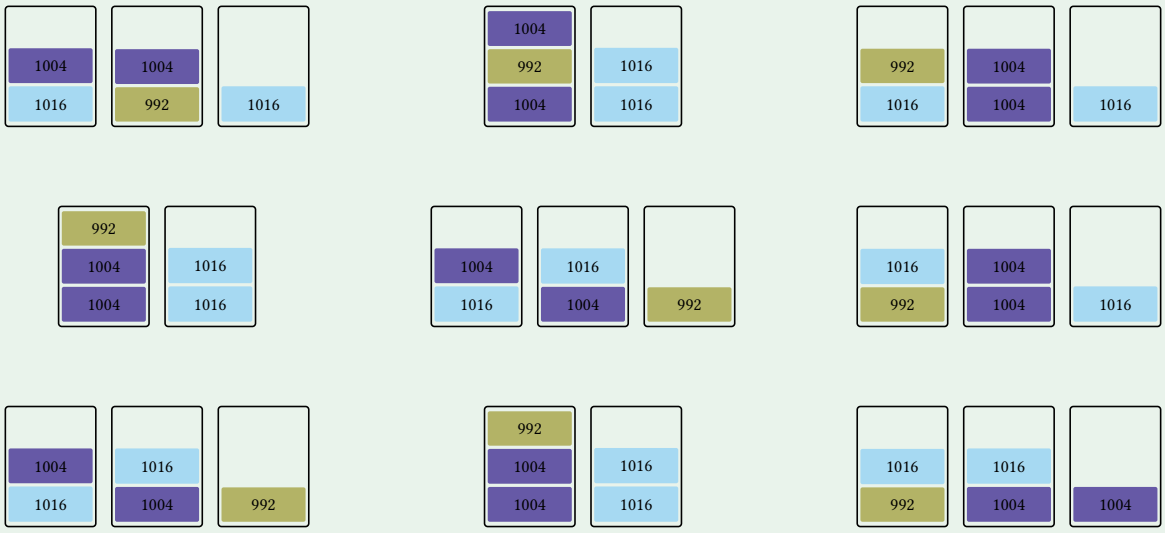


Figure 3: Nine different packings produced by Best-Fit on I with randomised order.

Using FunSearch, we found a sequence of instances $I_1, I_2, \ldots$ for which $\mathbb{E}_{\pi \in S_{|I_j|}}\left[\mathscr{A}(\pi(I_j))/\operatorname{Opt}(I_j)\right]$ converges to 1.5, showing $\mathrm{RR}_{\mathrm{BestFit}} \geq 1.5$. Because this matches the upper bound, this proves that $\mathrm{RR}_{\mathrm{BestFit}} = 1.5$ exactly. The details are in Section 4.1.

2.2 Knapsack Problem

In the Knapsack-Problem, we are given a capacity c and a list I of n items, each item having both a non-negative weight $w_i \leq c$ and a non-negative profit p_i. Instead of minimising the number of bins we use, we only have *a single bin* of capacity c at our disposal, and the sum of weights of the items we put in this bin must not exceed c. Our objective instead is to *maximize* the sum of profits of the items we put in the bin.

A **solution** is any sub-list of the list of items I, regardless of whether it exceeds the capacity c. For some solution A, we denote by $\operatorname{Weight}(A)$ its total weight (i.e. the sum of the weights of the items in A), and by $\operatorname{Profit}(A)$ its total profit. We can visualize the space of *all* possible solutions —including those that exceed the maximum weight capacity— by plotting the tuple $(\operatorname{Weight}(A), \operatorname{Profit}(A))$ for all $2^{|I|}$ solutions A.

Example 2.2.1: We denote items by a column-vector $\binom{\text{Weight}}{\text{Profit}}$. We are given a capacity $c = 20$ and the following items:

$$I = \left[\binom{4}{9}, \binom{5}{1}, \binom{13}{14}, \binom{3}{8}, \binom{11}{4}, \binom{6}{14}\right]$$

An optimal solution is $\left[\binom{4}{9}, \binom{5}{1}, \binom{3}{8}, \binom{6}{14}\right]$, with total weight 18 and total profit 32.

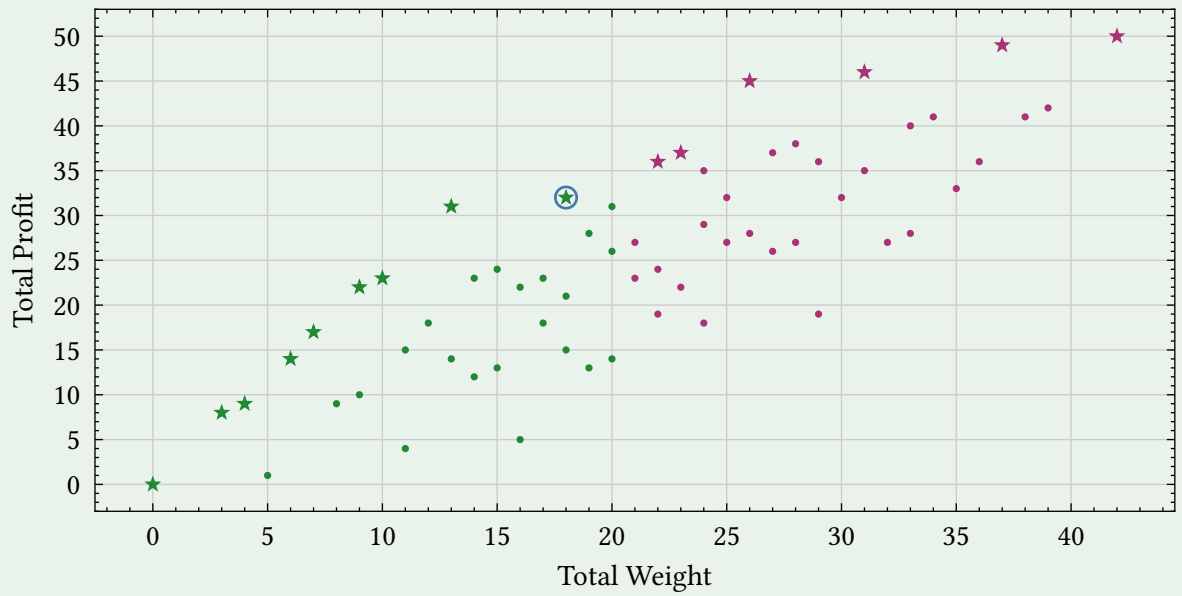


Figure 4: All 2^6 possible solutions to Example 2.2.1. Solutions exceeding capacity $c = 20$ are marked in purple. The optimum is circled in blue. Pareto-optimal solutions are marked by $\star$.

2.2.1 Pareto-Sets

In practice, one might not know the capacity c, or might have unlimited capacity but some tradeoff-function between weights and profits, for example $u(w, p) = p - w^2$ that must be maximised instead. The original objective ("maximise total profit while having total weight at most c") can also be expressed by such a tradeoff-function, $u(w, p) = p \cdot \chi_{w \leq c}$, with indicator-function χ.

All of these cases can be covered simultaneously: We can narrow down the space by eliminating all solutions that can *never* be optimal for any reasonable tradeoff-function. The set of those solutions is the *Pareto-set* (Röglin 2020):

Definition 2.2.2: For solutions A and B, we say A **dominates** B if and only if:

$$\text{Weight}(A) \leq \text{Weight}(B) \quad \text{and} \quad \text{Profit}(A) \geq \text{Profit}(B),$$

and at least one of those inequalities is strict. The **Pareto-set** $P(I)$ is the multiset of all solutions that are not dominated by any other solution.

See Figure 4 for an example. In the above definition, we say that $P(I)$ is a *multiset*, rather than a set. This is unfortunately necessary, consider the following instance:

$$I := \left[\binom{1}{1}, \binom{1}{1}\right] \quad \Longrightarrow \quad P(I) = \left\{[], \quad \left[\binom{1}{1}\right], \quad \left[\binom{1}{1}\right], \quad \left[\binom{1}{1}, \binom{1}{1}\right]\right\}.$$

Had we defined $P(I)$ as a *set of solutions*, the above $P(I)$ would only have 3 elements, as the two solutions $\left[\binom{1}{1}\right]$ are equal. As a multiset, the size of $P(I)$ is 4. Some authors mitigate this confusion by denoting solutions not as sub-lists of I, but as 0-1 vectors in $\{0, 1\}^I$, as that way, all solutions are unique.

In Figure 4, the Pareto-set has size 15, which is much smaller than the size of the entire solution-space, $2^6 = 64$. In fact, the Pareto-set is usually small in practice (Moitra and O'Donnell 2011; Röglin

2020), so one approach to finding an optimal solution is to compute the Pareto-Set $P(I)$ and, for a given tradeoff-function u, find a solution in $P(I)$ that maximizes u. If $P(I)$ has already been computed, a simple linear search yields an optimal solution in time $O(|P(I)|)$.

Let $n := |I|$. The standard algorithm for computing $P(I)$ is the *Nemhauser-Ullmann algorithm* (Nemhauser and Ullmann 1969; Röglin 2020), which incrementally computes the Pareto-sets $P_i := P(I_{1:i})$ for $i = 1, ..., n$, where "$I_{1:i}$" denotes the instance containing the first i items of I. It works as follows:

Algorithm 1: Nemhauser-Ullmann Algorithm for Pareto-Sets

```
Set P_0 = {∅}.
For i = 1, ..., |I|:
    Let x be the i-th item of I.
    Set Q_i := P_{i-1} ∪ {A ∪ {x} | A ∈ P_{i-1}}
    Compute P_i := {A ∈ Q_i | A is not dominated by any B ∈ Q_i}
```

This algorithm works correctly because P_i is always a subset of Q_i. We need the following definition for stating the runtime of Algorithm 1.

Definition 2.2.3: Consider the equivalence-relation $\sim$ between solutions A, B of some instance I defined by: $A \sim B$ iff [$\text{Weight}(A) = \text{Weight}(B)$ and $\text{Profit}(A) = \text{Profit}(B)$].

The **deduplicated Pareto-Set** $P_{\text{dedup}}(I)$ is the quotient $P(I)/\sim$. In other words, it is the Pareto-Set, but two solutions are treated as identical iff they have the same total weight and the same total profit.

With some work, and if one is only interested in computing $P_{\text{dedup}}(I)$, Algorithm 1 can be implemented to run in time $O\big(|P_{\text{dedup}}(I_{1:1})| + ... + |P_{\text{dedup}}(I_{1:n})|\big)$ (Röglin 2020).

Intuitively, one might think that:

- P_{i-1} is always smaller than P_i, which would imply a runtime of $O(n \cdot |P_n|)$, because $\left|P_{\text{dedup}}(J)\right| \leq |P(J)|$.
- $P_{\text{dedup}}(I_{1:i-1})$ is always smaller than $P_{\text{dedup}}(I_{1:i})$, which would imply a runtime of $O\Big(n \cdot \left|P_{\text{dedup}}(I)\right|\Big)$.

However, neither statement is true in general:

Example 2.2.4: Consider the items:

$$I := \left[\binom{4}{4}, \quad \binom{4}{4}, \quad \binom{2}{1}, \quad \binom{1}{2}, \quad \binom{2}{2}\right].$$

- P_4 has size 12, while $P_5 = P(I)$ has size 10.
- $P_{\text{dedup}}(I_{1:4})$ has size 9, while $P_{\text{dedup}}(I)$ has size 8.

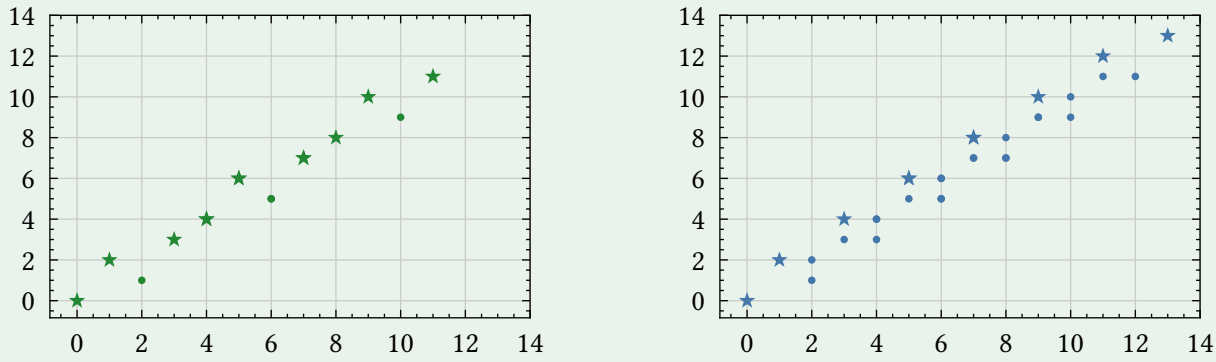


Figure 5: The solution-space for $I_{1:4}$ (left) and $I_{1:5} = I$ (right), plotting $(\mathrm{Weight}(A), \mathrm{Profit}(A))$ for every solution A, with Pareto-optimal solutions marked by $\star$. Because solutions with the same total weight and total profit are plotted at the same point, only the deduplicated Pareto-Sets are visible.

Let $n := |I|$ again. It had been unknown whether $|P_i|$ can be bounded by some $O(|P_n|)$ at all, i.e. it had been unknown whether

$$\mathrm{Score}(I) \quad := \quad \frac{\max_{1 \le i \le n} |P(I_{1:i})|}{|P_n|}$$

can always be bounded by some constant not depending on I. For the specific I in Example 2.2.4, $\mathrm{Score}(I) = \frac{12}{10} = 1.2$. Note that, for any instance, $\mathrm{Score}(I) \le 2^n$, because every $|P_i|$ is at most 2^n.

So far, the instances with the highest score only achieved a Score around 2. Using FunSearch, we were first able to find an instance with Score around 5.766, and after modifying this instance by hand, we obtained a sequence of instances $I_1, I_2, \ldots$ with $\mathrm{Score}(I_j) \ge n^{\Omega(\sqrt{n})}$, or more precisely $\mathrm{Score}(I_j) \ge \Omega\left((n/2)^{(\sqrt{n/2}-3)/2}\right)$. These instances had the property that $P_{\mathrm{dedup}}(I_{1:i}) = P(I_{1:i})$, so this bound can also be applied to the runtime of Algorithm 1.

Afterwards, we found an unrelated construction without FunSearch that achieved a Score around $\Omega(1.037^n)$, making the previous result obsolete. We still include the previous result and its proof. Using FunSearch, we could improve this to $\Omega(1.509^n)$ for the above scoring-function, and to $\Omega(1.0519^n)$ for the runtime of the Nemhauser-Ullmann algorithm, i.e. $\frac{\max_{1 \le i \le n} |P_{\mathrm{dedup}}(I_{1:i})|}{|P_{\mathrm{dedup}}(I)|} \ge \Omega\left(1.0519^{|I|}\right)$. The details are in Section 4.2.

2.3 k-Median Clustering

In the clustering-problem, we are given n unlabeled data points $p_1, \ldots, p_n \in \mathbb{R}^d$ and a number k. Our task is to find a **k-clustering**: A partition of the n points into k clusters $C_1, \ldots, C_k$, such that "close" points belong to the same cluster. Clustering is a useful tool for data-analysis. There exist different objectives to quantify "closeness" (Arutyunova and Röglin 2025):

- In k-median clustering, the cost of a cluster C is:

$$\mathrm{Cost}(C) = \min_{\mu \in C} \sum_{x \in C} \|x - \mu\|_1$$

  That is, for the best choice of a center $\mu \in C$, sum the distances of all points $x \in C$ from μ. The total cost of a clustering $C_1, \ldots, C_k$ is the sum of its costs: $\mathrm{Cost}(C_1) + \ldots + \mathrm{Cost}(C_k)$.

- k-means clustering is similar to k-median, but we use the squared L_2 norm and the center μ may be any point from the ambient space, it need not be a point in C. In this scenario, the optimal choice of μ turns out to be the average of all points in C:

$$\text{Cost}(C) = \sum_{x \in C} \|x - \mu(C)\|_2^2, \quad \text{where } \mu(C) := \frac{1}{|C|} \cdot \sum_{x \in C} x.$$

  The total cost of a clustering is again the sum of the cost of its clusters.

- In k-center clustering, the cost of the cluster C is the *radius* of that cluster:

$$\text{Cost}(C) = \min_{\mu \in C} \left(\max_{x \in C} \|x - \mu\|_2 \right)$$

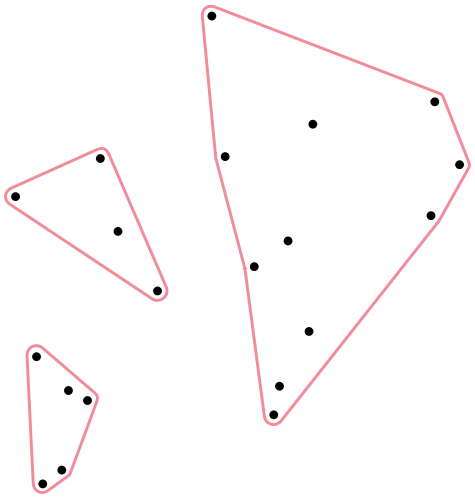
  The cost of a clustering is the *maximum* of the costs of its clusters.

- In k-diameter clustering, the cost of the cluster C is the *diameter* of that cluster:

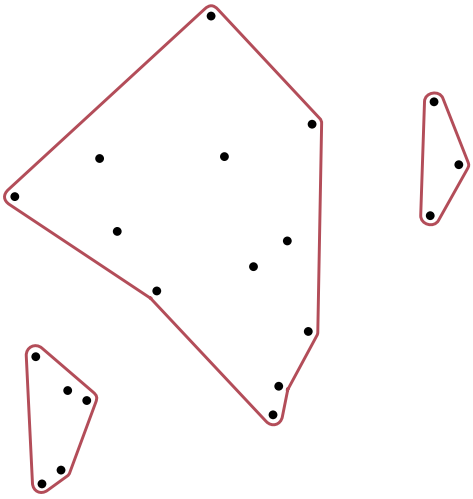
$$\text{Cost}(C) = \max_{x,y \in C} \|x - y\|_2$$

  The cost of a clustering is again the *maximum* of the costs of its clusters.

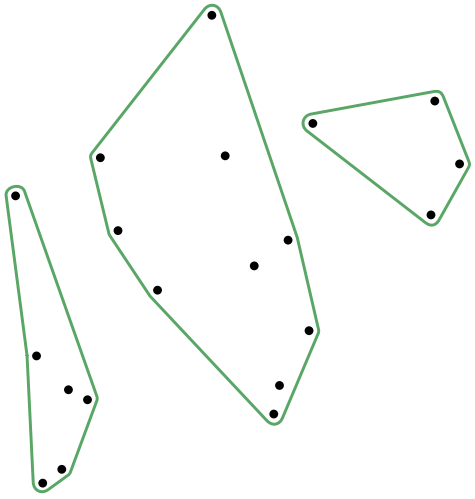
Naturally, different objectives can yield different optimal clusterings, as seen in Figure 6.



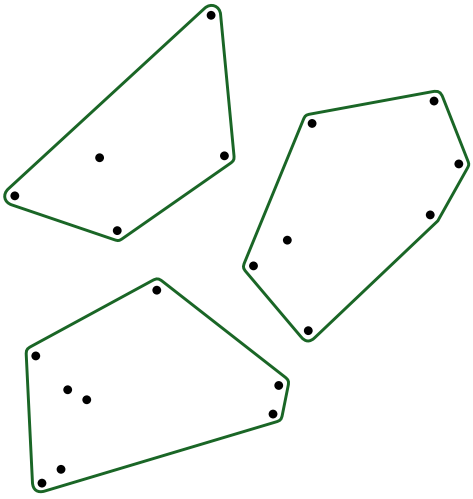
(a) A sub-optimal k-median clustering, obtained via agglomerative clustering.



(b) A sub-optimal k-means clustering, obtained via agglomerative clustering.



(c) An optimal k-median clustering.



(d) An optimal k-means clustering.

Figure 6: Four different k=3-clusterings for the same 20 points in $\mathbb{R}^2$.

There are also weighted versions of the k-median and k-means objectives, where each point p has an associated non-negative weight $w(p)$, and the distance between p and the centre μ is multiplied by $w(p)$. For k-means, the optimal choice for μ then is simply the weighted average of all points. The weighted and unweighted problems are mostly equivalent: If we restrict ourselves to integral weights, an equivalent unweighted instance has $w(p)$-many unweighted copies of p, for each p. If we restrict ourselves to rational weights, we can multiply by the product of the denominators to achieve integral weights again.

When trying to cluster unlabeled data, we usually are not given a number k of clusters to use. In such a scenario, we could use heuristics to determine a good choice of k (see e.g. Schubert (2023)). Alternatively, we could compute a *Hierarchical Clustering*, which is a sequence of nested k-clusterings for every choice of k (Johnson 1967):

Definition 2.3.1: A **hierarchical clustering** on n points is a sequence $(H_1, ..., H_n)$ of clusterings such that:

- H_i is an i-clustering (i.e., it consists of i clusters), for every $i = 1, ..., n$, and
- For every $i = 1, ..., n-1$, the clustering H_i can be obtained by merging two clusters in H_{i+1}. In other words, there exist clusters $C, C' \in H_{i+1}$ such that:

$$H_i \quad = \quad (H_{i+1} \setminus \{C, C'\}) \cup \{C \cup C'\}.$$

This nested structure of hierarchical clusterings is useful for, for example, taxonomy. Finding *some* hierarchical clustering in practice can be done via **agglomerative clustering**, a greedy method where we start with H_n as having each point in a singleton cluster, and construct H_{i-1} from H_i by choosing to merge a pair of clusters that increases the objective the least.

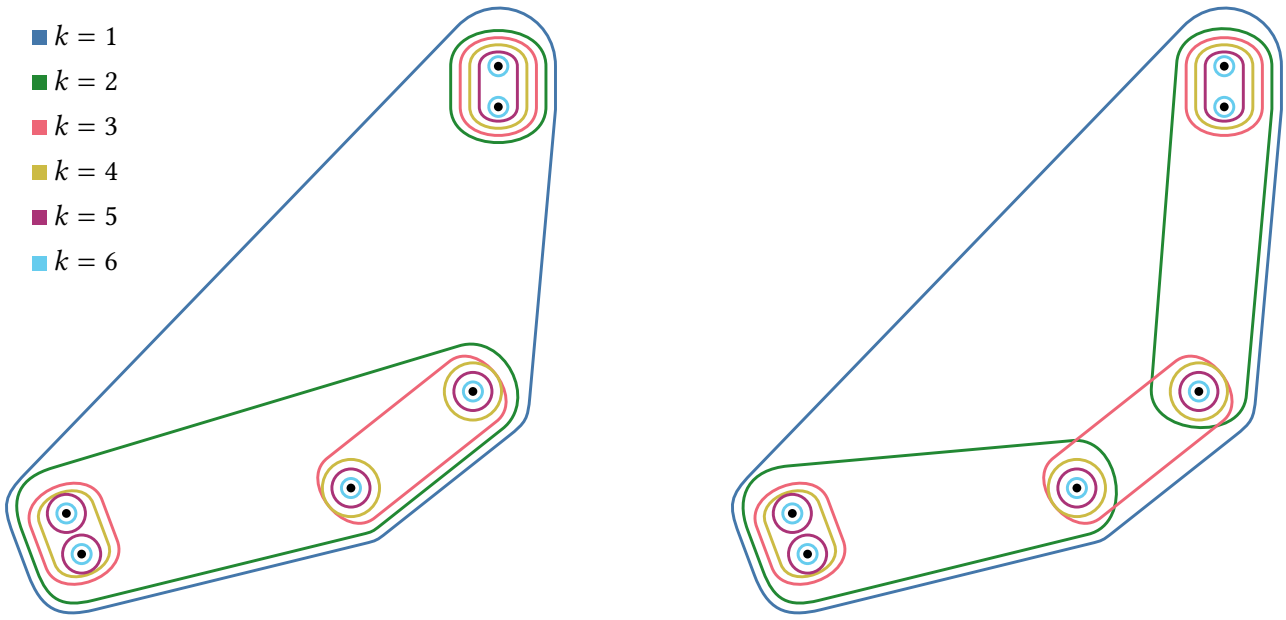


Figure 7: Left: An optimal hierarchical clustering on 6 points for the (unweighted) k-median objective. Right: For each $k = 1, ..., 6$, an optimal k-median clustering on the same 6 points. There is no hierarchical clustering $(H_1, ..., H_n)$ such that each H_k is an optimal k-clustering. The shown hierarchical and optimal clusterings only differ at level $k = 2$.

The additional structure of hierarchical clustering does come at a cost, however: Usually, the optimal k-clusterings need not have a nested structure, so a hierarchical clustering $(H_1, ..., H_n)$ such that every H_i is an *optimal* i-clustering **need not exist**. The set of points in Figure 7 is such an example.

To measure the quality of a hierarchical clustering $(H_1, ..., H_n)$, we could simply sum the the costs of each level: $\text{Cost}(H_1) + ... + \text{Cost}(H_n)$. However, k-clusterings for small k usually have significantly higher cost than k-clusterings for large k, so this would lose information about the quality of the H_i for small i. To avoid this, we can instead compare each level H_i of the hierarchy to an *optimal* i-clustering, and taking the maximum across all levels (Arutyunova and Röglin 2025):

Definition 2.3.2: For a clustering-instance I and a cost-function Cost, the **approximation-factor of a hierarchical clustering** $(H_1, ..., H_n)$ for I is:

$$\text{Apx}_{\text{Cost}}(H_1, ..., H_n) \quad := \quad \max_{i=1,...,n} \frac{\text{Cost}(H_i)}{\text{Cost}(\text{Opt}_i)},$$

where Opt_i is an optimal i-clustering for I with respect to Cost.

For a fixed cost-function Cost, we say that a hierarchical clustering of an instance I is **optimal** if it has the lowest possible approximation-factor among all hierachical clusterings on I (this always exists because the set of hierarchical clusterings is finite).

The hierarchical clusterings and optimal clusterings in Figure 7 only differ for $k = 2$, where $\text{Cost}(H_2) = 1.78$ and $\text{Cost}(\text{Opt}_i) = 1.41$, so the approximation-factor of that hierarchical-clustering is $\frac{1.78}{1.41} \approx 1.262$. This hierarchical clustering is optimal (without proof).

Although agglomerative clustering computes a hierarchical clustering whose approximation-factor is low in practice, this hierarchical clustering need not be an optimal hierarchical clustering. In this work, we only concern ourselves with optimal hierarchical clusterings.

For a cost-function Cost, we can ask what we sacrifice by imposing a hierarchical structure, not just for some fixed instance I, but for *all* instances I. This is the Price of Hierarchy (Arutyunova and Röglin 2025):

Definition 2.3.3: Let $\mathcal{I}$ be the set of all clustering-instances, and Cost some cost-function. For a fixed clustering-instance I, let $\mathcal{H}(I)$ be the (finite) set of all hierarchical clusterings on I. The **Price of Hierarchy for** Cost is defined as:

$$\text{PoH}_{\text{Cost}} \quad := \quad \sup_{I \in \mathcal{I}} \left(\min_{H \in \mathcal{H}(I)} \text{Apx}_{\text{Cost}}(H) \right).$$

In particular, the instance in Figure 7 proves that $\text{PoH}_{k\text{-median}} \geq 1.26$, because the hierarchical clustering there is optimal for this instance.

For the cost-functions mentioned earlier, the following bounds on their Price of Hierarchy are known:

- $\text{PoH}_{k\text{-median}} \leq 16$ (Dai 2014)
- $\text{PoH}_{k\text{-means}} \leq 32$ (Großwendt 2020)
- $\text{PoH}_{k\text{-center}} = 4$ (Arutyunova and Röglin 2025)
- $\text{PoH}_{k\text{-diameter}} = 3 + 2\sqrt{2} \approx 5.828$ (Arutyunova and Röglin 2025)

No non-trivial lower bounds on $\mathrm{PoH}_{k\text{-median}}$ are known. Using FunSearch, we obtained a sequence of instances that shows $\mathrm{PoH}_{k\text{-median}} \geq \frac{1+\sqrt{5}}{2} \approx 1.618$. See Section 4.3 for details.

We also made similar attempts for other objectives, and for the approximation-ratio of agglomerative-clustering, but to no success. We briefly discuss this in Section 4.5.

2.4 Generalised Gasoline-Problem

As a motivating example (similar to Lorieau (2024)) for the generalised gasoline-problem, we are in charge of a factory that produces cookies every day of the week. In doing so, it consumes exactly two ingredients: Flour and sugar. Each day of the week, both the amount of cookies and their sugar-content must follow a certain schedule. For instance, each Monday, we are asked to use $\binom{\text{Flour}}{\text{Sugar}}$-amounts equal to $y_1 = \binom{3}{4}$ for our cookie-production, whereas each Tuesday, we must produce more and sweeter cookies, having to use $y_2 = \binom{2}{8}$ amounts of flour and sugar.

We get flour and sugar delivered to our factory overnight, but we must pick these amounts from a list of seven possible delivery-trucks that are the same every week. We can choose on which day of the week we would like to receive each truck. For instance, we can choose to have $x_1 = \binom{5}{3}$ flour and sugar delivered to our factory on Monday, or $x_2 = \binom{3}{10}$. Within a week, we must receive each of the seven delivery-trucks exactly once, and we can only accept one delivery-truck per night because our driveway is too narrow. It's unlikely that we will be fortunate enough to have, for every demand-vector y_i, a matching delivery-vector x_i (in which case we would just order exactly the ingredients overnight that we would need on the next day), so we must resort to storing leftover ingredients overnight in our yet-to-be-built warehouse.

Corporate has been kind enough to ensure that $y_1 + \ldots + y_7 = x_1 + \ldots + x_7$, meaning that, at the end of every week, we will have exactly the same amount of ingredients in our warehouse as at the beginning of the week. However, storing ingredients takes costly space, so we would like to minimise the total amount of warehouse we need to build, while the only free variable under our control is the permutation of the delivery-trucks across the week. If we are not clever in choosing this permutation, we might end up ordering some large trucks on some days with low production, and end up needing more warehouse space than would be necessary otherwise.

Let S_n be the set of permutations on n elements. Mathematically, our task is to find a permutation π:

$$\min_{\pi \in S_7} \quad \|\alpha - \beta\|_1$$

$$\text{where} \quad \alpha = \min_{1 \leq k \leq 7} \left(\sum_{i=1}^{k} x_{\pi(i)} - \sum_{i=1}^{k} y_i \right) \quad \begin{array}{c}\text{“In the evening, we must have at} \\ \text{least } \alpha \text{ ingredients left over.”}\end{array}$$

$$\beta = \max_{1 \leq k \leq 7} \left(\sum_{i=1}^{k} x_{\pi(i)} - \sum_{i=1}^{k-1} y_i \right) \quad \begin{array}{c}\text{“After the delivery overight, we} \\ \text{must store at most } \beta \text{ ingredients”}\end{array}$$

where the minimum across vectors is taken entry-wise. As an objective, we choose $\|\alpha - \beta\|_1$, meaning we trade off the cost for space in the flour-warehouse linearly against the cost of space in the sugar-warehouse. This does not lose generality on the tradeoff-*ratio* between the two, since tradeoffs like “Sugar-warehouse space is twice as expensive as flour-warehouse space” can be captured by choosing different units for measuring amounts of flour and sugar in the x_i and y_i vectors. Non-linear tradeoffs are not captured, however. We write $X = (x_1, \ldots, x_7)$ and $Y = (y_1, \ldots, y_7)$.

Example 2.4.1: Consider:

$$X = \left[\begin{pmatrix}5\\3\end{pmatrix}, \begin{pmatrix}3\\10\end{pmatrix}, \begin{pmatrix}7\\8\end{pmatrix}, \begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}8\\9\end{pmatrix}, \begin{pmatrix}7\\4\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}\right], \quad Y = \left[\begin{pmatrix}3\\4\end{pmatrix}, \begin{pmatrix}2\\8\end{pmatrix}, \begin{pmatrix}8\\1\end{pmatrix}, \begin{pmatrix}1\\5\end{pmatrix}, \begin{pmatrix}9\\4\end{pmatrix}, \begin{pmatrix}2\\10\end{pmatrix}, \begin{pmatrix}7\\5\end{pmatrix}\right],$$

($x_1+\ldots+x_7$ equals $y_1+\ldots+y_7$, as promised by corporate), together with the following permutation of deliveries:

$$\pi(X) \coloneqq \left[\begin{pmatrix}5\\3\end{pmatrix}, \begin{pmatrix}3\\10\end{pmatrix}, \begin{pmatrix}7\\8\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}7\\4\end{pmatrix}, \begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}8\\9\end{pmatrix}\right].$$

We visualise the state of our warehouse over the course of the week as follows: We use colored bars to represent the current amount of flour (blue) and sugar (purple) in our warehouse. Vectors preceded by "↑" indicate deliveries to our warehouse at night, vectors preceded by "↓" indicate us consuming ingredients from the warehouse to bake cookies during the day. The two horizontal colored lines indicate the maximum number of the respective ingredient that the warehouse must store across the week. We choose the initial stocking of our warehouse *minimally* such that we will always have enough ingredients to never run out (this choice is exactly β from the above optimization problem). This ensures that our warehouse has the smallest possible size for this permutation, and that for both ingredients, there must be a day on which that ingredient's warehouse is fully depleted (otherwise our choice would not be minimal, we would have wasted space).

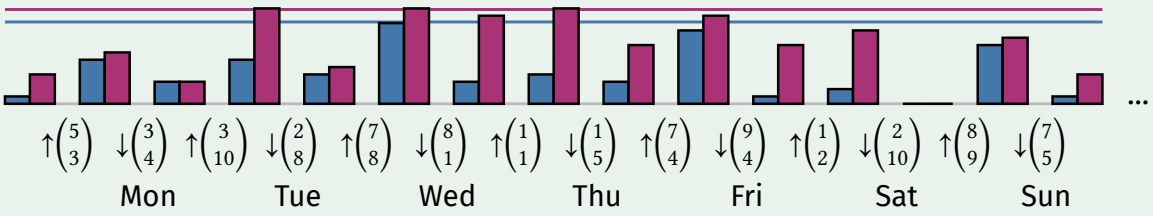


Figure 8: The (cyclical) state of the warehouse across the week for permutation π.

For this permutation, the warehouse must store a peak of 11 flour on the night between Tuesday and Wednesday, and a peak of 13 sugar on several nights between Tuesday and Thursday. There is a better permutation, though:

$$\pi_{\text{Opt}}(X) \coloneqq \left[\begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}7\\8\end{pmatrix}, \begin{pmatrix}5\\3\end{pmatrix}, \begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}8\\9\end{pmatrix}, \begin{pmatrix}7\\4\end{pmatrix}, \begin{pmatrix}3\\10\end{pmatrix}\right],$$

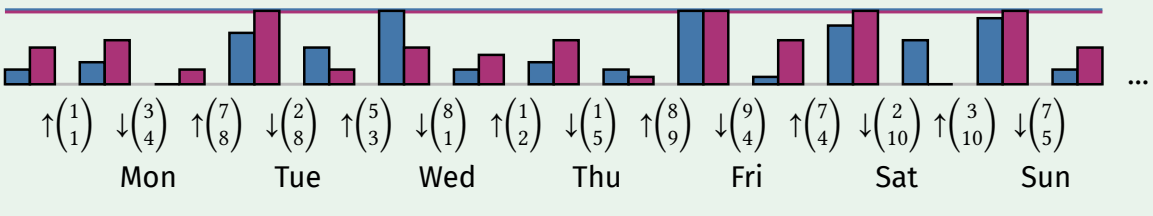


Figure 9: The (cyclical) state of the warehouse across the week for permutation π_{Opt}.

Here, the peak-capacity of the warehouse is only 10 for both flour and sugar, so π_{Opt} is a better choice than π regardless of the tradeoff between the cost of flour-warehouse and sugar-warehouse.

For a different visualisation, we can trace the state of the warehouse in phase-space:

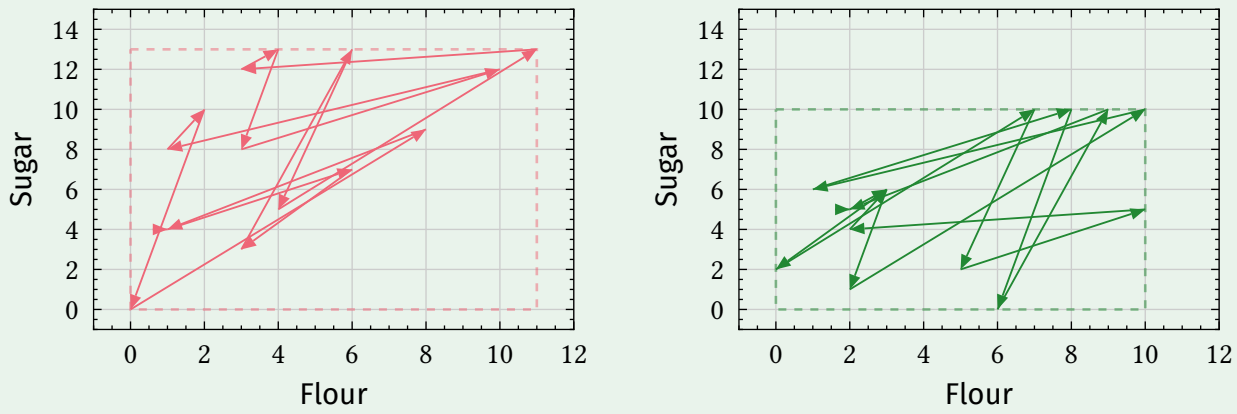


Figure 10: Tracing the states of the warehouses of π (left) and π_{Opt} (right) in phase-space, along with the smallest rectangles containing all points.

The width and height of the smallest rectangle encompassing all those points is exactly the maximum capacity that our flour-warehouse and sugar-warehouse require.

With the L_1 cost-function used above, π has a cost of $11 + 13 = 24$, whereas π_{Opt} has a cost of $10 + 10 = 20$ and is indeed an optimal permutation for this instance.

Generally, an instance of the Gasoline-Problem (named so due to a puzzle by Lovász (1979) involving gas-stations along a circular race-track) consists of two sequences of d-dimensional vectors containing non-negative integral entries:

$$X = (x_1, ..., x_n) \in \mathbb{Z}_{\geq 0}^{n \times d}, \quad Y = (y_1, ..., y_n) \in \mathbb{Z}_{\geq 0}^{n \times d},$$

which have the same total sum $x_1 + ... + x_n = y_1 + ... + y_n$. In the above example, $d = 2$ (the number of different ingredients) and $n = 7$ (the length of a week). Our objective is to find a permutation $\pi \in S_n$ of the X-entries that minimises the prefix-sum discrepancy:

$$\min_{\pi \in S_n} \quad \|\alpha - \beta\|_1$$

$$\text{where} \quad \alpha = \min_{1 \leq k \leq n} \left(\sum_{i=1}^{k} x_{\pi(i)} - \sum_{i=1}^{k} y_i \right) \in \mathbb{Z}^d$$

$$\beta = \max_{1 \leq k \leq n} \left(\sum_{i=1}^{k} x_{\pi(i)} - \sum_{i=1}^{k-1} y_i \right) \in \mathbb{Z}^d.$$

A different interpretation of the problem is: We are given two sequences X and Y of vectors, with the same total sum. We must find a permutation π of X such that, when we plot the polygonal-chain in $\mathbb{R}^d$ traced by the prefix-sums of $\pi(x_1) - y_1 + \pi(x_2) - y_2 + ... + \pi(x_n) - y_n$, the sum of the sidelengths of the box containing all those points is minimal (see Figure 10 for an example).

Even for $d = 1$, this problem is NP-hard (Newman et al. 2018), so approximation-algorithms have been studied instead:

Definition 2.4.2: Let $\mathcal{I}_d$ be the set of all d-dimensional instances of the gasoline-problem. For some instance $I \in \mathcal{I}_d$, let $\text{Opt}(I)$ be the value of an optimal solution, and $\mathcal{A}(I)$ be the value of the solution found by some algorithm $\mathcal{A}$. The **approximation-ratio of $\mathcal{A}$ in d dimensions** is:

$$\rho_{\mathscr{A}}^{(d)} \quad := \quad \sup_{I \in \mathscr{I}_d} \frac{\mathscr{A}(I)}{\mathrm{Opt}(I)}.$$

For $d = 1$, an algorithm with approximation-ratio 2 exists (Newman et al. 2018). For general d, a different algorithm exists, based on iterative rounding, for which we first write the gasoline-problem as an ILP. Let $\mathbb{1}$ be a vector of appropriate dimensions whose entries consist only of 1s.

$$\begin{aligned}
\min_{Z,\alpha,\beta} \quad & \|\alpha - \beta\|_1 \\
\text{s.t} \quad & \alpha \leq \sum_{i=1}^{k} Zx_i - \sum_{i=1}^{k} y_i, \quad k = 1, \ldots, n \\
& \beta \geq \sum_{i=1}^{k} Zx_i - \sum_{i=1}^{k-1} y_i, \quad k = 1, \ldots, n \\
& \mathbb{1}^T Z \leq \mathbb{1}^T, \quad Z^T \mathbb{1} \leq \mathbb{1} \qquad (\text{"}Z \text{ is a permutation-matrix"}) \\
& Z \in \{0,1\}^{d \times d} \\
& \alpha, \beta \in \mathbb{R}^d.
\end{aligned}$$

Program 1: The integer linear program for the generalised gasoline-problem.

Because $\beta \geq \alpha$, the objective "$\|\alpha - \beta\|_1$" is the same as "$\mathbb{1}^T(\beta - \alpha)$", so this is indeed a linear objective.

Algorithm 2: Iterative-Rounding for the Gasoline-Problem

```
Initialise UnfixedRows := {1, ..., n}. This keeps track of which rows of Z we did not fix to integral values yet.
Initialise LP as the LP-relaxation of Program 1, i.e. replace the constraint Z ∈ {0,1}^{d×d} with the constraint Z ∈ [0,1]^{d×d}.
For ColumnIndex = 1, ..., n:
    Initialise BestRowIndex := −1 and BestRowValue := ∞
    For RowIndex ∈ UnfixedRows:
        Let LP′ be the program LP with the added constraint "Z_{RowIndex, ColumnIndex} = 1".
        Let RowValue be the optimum value of the LP′.
        If RowValue < BestRowValue:
            BestRowIndex := RowIndex and BestRowValue := RowValue.
    Permanently add the constraint "Z_{BestRowIndex, ColumnIndex} = 1" to LP.
    Remove BestRowIndex from UnfixedRows.
UnfixedRows is empty and Z is fixed entirely. Return the permutation described by Z.
```

In this work, we are interested in finding lower bounds on the approximation-ratio $\rho_{\mathrm{IterRound}}^{(d)}$ of Algorithm 2. It holds that $\rho_{\mathrm{IterRound}}^{(1)} \leq \rho_{\mathrm{IterRound}}^{(2)} \leq \ldots$, because embedding a d-dimensional instance into $\mathbb{R}^{d+1}$ in the canonical way ($\mathbb{R}^d \ni x \mapsto (x, 0)$) yields a $(d+1)$-dimensional instance with the same IterRound- and Opt-values (see Lorieau (2024, Section 3.2)).

The permutation π in Example 2.4.1 is the output of Algorithm 2 on that instance. There, $\mathrm{IterRound}(I) = 24$, whereas $\mathrm{Opt}(I) = 20$, which shows $\rho_{\mathrm{IterRound}}^{(2)} \geq 1.2$. Lorieau (2024) constructed a sequence of

instances in $I_1, I_2, \ldots \subseteq \mathcal{I}_1$ for which $\text{IterRound}(I_j) / \text{Opt}(I_j)$ converged to a value of at least 2, proving that $\rho^{(1)}_{\text{IterRound}} \geq 2$.

Rajković (2022) conjectured that $\rho^{(1)}_{\text{IterRound}} = 2$, and $\rho^{(d)}_{\text{IterRound}} = 2$ for any $d > 1$. Though we will not make progress on the first conjecture, we did manage to disprove the second conjecture using an instance found by FunSearch. We also provide empirical data that weakly suggests $\rho^{(d)}_{\text{IterRound}} \geq \Omega(d)$. see Section 4.4 for details.

3 FunSearch

Making progress on the different problems introduced in Section 2 involves a similar task for all of them: We would like to find instances that have some problem-specific undesirable quality:

- For bin-packing, we would like to find an instance I where, if I is shuffled randomly, the Best-Fit algorithm performs, in expectation, poorly compared to an optimum solution:

$$\text{Score}(I) \quad = \quad \mathbb{E}_{\pi \in S_{|I|}}\left[\frac{\text{BestFit}(\pi(I))}{\text{Opt}(I)}\right].$$

- For the Pareto-sets of knapsack-instances, we would like to find an instance I where an intermittent Pareto-set $P(I_{1:i})$ is much larger than the Pareto-set $P(I)$ of the whole instance:

$$\text{Score}(I) \quad = \quad \frac{\max_{1 \le i \le |I|} |P(I_{1:i})|}{|P(I)|}$$

- For the Price of Hierarchy for k-median clustering, we would like to find an instance I where the optimal hierarchical clustering has a high approximation-factor:

$$\text{Score}(I) \quad = \quad \min_{H \in \mathscr{H}(I)} \text{Apx}_{\text{Cost}}(H) \quad = \quad \min_{H \in \mathscr{H}(I)}\left[\max_{i=1,\ldots,n} \frac{\text{Cost}(H_i)}{\text{Cost}(\text{Opt}_i)}\right]$$

- For the generalised gasoline problem, we would like to find an instance I where Algorithm 2 performs poorly compared to an optimum soution:

$$\text{Score}(I) \quad = \quad \frac{\text{IterRound}(I)}{\text{Opt}(I)}$$

Algorithm 3: Local Search for Instances Randomised Best-Fit Performs Poorly On

```
Fix the size n of an instance, e.g. n = 10.
Define the Score≈(I) of a bin-packing instance I:
    Calculate the value Opt of an optimum solution to I.
    Calculate 10000 trials of:
        Let I′ be a random permutation of I
        Run Best-Fit on I′
    Let Avg be the average number of bins used across these trials
    Return Avg / Opt
Define a Mutation(I) of an instance I:
    Define a new list of items I′ that arises from I by adding independently
    standard-normally distributed noise to each entry.
    Clamp the entries of I′ to be between 0 and 1.
    Return I′.
Initialise I as the list [1/2, ..., 1/2] of length n.
Repeat the following until some stopping-criterion is met:
    Calculate I′ = Mutation(I)
    If Score≈(I′) > Score≈(I):
        Replace I with I′
    Otherwise:
        Keep I unchanged.
```

3.1 Local Search

Even without having intuition for or experience with the different problems, we can still attempt to find such instances. A standard approach (see e.g. Stinson and Vanstone (1985); Parczyk et al. (2023), Lorieau (2024), or for a general overview Laarhoven and Aarts (1987)) is to employ some search-algorithm that searches for an instance of a high Score across the space of all instances. For bin-packing with capacity $c = 1$, Algorithm 3 is such an algorithm. It randomly changes the currently-best solution, checks whether this improves the objective, and declares this changed solution the new currently-best solution if it does.

Note that $\mathrm{Score}_{\approx}$ is only an approximation of the actual $\mathrm{Score}(I) = \mathbb{E}_{\pi \in S_{|I|}}\left[\frac{\mathrm{BestFit}(\pi(I))}{\mathrm{Opt}(I)}\right]$, as the exact Score can be intractable to compute due to the size of the symmetric group. It is also a (pseudo-)random variable, though this can be avoided by using a fixed seed for the evaluation. In our experiments, a number of 10000 calculations for $\mathrm{Score}_{\approx}$ was large enough to lead to accurate estimates, while still being small enough to compute quickly.

Variants of Algorithm 3 include decreasing the mutation-rate over time, e.g. by decreasing the noise's variance in $\mathrm{Mutation}$, or stochastically allowing to replace I with I' even if I' has a worse score, to prevent getting stuck in local optima. See Figure 11 for trajectories drawn from Algorithm 3.

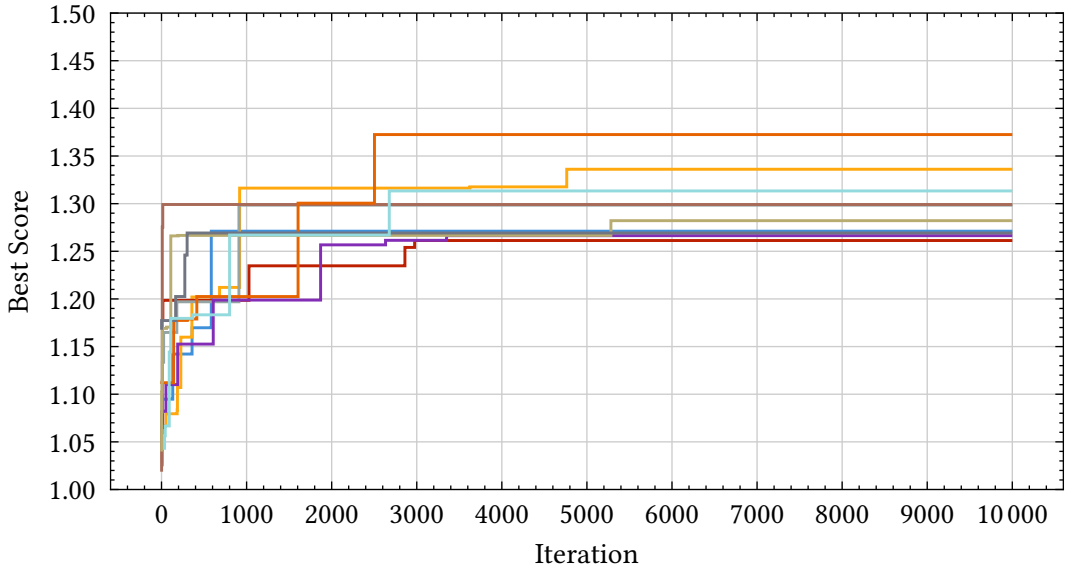


Figure 11: Ten example trajectories of Algorithm 3, terminating Line 14 after 10000 iterations. For each of the ten trajectories, we plot the score of the best solution I over time.

Enterprising readers will remember from Section 2.1 that the best-known instance for randomised Best-Fit had a score of 1.3, which the results from Figure 11 seem to beat ($\mathrm{Score}_{\approx}$ is only an estimate of Score, so this is not certain), as the best trial achieved a score of 1.3725. *If* we wanted to prove this rigorously, we would calculate the true score by running best-fit for all $10! \approx 3.6 \cdot 10^6$ permutations, possibly exploiting symmetries in the instance. We will not do so, however, in favour of proving a better result later on. Instead, to motivate our next steps, we will try learning from the instance, perhaps spotting structures in it, hoping to use these to manually construct instances of even higher scores. Alas, Figure 12 gives us little hope: Unlike e.g. the instance in Example 2.1.4, the instance found by Algorithm 3 does not seem to have any discernible pattern or noticeable symmetries. The four zero-weight items are a product of negative items in the mutation I' being rounded up to 0, and contribute nothing to the instance. Unable to learn from this instance, we will try a different approach.

```
[0, 0, 0, 0,
0.13941968656458636,
0.1415175313246237,
0.18488603733618258,
0.20818251654978343,
0.6014145332633378,
0.7129758245684663]
```

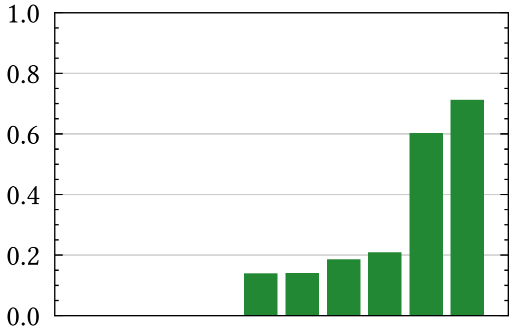


Figure 12: The sorted best instance found in the trials of Figure 11, achieving a score of 1.3725.

3.2 FunSearch: Local Search on Code Instead of Vectors

We could —instead of searching for lists of numbers— search for *short descriptions* of lists of numbers, i.e. we search for short *python-code* generating a list of numbers. While plain lists of numbers encode symmetric and structured instances just the same way as any other instances, python-code almost always produces symmetric and structured instances, assuming we avoid `import random` and hard-coding lists of numbers.

Example 3.2.1: An instance in the lower-bound construction given by Dósa and Sgall (2014) to prove that Best-Fit has approximation-ratio 1.7, can be expressed via these hardcoded numbers:

```
items = [
0.17166666666666665751889153668230392213445156812668, 0.16791666666666665744082898026334760288591496646404, 0.16697916666666665742131334115860852307378081604838,
0.16674479166666665741643443138242375312074727844447, 0.16668619791666665741521470393837756063248889404349, 0.16667154947916665741490977207736601251042429794325,
0.16666788736979165741483353911211312547990814891818, 0.16666697184244790741481448087079990372227911166192, 0.16666674296061196991480971631047159828287185234785,
0.16666668574015298553980852517038952192302003751934, 0.16666667143503823944605822738536900283305708381221, 0.16666666785875955292262065293911387306056634538542,
0.16666666696468988129176125932755009061744366077873, 0.16666666674117246338404641092465914500666298962706, 0.16666666668529310890711769882393640860396782183914,
0.32833333333333333147255159070298374812676227148475 5, 0.33208333333333331480357846344879380051615931651019, 0.33302083333333331482309410255353288032829346692585,
0.33325520833333331482797301232971765028132700452976, 0.33331380208333331482919273977376384276958538893074, 0.33332845052083331482949767163477539089164998503099,
0.33333211263020831482957390460002827792216613405605, 0.33333302815755206482959296284134149967979517131231, 0.33333325703938800232959772740166980511920243062638,
0.33333331425984698670459891854175188147905424545490, 0.33333332856496173279834921632677240056901719916203, 0.33333333214124041932178679077302753034150793758881,
0.33333333303531009095264618438459131278463062219550, 0.33333333325882750886036103278748225839541129334718, 0.33333333331470686333728974488820499479810646113510,
0.14666666666666665699847449388926179381087422370911, 0.16166666666666665731072471956508707080502063035965, 0.16541666666666665738878727598404339005355723202229,
0.16635416666666665740830291508878246986569138243794, 0.16658854166666665741318182486496723981872492004186, 0.16664713541666665741440155230901343230698330444284,
0.16666178385416665741470648417002498042904790054308, 0.16666544596354165741478271713527786745956404956814, 0.16666636149088540741480177537659108921719308682441,
0.16666659037272134491480653993691939465660034613848, 0.16666664759318032928980773107700147101645216096699, 0.16666666189829507538355802886202199010641511467412,
0.16666666547457376190699560330827711987890585310090, 0.16666666636864343353785499691984090232202853770760, 0.16666666659216085144556984532273184793280920885927,
0.35333333333333333152459329498228796095912000592651 3, 0.33833333333333331493368272414705433259705365261458, 0.33458333333333331485562016772809801334851705095195,
0.33364583333333331483610452862335893353638290053629, 0.33341145833333331483122561884717416358334936293237, 0.33335286458333331483000589140312797109509097853139,
0.33333821614583331482970095954211642297302638243115, 0.33333455403645831482962472657686353594251023340609, 0.33333363850911456482960566833555031418488119614982,
0.33333340962727862732960090377522200874547393683576, 0.33333335240681964295459971263513993238562212200724, 0.33333333810170489686084941485011941329565916830011,
0.33333333452542621033741184040386428352316842987333, 0.33333333363135653870655244679230050108004574526663, 0.33333333340783912079883759838940955546926507411496,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526,
0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526, 0.50000000000000000001694065894508600713401475526]
```

Listing 1: The lower-bound instance in Dósa and Sgall (2014) for $k = 3$.

However, they actually defined the instance more like this:

```python
k = 3
OPT = 10*k
δ = 1/50
d = lambda j: δ/(4**j)
ε = d(10*k + 5)

b_plus   = [1/6 + d(j)     for j in range(1, 1+OPT//2)]
c_minus  = [1/3 - d(j) - ε for j in range(1, 1+OPT//2)]
b_minus  = [1/6 - d(j)     for j in range(0,   OPT//2)]
c_plus   = [1/3 + d(j) - ε for j in range(0,   OPT//2)]
trailing = [1/2 + ε] * OPT

items = b_plus + c_minus + b_minus + c_plus + trailing
```

Listing 2: The same instance as in Listing 1, using a more structured definition.

For larger k, Listing 1 grows even longer, while Listing 2 remains short and interpretable.

However, if we now tried to implement Algorithm 3 by searching on the space of python-code instead of the space $\mathbb{R}^{10}$, we will have trouble defining a $\mathrm{Mutation}$-function, which is meant to return a mutated variant of our current solution. Defining $\mathrm{Mutation}$ by throwing noise onto the python-code (e.g. randomly change or swap characters) like we did for $\mathbb{R}^{10}$ would lead to most mutated programs failing to compile. One can try circumventing this by not interpreting python-code as a sequence of characters, but as a composition-tree of basic computational functions, an approach known as *Genetic Programming* (Koza 1994; Petersen 2019; Cranmer 2023).

Instead of Genetic Programming, we will follow the approach of Romera-Paredes et al. (2024) called **FunSearch**. Instead of mutating python-code by randomly changing characters, it mutates python-code by querying a large language model (LLM). An example for such a query is shown in Listing 3, and an example-response in Listing 4. The advantage of this method is that we retain both interpretable structure, and python-code that compiles most of the time.

Furthermore (though this was not done in the shown examples), the python-code can be generalised on some sets of parameters. For instance, the `get_items` functions could accept an integer-parameter `length` that tells the function the maximum allowed size of the list. Our evaluation-function Score then rejects lists exceeding the provided `length` by returning a score of 0, and we may mathematically analyse the asymptotic behaviour of the function for large `length` after the fact. To make generalisation across different values of `length` more likely, we could also evaluate the function on multiple different values and take a weighted average.

I'm trying to find instances of the bin-packing problem where, if the input is shuffled, the best-fit online-heuristic performs poorly in expectation. All bins have capacity 1.0.

To generate instances that best-fit performs poorly on, I have tried the following functions so far. Please write another one that returns an instance and is similar, but has some lines altered.

```python
import math


def get_items_v0() -> list[float]:
    """Return a new bin-packing-instance, specified by the list of items.

    The items must be floats between 0 and 1.
    """
    """Generate a bin-packing-instance similar to v0 but with some modifications to induce worse performance after shuffling."""
    items = []
    for i in range(1, 11):
        if i % 3 == 0:
            items.append(0.7)  # add larger items that challenge best-fit
        else:
            items.append(1.0 / i)
    # add some small items to increase total items and create fragmentation
    items.extend([0.05, 0.05, 0.05])
    return items


def get_items_v1() -> list[float]:
    """Improved version of `get_items_v0`."""
    """Return a new bin-packing-instance, similar to get_items_v0,
    but with some lines altered to create input that challenges best-fit heuristics.
    """
    items = []
    for i in range(2, 11):
        items.append(1.0 / i + 0.05)
    # Add some larger items to cause early packed bins
    items.extend([0.6, 0.55])
    # Add smaller items to increase mix complexity
    for i in range(15, 20):
        items.append(1.0 / i)
    return items
```

Listing 3: An example prompt given to the LLM, along with the system-prompt "`You are a helpful coding assistant who only responds with python-code and no markdown-formatting.`"

```python
def get_items_v2() -> list[float]:
    """Return a bin-packing-instance similar to previous ones but with some lines altered to cause worse performance after shuffling."""
    items = []
    # Replace small fractional items with slightly larger ones to increase fragmentation
    for i in range(1, 9):
        if i % 2 == 0:
            items.append(0.4 + 0.02 * i)  # slightly larger items that challenge best-fit
        else:
            items.append(1.0 / (i + 3))
    # Include some large items to force inefficient packing
    items.extend([0.65, 0.63])
    # Add very small items close to 0.02 to fill gaps after shuffling
    items.extend([0.02, 0.02, 0.02])
    return items
```

Listing 4: A response to the prompt in Listing 3. The responding LLM was `gpt-4.1-nano` *with a temperature-parameter of* 1.2.

3.3 Tuning FunSearch's Output

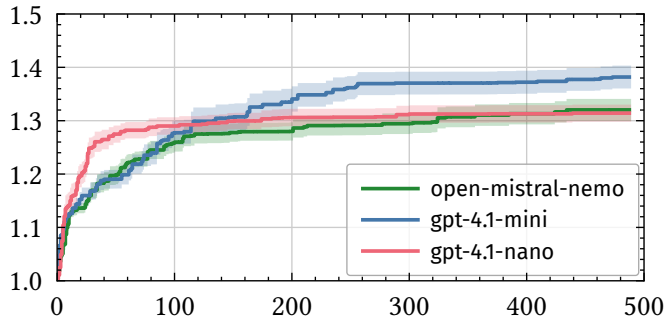
We used FunSearch to find "bad" instances for the four problems listed above. After FunSearch concluded, we manually searched through its output for promising code. We then manipulated the code, for instance by removing redundant items (see e.g. Listing 5 and Listing 12) or making the instance more symmetrical (see Listing 12, where we replaced `np.linspace`, which produces a sequence of evenly-spaced numbers, with `np.ones`, which produces a sequence of identical numbers), and checking after every step that the program's score didn't decrease noticeably.

Not all programs found by FunSearch lend themselves to this. Some programs just produced pseudo-random numbers, e.g. using trigonometric functions. If tuning was successfull, though, we ended up with a concise, interpretable, symmetric instance that we could use to try and prove new results.

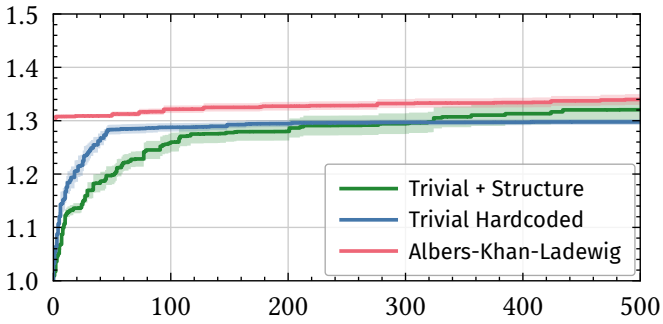
3.4 Ablations

When using FunSearch, one must make decisions about certain hyper-parameters, including:

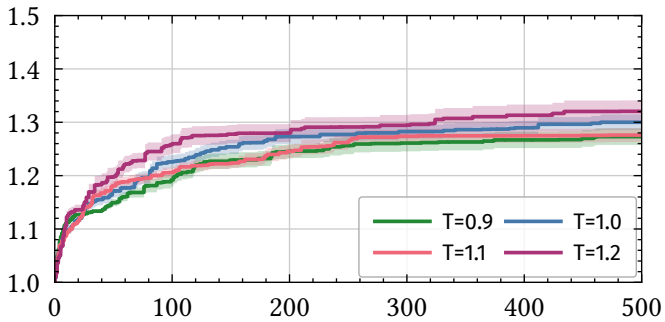
- The large lanugage model (LLM).
- The LLM's temperature-parameter T. In local-search, one must decide how to choose a random neighbour v' of v, e.g. v' arises from adding normally-distributed vector to v. The temperature-parameter in FunSearch is similar to the standard-deviation of local-search's normal distribution, i.e. low temperatures will lead to the distribution of LLM-responses to be highly concentrated, and high temperatures spread out the distribution.
- The initial program to start with. This is usually either some trivial hardcoded instance, a hardcoded instance with additional program-structure (like for-loops), or the state of the art.



(a) Variations across LLMs, each with $T = 1.0$ and starting with a hardcoded instance.



(b) Variations across initial programs, each with gpt-4.1-mini and $T = 1.0$.



(c) Variations across temperatures for gpt-4.1-mini, starting with a hard-coded instance.

Figure 13: Ablations for different hyper-parameters on the bin-packing problem. For each choice, we ran 25 trials of FunSearch, tracking the running maximum score for each trial. We then plot the average of these running maxima, together with their standard error, across time (number of LLM-samples).

We ran ablations (Figure 13) across these hyper-parameters on the bin-packing problem by measuring the average maximum scores across different trials. This is not a great measure, because we do not

actually care about getting a high average score. We care about finding structured, generalizable instances, but we found no good way of measuring that, so the high average score serves as a proxy.

- gpt-4.1-mini seems to outperform gpt-4.1-nano (a smaller model) and open-mistral-nemo.
- When starting with the state-of-the-art instance by Albers et al. (2021, Lemma 4.3), the score starts off at 1.3, and improves slightly over time. Starting with trivial instances performs slightly worse. Encouraging structure in the initial instance by using for-loops leads to better long-term performance. When just starting with a hardcoded list of numbers without structure, FunSearch would frequently stick to just changing the numbers instead of introducing more structure.
- Higher temperatures seem to be better.

3.5 Implementation Details

Our implementation[1] is a fork of Johannes Aalto's implementation[2], which is a fork of Google DeepMind's repository[3]. We replaced the single-threaded evaluation-loop (query the LLM to get one new program, evaluate the program, repeat) with a multi-threaded producer-consumer pattern, where multiple queries are made in parallel, and evaluated asynchronously across different threads. Furthermore, each query is batched, producing several new programs (default: 4) instead of just one, which is more cost-effective as the input-tokens are only billed once per batch.

We also created an interface to display results about FunSearch runs in the form of a website[4]. This helped with collaboration, analysing the outcomes and benchmarking different choices of parameters.

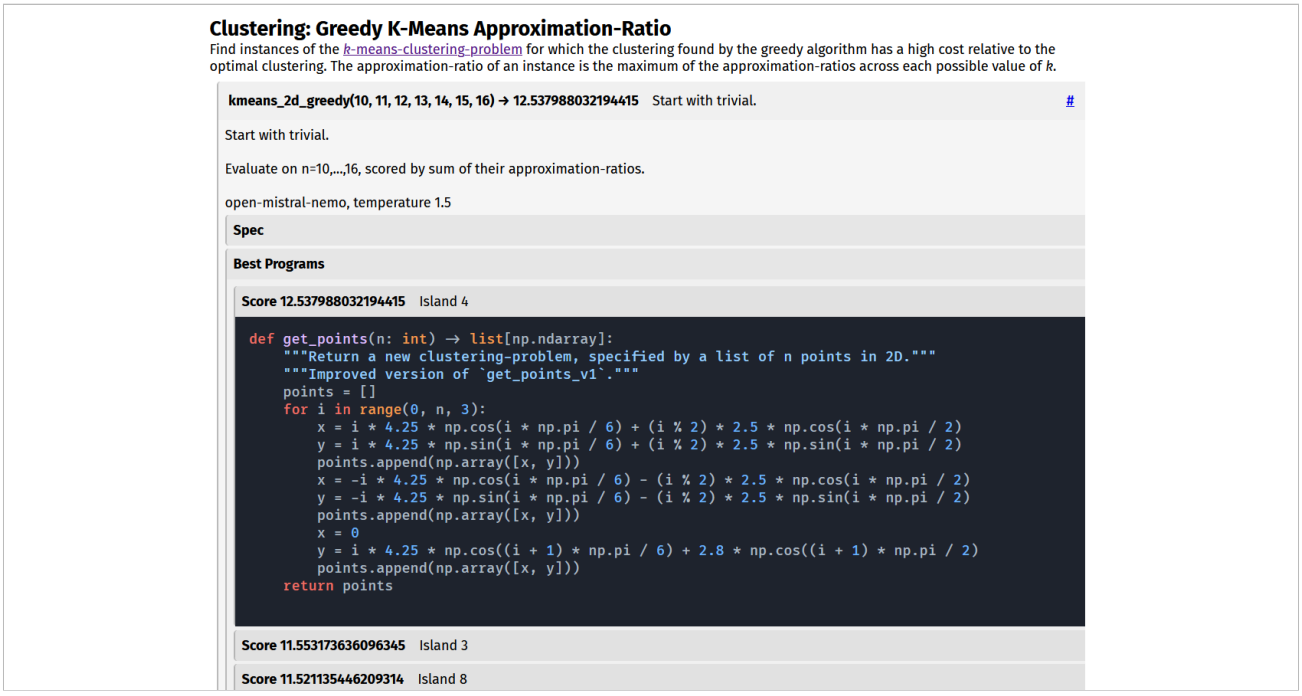


Figure 14: The website showing outcomes of FunSearch runs.

When a query returns a program, it is evaluated by assigning it a score (higher being better). These scores were problem-specific.

3.5.1 Scoring Bin-Packing

For a bin-packing instance I, we calculated the optimal (smallest) number of bins $\mathrm{Opt}(I)$ by calling an

[1] github.com/lumi-a/funsearch
[2] github.com/jonppe/funsearch
[3] github.com/google-deepmind/funsearch
[4] lumi-a.github.io/funsearch

existing solver `packingsolver`[5] (Fontan and Libralesso 2020), which is based on column-generation.

We did not calculate the expected number of bins used by Best-Fit ($\mathbb{E}_{\pi \in S_{|I|}}[\mathrm{BestFit}(\pi(I))]$) exactly, but instead ran 10000 trials of Best-Fit under random permutations, and used the mean number of bins Avg as an estimate.

The score assigned to I was $\mathrm{Avg} \,/\, \mathrm{Opt}(I)$.

3.5.2 Scoring Knapsack

We implemented Algorithm 1 in the way described in Röglin (2020, Theorem 5), but using multi-sets for the sets $\mathrm{val}(P_i)$ in order to accurately track the true size of the Pareto-Set, and not just the size of the deduplicated Pareto-Sets.

For a knapsack-instance I, we run this implementation of Algorithm 1 and keep track of the largest Pareto-Set P_{largest} and the running maximum of $\left|P_{\mathrm{largest}}\right|/|P_i|$ over time. The final score is this maximum. That is, the assigned score is:

$$\mathrm{Score}(I) \quad = \quad \max_{i=1,\ldots,|I|} \left[\max_{1 \leq j \leq i} \frac{|P_j|}{|P_i|} \right]$$

3.5.3 Scoring Hierarchical Clustering

The score of an instance is the approximation-factor of an *optimal* hierarchical clustering in the sense of Definition 2.3.2. We did not find any existing solver for this, and brute-force methods are intractable: The number of different hierarchical clusterings on 32 points is around $1.78 \cdot 10^{42}$. At first, we attempted to formulate the problem as an ILP to be solved with Gurobi (Gurobi Optimization, LLC 2024), but that also proved ineffective, so we wrote our own solver instead.

For every $k = 1, \ldots, n$, the solver first computes an approximate k-clustering via heuristics ($k$means++ or agglomerative clustering), providing an upper bound on the value of an optimal k-clustering. An actual optimum is then computed via branch-and-bound: Let $P = [p_1, \ldots, p_n]$ be the list of vertices. We keep a priority-queue storing partial clusterings, i.e. partitions of some initial sub-list $[p_1, \ldots, p_i]$. The neighbors of such a partial clustering are all possible partial clusterings obtained by adding p_{i+1} to either an existing clustering, or putting it into a new singleton-cluster (if the total number of clusters is smaller than k). At each step, we take the partial clustering with the lowest priority off the priority-queue and add its neighbours to the priority-queue, where their priority is simply the total cost of that partial clustering, filtering out neighbours whose priority exceeds the upper bound on the optimal clustering (although the clustering is only partial, its value can only increase if more points are added).

After computing an optimal k-clustering for each k, we compute *some* hierarchical clustering via agglomerative clustering, and then an optimal hierarchical clustering, again via branch-and-bound, but this time proceeding level-wise from level $k = n$ to level $k = 1$ (if we started from $k = 2$, we would immediately have 2^n neighbours to inspect). This time, the priority of a partial hierarchical clustering is the maximum of $\frac{\mathrm{Cost}(H_i)}{\mathrm{Opt}_i}$ over all levels H_i that have been clustered so far (compare Definition 2.3.2).

To speed up the search, we used memoization for computing costs of individual clusters, efficient memory-representation via bit-vectors and allocating on the stack as much as possible. With this, we were able to compute optimal hierarchical clusterings on 32 points.

[5] github.com/fontanf/packingsolver

Written in rust, it is available on crates.io[6] with documentation on docs.rs[7], the repository is on GitHub[8]. We also provide python-bindings (on PyPi[9], GitHub[10]) via Maturin. The code is heavily benchmarked, tested, and documented, so that other researchers may easily use it.

3.5.4 Scoring Gasoline

An instance I was scored by its approximation-ratio $\mathrm{IterRound}(I)/\mathrm{Opt}(I)$, for which we could simply use the code[11] by Lorieau (2024), specifically $\mathrm{Score}(I) =$ `iterative_rounding.SlotOrdered().run(I)`. This solver calls Gurobi (Gurobi Optimization, LLC 2024) to calculate an optimal permutation.

[6] crates.io/crates/exact-clustering
[7] docs.rs/exact-clustering
[8] github.com/lumi-a/exact-clustering
[9] pypi.org/project/exact-clustering
[10] github.com/lumi-a/py-exact-clustering
[11] github.com/ath4nase/gasoline

4 Results

4.1 Bin-Packing

We started with a trivial hardcoded instance (score 1.0), and FunSearch soon found an instance with score 1.49815. The theoretical upper bound is 1.5, and the instance had an extremely simple structure.

```python
def get_items() -> list[float]:
    """Return a new bin-packing-instance, specified by the list of items.

    The items must be floats between 0 and 1."""
    items = [0.4, 0.5, 0.6]
    return items
```

(a) Initial program.

```python
def get_items() -> list[float]:
    a = 7
    b = 5
    return [1.0 / a] * a + [1.0 / b] * b
```

(b) After tuning Listing 5c by hand.

```python
def get_items() -> list[float]:
    """Return a new bin-packing-instance, specified by the list of items.

    The items must be floats between 0 and 1."""
    """Yet another version of `get_items_v0`, `get_items_v1`, and `get_items_v2`, with some lines altered."""
    items = [0.8, 0.2, 0.6, 0.4]
    # Split the first item into seven smaller items and the fourth item into five smaller items
    items = [0.114, 0.114, 0.114, 0.114, 0.114, 0.114, 0.114] + items[1:3] + [0.08, 0.08, 0.08, 0.08, 0.08]
    return items
```

(c) A program found by FunSearch after 10 trials of $2,400$ samples each.

Listing 5: The evolution of programs generating bin packing instances, with model open-mistral-nemo and a temperature of 1.5.

Further experimentation with the instance in Listing 5b indicated that, for the instance to have a high score, the constants `a` and `b` should be large and coprime. So for fixed $m \in \mathbb{N}$, consider the instance:

$$I := \Big[\underbrace{m+1,\ \ldots,\ m+1}_{m \text{ times}},\ \underbrace{m,\ \ldots,\ m}_{m+1 \text{ times}}\Big], \qquad \text{maximum bin capacity } \ c := m \cdot (m+1).$$

An optimal packing puts the first m items into one bin, and the remaining $m+1$ items into a second bin. This fills both bins exactly to their maximum capacity.

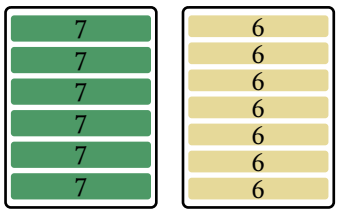


Figure 15: An optimal packing for $m = 6$, using two bins. The bins have capacity $c = m \cdot (m+1) = 42$.

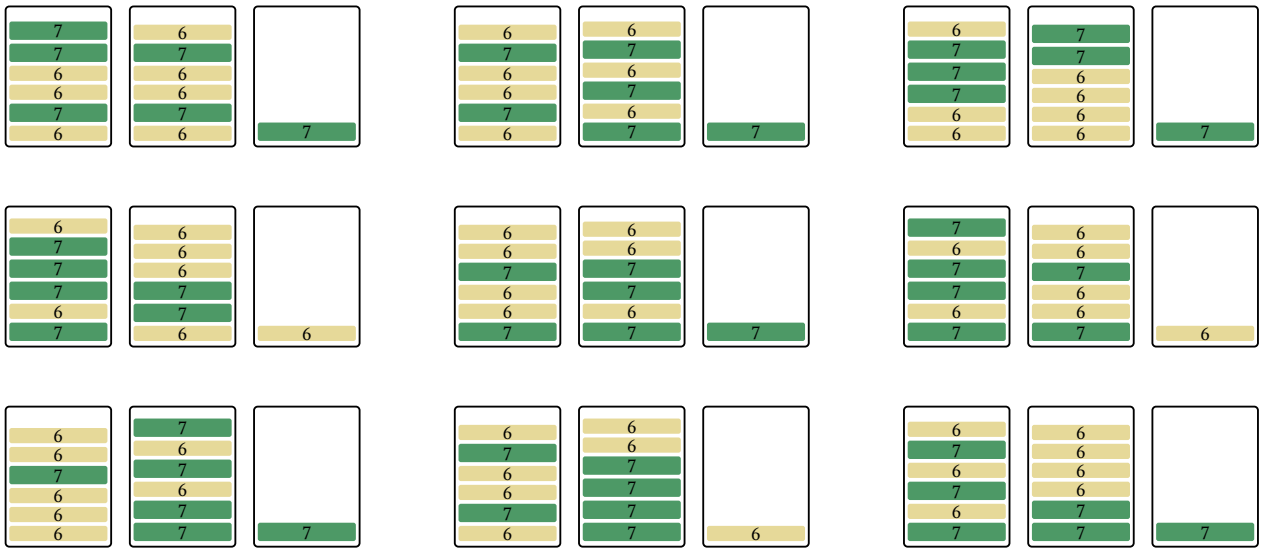


Figure 16: Nine different packings produced by randomised Best-Fit. These were not cherry-picked.

It turns out that this is the *only* optimal packing (up to re-labeling the two bins):

Lemma 4.1.1: An optimal packing can not have a bin that contains both an item of weight m and an item of weight $m+1$.

Proof: Every optimal packing must fill both bins exactly to their full capacity c, because the sum of all items is $2c$. Assume, for contradiction, a bin contains $0 < i < m$ items of weight m and $0 < j < m$ items of weight $m+1$:

$$(m+1) \cdot m \quad = \quad c \quad = \quad im + j(m+1)$$

Rearranged:

$$(m+1-i) \cdot m = j \cdot (m+1)$$

Because m and $m+1$ are coprime, their least common multiple is $m(m+1)$, so j must be either 0 or m, contradicting $0 < j < m$. □

Hence, if any bin contains both an item m and an item $m+1$, the packing must use at least 3 bins. Because the instance is shuffled, Best-Fit will put both an item of size m and an item of size $m+1$ into the same bin with high probability:

Lemma 4.1.2: Randomised Best-Fit returns an optimal packing with probability $\leq \frac{2}{m+2}$.

Proof:

- If the first item has weight m, then for Best-Fit to find the optimal solution, the next $m-1$ items must have weight m, as well. The probability of this happening is:

$$\frac{m}{2m} \cdot \frac{m-1}{2m-1} \cdot \ldots \cdot \frac{2}{m+2} \quad \leq \quad \frac{2}{m+2}.$$

- If the first item has weight $m+1$, then the next $m-1$ items must have weight m, as well. The probability of this happening is:

$$\frac{m-1}{2m} \cdot \frac{m-2}{2m-1} \cdot \ldots \cdot \frac{1}{m+1} \quad \leq \quad \frac{2}{m+2}.$$

□

With more effort, one could obtain tighter bounds on the probability. But that simply will not be necessary, as this weak bound already yields a sufficient lower-bound on the absolute random-order-ratio:

$$\mathrm{RR}_{\mathrm{BestFit}} \quad = \quad \sup_{I' \in \mathscr{I}} \mathbb{E}_{\pi \in S_{|I'|}} \left[\frac{\mathrm{BestFit}(\pi(I'))}{\mathrm{Opt}(I')} \right] \quad \geq \quad \frac{1}{2} \cdot \left[2 \cdot \frac{2}{m+2} + 3 \cdot \frac{m}{m+2} \right] \quad = \quad \frac{3}{2} - \frac{1}{m+2}.$$

For $m \to \infty$, this shows $\mathrm{RR}_{\mathrm{BestFit}} \geq 1.5$, and the upper bound $\mathrm{RR}_{\mathrm{BestFit}} \leq 1.5$ by Kenyon (1996) implies:

Theorem 4.1.3: The absolute random-order-ratio of Best-Fit $\mathrm{RR}_{\mathrm{BestFit}}$ is exactly 1.5.

4.2 Knapsack Problem

Our implementation of the scoring-function initially contained a bug, which under-estimated the size of some Pareto-sets. Unaware of this, we still let FunSearch run its course, and nevertheless obtained the following result.

```python
def get_instance() -> list[tuple[int, int]]:
    """Return an instance, specified by the list of
(weight, profit) pairs.

    Weights and profits must be non-negative integers.
    """
    return [(1, 2)] * 2 + [(4, 4), (2, 2), (1, 3)]
```

(a) Initial program.
Bugged score $= 1.25$, Actual score $= 1.0$.

```python
def get_instance() -> list[tuple[int, int]]:
    items = []
    n = 7
    items += [(1, 1)] * n
    items += [(4, 9)] * n
    items += [(2, 5), (3, 7)]
    return items
```

(b) Step in tuning Listing 6c by hand.
Bugged score = Actual score ≈ 12.64.

```python
def get_instance() -> list[tuple[int, int]]:
    """Create a variant with more diverse item types and weights to potentially influence Pareto set size."""
    items = []
    # Repeated very light, low profit items
    items += [(1, 1)] * 8
    # Mix of moderate weight and profit items with some unique entries
    items += [(4, 9), (4, 9), (5, 10)]
    # High-profit, lightweight items with more profit variation
    items += [(2, 16), (2, 14), (3, 15)]
    # Heavier items with varied weights and higher profits to increase trade-offs
    items += [(9, 20), (12, 30), (15, 40)]
    # Small, low to moderate profit items
    items += [(1, 3), (2, 5), (3, 7), (3, 8)]
    # Very heavy, high-profit rare items with similar weights
    items += [(20, 35), (21, 36), (22, 38)]
    # Larger weight, moderate profit item to diversify options
    items += [(18, 28)]
    # Additional medium-weight high-profit items to increase complexity
    items += [(10, 25), (11, 27)]
    return items
```

(c) A program found by FunSearch after 10 trials of 500 samples each.
Bugged score = Actual score ≈ 5.766.

Listing 6: The evolution of programs generating knapsack-instances using a bugged scoring-function, with model gpt-4.1-nano and a temperature of 1.0.

We also re-ran FunSearch with the fixed scoring-function, but did not manage to recover the same instance, see Section 4.2.1 for details.

The scores of the following instances were unaffected by the bug. After simplifying the output into Listing 6b, we continued tuning and scaled all items' weights up by a factor of 2 (which does not affect Pareto-optimality), decreased some profits by 1, and changed the last item to obtain the following tidier instance, which achieves slightly higher scores for the same n:

$$\left[\underbrace{\binom{8}{8},\ldots,\binom{8}{8}}_{n\text{ times}},\ \underbrace{\binom{2}{1},\ldots,\binom{2}{1}}_{n\text{ times}},\ \binom{4}{4},\binom{2}{2}\right].$$

From here, we attempted to prove results about the instance. After a first draft, we found it more natural to replace the first n items by n powers of 2, and saw that stronger results are possible by replacing the last two items by k powers of 2:

$$\left[\binom{2^{2k}}{2^{2k}},\binom{2^{2k+1}}{2^{2k+1}},\ldots,\binom{2^{2k+n}}{2^{2k+n}},\ \underbrace{\binom{2^k}{2^k-1},\ldots,\binom{2^k}{2^k-1}}_{n\text{ times}},\ \binom{2^{2k-1}}{2^{2k-1}},\binom{2^{2k-2}}{2^{2k-2}},\ldots,\binom{2^{k+1}}{2^{k+1}}\right].$$

Finally, we appended the factors $x_{i,k} := \left(1+\frac{2^{-i}}{2^k-1}\right)$ to the n center items, which ensured that $P_{\text{dedup}}(I_{1:i}) = P(I_i)$ for all i.

$$\text{Items} = \left[\binom{2^{2k}}{2^{2k}},\ldots,\binom{2^{2k+n}}{2^{2k+n}},\ \binom{x_{1,k}\cdot 2^k}{x_{1,k}\cdot\left(2^k-1\right)},\ldots,\binom{x_{n,k}\cdot 2^k}{x_{n,k}\cdot\left(2^k-1\right)},\ \binom{2^{2k-1}}{2^{2k-1}},\ldots,\binom{2^{k+1}}{2^{k+1}}\right]. \tag{1}$$

We will now analyze the sizes of the instance's and subinstances' Pareto-sets. To that end, define the two segments of the instance: For $a,b,d,n\in\mathbb{Z}_{\geq 1}$ with $d<a\leq b$, consider two lists:

$$I_{a,b} := \left[\binom{2^a}{2^a},\binom{2^{a+1}}{2^{a+1}},\ldots,\binom{2^b}{2^b}\right],\qquad J_{d,n} := \left[\binom{x_{1,d}\cdot 2^d}{x_{1,d}\cdot\left(2^d-1\right)},\ldots,\binom{x_{n,d}\cdot 2^d}{x_{n,d}\cdot\left(2^d-1\right)}\right].$$

Lemma 4.2.1: If a Pareto-optimal packing $A\in P\left(\left[I_{a,b},J_{d,n}\right]\right)$ does not contain all items from $I_{a,b}$, it contains fewer than 2^{a-d} items from $J_{d,n}$.

Proof: Subsets of $I_{a,b}$ can be represented by binary numbers of $(b-a+1)$ bits. If A does not contain all items from $I_{a,b}$ and contains at least 2^{a-d} items from $J_{d,n}$, we define a new packing A' as follows: Increment the binary number representing $A\cap I_{a,b}$ by 1, and remove 2^{a-d} items from $A\cap J_{d,n}$. This changes the weights and profits by:

$$\begin{aligned}
\text{Weight}(A')-\text{Weight}(A) &\leq 2^a-2^{a-d}\cdot\underbrace{\left(1+\frac{2^{-n}}{2^d-1}\right)}_{>1}\cdot 2^d &&< 0\\
\text{Profit}(A')-\text{Profit}(A) &\geq 2^a-2^{a-d}\cdot\left(1+\frac{2^{-1}}{2^d-1}\right)\left(2^d-1\right)\\
&= 2^a-2^{a-d}\cdot\left(2^d-2^{-1}\right) &&= 2^{a-d-1} > 0
\end{aligned}$$

Thus, A' dominates A, and $A\notin P\left(\left[I_{a,c},J_{d,n}\right]\right)$. □

On the other hand, all other packings are Pareto-optimal:

Lemma 4.2.2: If a packing A of $[I_{a,b}, J_{d,n}]$ contains all items from $I_{a,b}$ or contains fewer than 2^{a-d} items from $J_{d,n}$, then A is Pareto-optimal.

Proof: All items from $I_{a,b}$ have a profit-per-weight ratio of 1, while all items from $J_{d,n}$ have a profit-per-weight ratio of $\frac{2^d-1}{2^d} < 1$. Hence, a packing B that dominates A must satisfy

$$\text{Weight}(A \cap I_{a,b}) \quad < \quad \text{Weight}(B \cap I_{a,b}),$$

otherwise B can not have enough profit to dominate A. If A already contains all items from $I_{a,b}$, this is not possible, so only the case that A contains fewer than 2^{a-d} items from $J_{d,n}$ remains. Due to the definition of $I_{a,b}$, the above inequality implies:

$$\text{Weight}(A \cap I_{a,b}) + 2^a \quad \leq \quad \text{Weight}(B \cap I_{a,b}).$$

If B dominates A, it must hold that:

$$\begin{aligned} \text{Weight}(A \cap I_{a,b}) + \text{Weight}(A \cap J_{d,n}) \quad &\geq \quad \text{Weight}(B \cap I_{a,b}) + \text{Weight}(B \cap J_{d,n}) \\ \implies \text{Weight}(A \cap J_{d,n}) - 2^a \quad &\geq \quad \text{Weight}(B \cap J_{d,n}).f \end{aligned}$$

But A contains fewer than 2^{a-d} items from $J_{d,n}$, so:

$$\begin{aligned} \text{Weight}(A \cap J_{d,n}) \quad &\leq \quad 2^{a-d} \cdot \left(1 + \frac{2^{-1}}{2^d - 1}\right) \cdot \left(2^d - 1\right) \quad = \quad 2^{a-d} \cdot \left(2^d - 2^{-1}\right) \\ &= \quad 2^a - 2^{a-d-1} \quad < \quad 2^a. \end{aligned}$$

This implies $0 > \text{Weight}(B \cap J_{d,n})$, a contradiction. □

Hence, we can describe the Pareto-set exactly:

$$P([I_{a,b}, J_{d,n}]) \quad = \quad \left\{A \cup B \;\middle|\; A \subsetneq I_{a,b},\ B \subseteq J_{d,n},\ |B| < 2^{a-d}\right\} \quad \dot\cup \quad \left\{I_{a,b} \cup B \;\middle|\; B \subseteq J_{d,n}\right\}.$$

Its size is exactly (using notation for binomial coefficients, not vectors):

$$|P([I_{a,b}, J_{d,n}])| = \left(2^{b-a+1} - 1\right) \cdot \left[\sum_{i=0}^{\min\left(n,\, 2^{a-d}-1\right)} \binom{n}{i}\right] + 2^n.$$

For $k, n \in \mathbb{N}$ with $2^k \leq n/2$, consider two instances (back to vectors, instead of binomial coefficients):

$$\begin{aligned} \mathbb{I}_1 &:= [I_{2k,\,2k+n}, J_{k,n}], \\ \mathbb{I}_2 &:= \left[\mathbb{I}_1, \binom{2^{k+1}}{2^{k+1}}, \binom{2^{k+2}}{2^{k+2}}, \ldots, \binom{2^{2k-1}}{2^{2k-1}}\right]. \end{aligned}$$

$\mathbb{I}_1$ is a sub-instance of $\mathbb{I}_2$. The instance $\mathbb{I}_2$ (which is exactly the instance in Equation 1) contains the same items as $[I_{k+1,\,2k+n}, J_{k,n}]$. The sizes of their Pareto-sets can be bounded by:

$$\begin{aligned} |P(\mathbb{I}_1)| \quad &\geq \quad \left(2^{n+1} - 1\right) \cdot \binom{n}{2^k - 1} + 2^n \quad \geq \quad \left(2^{n+1} - 1\right) \cdot \left(\frac{n}{2^k - 1}\right)^{\left(2^k - 1\right)} \\ |P(\mathbb{I}_2)| \quad &\leq \quad \left(2^{k+n} - 1\right) \cdot (n+1) + 2^n \quad \leq \quad \left(2^{k+n} - 1\right) \cdot (n+2). \end{aligned}$$

The ratio between the two sizes is:

$$\frac{|P(\mathbb{I}_1)|}{|P(\mathbb{I}_2)|} \geq \frac{2^{n+1}-1}{2^{k+n}-1} \cdot \left(\frac{n}{2^k-1}\right)^{\left(2^k-1\right)} \cdot \frac{1}{n+2}$$

For $k = \log_2(\sqrt{n}) + 1$, we obtain:

$$\frac{|P(\mathbb{I}_1)|}{|P(\mathbb{I}_2)|} \geq \frac{2^{n+1}-1}{\left(\sqrt{n}+1\right)\cdot 2^n - 1} \cdot \left(\frac{n}{\sqrt{n}}\right)^{\sqrt{n}} \cdot \frac{1}{n+2} = \theta\left(n^{\left(\sqrt{n}-3\right)/2}\right).$$

The length of the instance $\mathbb{I}_2$ is not n but $m := |\mathbb{I}_2| = 2n + k$, resulting in an actual lower bound of $\Omega\left((m/2)^{\left(\sqrt{m/2}-3\right)/2}\right)$.

Theorem 4.2.3: There exist instances I such that:

$$\text{Score}(I) = \frac{\max_{j=1,\ldots,|I|}|P(I_{1:j})|}{|P(I)|} \geq \Omega\left(\left(\frac{|I|}{2}\right)^{\left(\sqrt{|I|/2}-3\right)/2}\right).$$

The only purpose of the leading factors $x_{i,d} = \left(1 + \frac{2^{-i}}{2^d-1}\right)$ in $J_{d,n}$ is to prevent two Pareto-optimal packings from having the same total profit:

Lemma 4.2.4: If $A, B \subseteq [I_{a,b}, J_{d,n}]$ are two distinct Pareto optimal packings, then $\text{Profit}(A) \neq \text{Profit}(B)$.

Proof: Because both A and B are Pareto-optimal, we know by Lemma 4.2.1 that $|A \cap J_{d,n}| < 2^{a-d}$ (same for B), hence:

$$\begin{aligned}\text{Profit}(A \cap J_{d,n}) &< 2^{a-d} \cdot \left(1 + \left(2^{-1}\right)\left(2^d - 1\right)\right) \cdot \left(2^d - 1\right) \\ &= 2^{a-d} \cdot \left(2^d - \frac{1}{2}\right) \\ &= 2^a - 2^{a-d-1} < 2^a.\end{aligned}$$

(same for $\text{Profit}(B \cap J_{d,n})$).

- If $A \cap I_{a,b} \neq B \cap I_{a,b}$, the difference between $\text{Profit}(A \cap I_{a,b})$ and $\text{Profit}(B \cap I_{a,b})$ would be at least 2^a, due to the definition of $I_{a,b}$. In this case, the above inequality already shows $\text{Profit}(A) \neq \text{Profit}(B)$.
- If $A \cap I_{a,b} = B \cap I_{a,b}$, then $A \cap J_{d,n} \neq B \cap J_{d,n}$, and we need to show that $\text{Profit}(A \cap J_{d,n}) \neq \text{Profit}(B \cap J_{d,n})$. This is equivalent to showing that any two distinct subsets of:

  $$\left\{\left(2^d - 1\right) + 2^{-1},\ \left(2^d - 1\right) + 2^{-2},\ \ldots,\ \left(2^d - 1\right) + 2^{-n}\right\},$$

  have a distinct sum. This is true, because the total sum of the summands $2^{-1}, \ldots, 2^{-n}$ is always smaller than 1, whereas $2^d - 1 \geq 1$.

□

Thus, the deduplicated Pareto-Sets and Pareto-Sets of $\mathbb{I}_2$ coincide, meaning Theorem 4.2.3 applies to P_{dedup} as well, which disproves that Algorithm 1 runs in polynomial time.

4.2.1 FunSearch With Corrected Scoring-Function

After finding and eliminating the bug in our scoring-function, we re-ran FunSearch. This time, the result had a score exceeding 600, *before* being tuned by hand. This was done after we had already proven the above result.

```python
def get_instance() -> list[tuple[int, int]]:
    """Return an instance, specified by the list of
(weight, profit) pairs.

    Weights and profits must be non-negative integers.
    """
    return [(1, 2)] * 2 + [(4, 4), (2, 2), (1, 3)]
```

(a) Initial program, score $= 1.25$.

```python
def get_instance() -> list[tuple[int, int]]:
    items = [(x, x + 25) for x in list(range(25, 60))] * 3
    items += [(1 + 2 * i, 1 + 3 * i) for i in range(1, 8)]
    return items
```

(b) After tuning Listing 7c, score $\approx 74\,612\,418$.

```python
def get_instance_0() -> list[tuple[int, int]]:
    """New instance designed to highlight a larger Pareto-set in a subinstance."""
    items = [
        (60, 85), (57, 82), (54, 79), (51, 76), (48, 73),
        (45, 70), (42, 67), (39, 64), (36, 61), (33, 58),
        (56, 81), (53, 78), (50, 75), (47, 72), (44, 69),
        (41, 66), (38, 63), (35, 60), (32, 57), (29, 54),
        (55, 80), (52, 77), (49, 74), (46, 71),
        (43, 68), (40, 65),  # Weight adjustments for more trade-offs
        (27, 42), (22, 37),  # Front-loaded higher profit high-weight items
        (11, 18),  # Small diverse item
        (14, 24), (17, 28),
        (12, 19), (16, 25), (20, 32),
        (8, 14), (7, 10), (13, 17)
    ] + [(15, 21), (18, 27), (21, 33)]
    return items
```

(c) A program found by FunSearch after 30 trials of 1000 samples each, score ≈ 646.

Listing 7: The evolution of programs generating knapsack-instances using a correct scoring-function, with model gpt-4.1-nano and a temperature of 1.2.

While this instance is concise, its structure is still messy. We neither saw a way of turning Listing 7b into the instance in Equation 1, nor did we see how we could use Listing 7b to obtain a result similar to the above theorem.

4.2.2 An Exponential Bound

After finishing the above work, we came up with a simpler construction. Although we were working on FunSearch during that time, this construction was not found by a FunSearch trial. Let I again be a list of items, given as weight-profit tuples.

- Let $\oplus$ denote list-concatenation, $[1,2] \oplus [3,4] = [1,2,3,4]$. This is associative but not commutative.
- For some scalar $\alpha \in \mathbb{R}_{>0}$, let $\alpha I := \left[\binom{\alpha \cdot w}{\alpha \cdot p} \,\middle|\, \binom{w}{p} \in I \right]$.
- Let $\|I\| := \left\| \sum_{x \in I} x \right\|_\infty + 1$.
- Call I **integral** iff the weights and profits of all its items are integral.

Example 4.2.5: Let $I := \left[\binom{1}{2}, \binom{2}{1}, \binom{3}{3} \right]$. Here, $\|I\| = 7$. Note the fractal-like structure of the following plots.

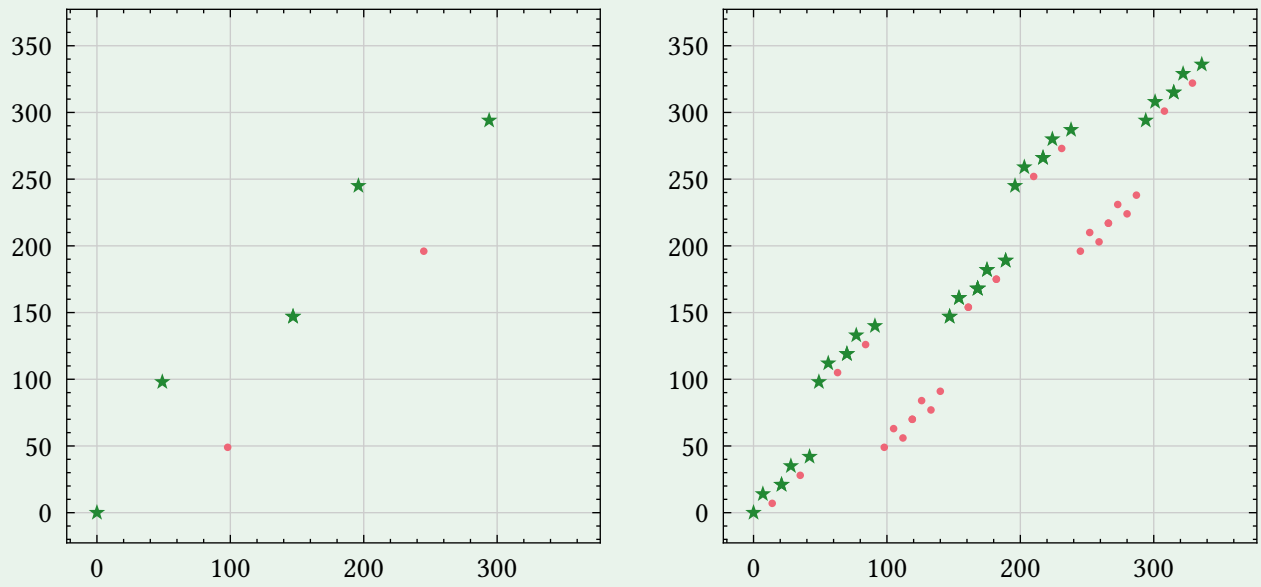


Figure 17: Plotting (Weight, Profit) *for all solutions of* $7I$ *(left), and* $I \oplus 7I$ *(right). There are* 6 *(left) and* 6^2 *(right) Pareto-optimal solutions, they are marked with a* $\star$.

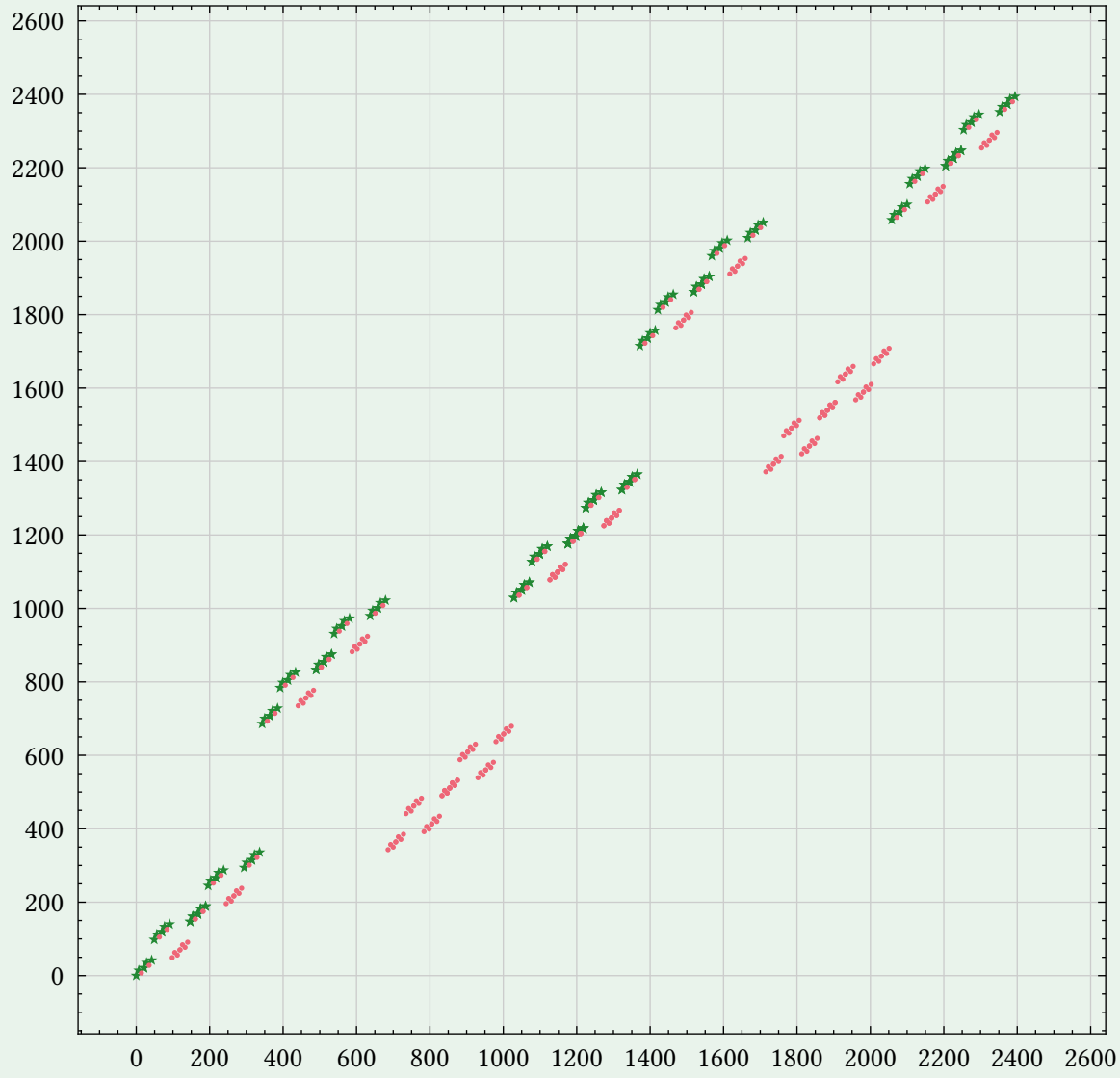


Figure 18: Plotting (Weight, Profit) *for all solutions of* $I \oplus (7I) \oplus (7^2 I)$. *There are* 6^3 *Pareto-optimal solutions, they are marked with a* $\star$.

This example illustrates how the size of $I \oplus \|I\| I$ is twice the size of I, but $|P(I \oplus \|I\| I)| = |P(I)|^2$. Indeed, this holds more generally:

Lemma 4.2.6: Let A and B be instances of the Knapsack-problem, let B be integral, and $\alpha \geq \|A\|$. Let $L \coloneqq L_A \oplus \alpha L_B$ be a sublist of $A \oplus \alpha B$, where L_A is a sublist of A and L_B is a sublist of B. The following are equivalent:

(1) $L \in P(A \oplus \alpha B)$,

(2) $L_A \in P(A)$ and $L_B \in P(B)$.

In other words, $P(A \oplus \alpha B) = \left[S_A \oplus \alpha S_B \,\middle|\, S_A \in P(A), S_B \in P(B)\right]$.

Proof:

- $\neg(2) \Rightarrow \neg(1)$: If (2) is false, one of the following must hold:
  - There is a sub-list $D_A \subseteq A$ such that D_A dominates L_A. Then $D_A \oplus \alpha L_B$ dominates L.
  - There is a sub-list $D_B \subseteq B$ such that D_B dominates L_B. Then $L_A \oplus \alpha D_B$ dominates L.
- $\neg(1) \Rightarrow \neg(2)$: If (1) is false, there exist sublists $D_A \subseteq A$ and $D_B \subseteq B$ such that $D \coloneqq D_A \oplus \alpha D_B$ dominates L. We only consider the case "$\text{Weight}(L) > \text{Weight}(D)$ and $\text{Profit}(L) \leq \text{Profit}(D)$", the other case is analogous. It follows that:

$$\begin{aligned} \text{Weight}(L_A) + \alpha \cdot \text{Weight}(L_B) &> \text{Weight}(D_A) + \alpha \cdot \text{Weight}(D_B) \\ \text{Profit}(L_A) + \alpha \cdot \text{Profit}(L_B) &\leq \text{Profit}(D_A) + \alpha \cdot \text{Profit}(D_B) \end{aligned}$$

  From this, it follows that $\text{Weight}(L_B) \geq \text{Weight}(D_B)$. Otherwise, $1 \leq \text{Weight}(D_B) - \text{Weight}(L_B)$, because we assumed B to be integral, and thus $\text{Weight}(L_A) - \text{Weight}(D_A) \geq \alpha \geq \|A\|$, which is impossible due to the definition of $\|A\|$.
  The same argument can be used to show $\text{Profit}(L_B) \leq \text{Profit}(D_B)$.

  Now, distinguish two cases:
  - If $\text{Weight}(L_B) > \text{Weight}(D_B)$ or $\text{Profit}(L_B) < \text{Profit}(D_B)$, then D_B dominates L_B, so $L_B \notin P(B)$.
  - If $\text{Weight}(L_B) = \text{Weight}(D_B)$ and $\text{Profit}(L_B) = \text{Profit}(D_B)$, the above inequalities imply:

$$\begin{aligned} \text{Weight}(L_A) &> \text{Weight}(D_A) \\ \text{Profit}(L_A) &\leq \text{Profit}(D_A), \end{aligned}$$

    so D_A dominates L_A, hence $L_A \notin P(A)$.

□

In particular, under the assumptions of the Lemma, $|P(A \oplus \alpha B)| = |P(A)| \cdot |P(B)|$.

Recall that $\text{Score}(I) \coloneqq \frac{\max_{1 \leq i \leq n} |P(I_{1:i})|}{|P(I)|}$. Let I be some integral instance, and $J \coloneqq \arg\max_{1 \leq i \leq n} |P(I_{1:i})|$. For some $k \in \mathbb{N}$, consider the following two instances:

$$I^k \coloneqq I \oplus \left(\alpha^1 I\right) \oplus \left(\alpha^2 I\right) \oplus \ldots \oplus \left(\alpha^{k-1} I\right),$$

$$J^k \coloneqq J \oplus \left(\alpha^1 J\right) \oplus \left(\alpha^2 J\right) \oplus \ldots \oplus \left(\alpha^{k-1} J\right).$$

With $A \coloneqq I \oplus \left(\alpha^1 I\right) \oplus \left(\alpha^2 I\right) \oplus \ldots \oplus \left(\alpha^{k-2} I\right)$ and $B \coloneqq \alpha^{k-1} I$, briefly verify the assumptions of Lemma 4.2.6:

$$\|A\| \quad = \quad 1 + (\alpha - 1) + \alpha^1(\alpha - 1) + \ldots + \alpha^{k-2}(\alpha - 1) \quad = \quad \alpha^{k-1}$$

By induction, we have $\left|P\left(I^k\right)\right| = |P(I)|^k$ and $\left|P\left(J^k\right)\right| = |P(J)|^k$, hence $\text{Score}\left(I^k\right) \geq \frac{|P(I)|^k}{|P(J)|^k} = \text{Score}(I)^k$, which grows exponentially. However, the length of I^k is $k \cdot |I|$, growing linearly.

Theorem 4.2.7: For any integral instance I:

$$\text{Score}\left(I^k\right) \quad \geq \quad \text{Score}(I)^k \quad = \quad \left(\text{Score}(I)^{1/|I|}\right)^{|I^k|}.$$

For example, for the instance in Example 2.2.4, $\text{Score}(I) = 1.2$ and $|I| = 5$, so:

$$\text{Score}\left(I^k\right) \geq \left(\sqrt[5]{1.2}\right)^{|I^k|} \approx 1.0371^{|I^k|}.$$

This bound is exponential, obsoleting Theorem 4.2.3. If we found instances J where $\text{Score}(I)^{1/|I|}$ were larger than 1.0371, we would obtain an even higher base. As mentioned before, the pareto-set $P(J)$ always has size at most $2^{|J|}$, hence 2 is an upper bound on this base.

The statement of Lemma 4.2.6 also applies to the deduplicated Pareto-sets. We only formulate the statement in terms of cardinalities, because equivalence-classes cause troubles otherwise (what's the concatenation "$\oplus$" of two equivalence-classes $A, B \in P_{\text{dedup}}(J)$?).

Lemma 4.2.8: If A and B are instances of the Knapsack-problem, B integral, and $\alpha \geq \|A\|$, then:

$$\left|P_{\text{dedup}}(A \oplus \alpha B)\right| \quad = \quad \left|P_{\text{dedup}}(A)\right| \cdot \left|P_{\text{dedup}}(B)\right|.$$

Proof: Let $L := L_A \oplus \alpha L_B$ and $\tilde{L} := \tilde{L}_A \oplus \alpha \tilde{L}_B$ be two solutions. If $\text{Weight}(L) = \text{Weight}\left(\tilde{L}\right)$, then $\text{Weight}(L_A) = \text{Weight}\left(\tilde{L}_A\right)$ and $\text{Weight}(L_B) = \text{Weight}\left(\tilde{L}_B\right)$, because α is so large. The same applies to the profits. Now, apply Lemma 4.2.6. □

To get the same result for the runtime of Algorithm 1, define:

$$\text{Score}_{\text{dedup}}(I) := \frac{\max_{1 \leq i \leq n} |P_{\text{dedup}}(I_{1:i})|}{|P_{\text{dedup}}(I)|}$$

By the same reasoning as for Theorem 4.2.7, we get:

Theorem 4.2.9: For any integral instance I:

$$\text{Score}_{\text{dedup}}\left(I^k\right) \quad \geq \quad \text{Score}_{\text{dedup}}(I)^k \quad = \quad \left(\text{Score}_{\text{dedup}}(I)^{1/|I|}\right)^{|I^k|}.$$

For example, for the instance in Example 2.2.4, $\text{Score}_{\text{dedup}}(I) = 1.125$ and $|I| = 5$, so:

$$\text{Score}_{\text{dedup}}\left(I^k\right) \geq \left(\sqrt[5]{1.125}\right)^{|I^k|} \approx 1.0238^{|I^k|}.$$

As before, finding instances J with $\text{Score}(I)^{1/|I|} \geq 1.0238$ would yield even higher bases, and the base is upper-bounded by 2.

4.2.3 Finding Better Bases

Naturally, we set out to find such instances via FunSearch. We used the scoring-function:

$$\overline{\text{Score}}(I) = \max_{i=1,\ldots,|I|} \left(\max_{1 \leq j \leq i} \frac{|P_j|}{|P_i|} \right)^{1/i} \quad \text{instead of the former} \quad \text{Score}(I) = \max_{i=1,\ldots,|I|} \left(\max_{1 \leq j \leq i} \frac{|P_j|}{|P_i|} \right) \quad \text{from Section 3.5.2.}$$

This time, we used a recursive approach: We started with a trivial instance $I^{(0)}$, ran FunSearch to obtain some instance $F^{(0)}$, tuned it into an instance $T^{(0)}$, and then used $T^{(0)} =: I^{(1)}$ as the starting-point of another FunSearch trial, producing $F^{(1)}$, and so on, up to $T^{(2)}$. When giving values of scores here, we round them down.

```
items = [(4, 4)] * 2 + [(2, 1), (1, 2)] + [(2, 2)]
```

(a) $I^{(0)}, \overline{\text{Score}}\left(I^{(0)}\right) \approx 1.037$

```
# Mix of items with near-identical profit-to-weight ratios to introduce a complex Pareto front
items = [
    (50, 100), (49, 99), (48, 98), (47, 97), (46, 96),
    (45, 95), (44, 94), (43, 93), (42, 92), (41, 91),
    (40, 90), (39, 89), (38, 88), (37, 87), (36, 86)
] * 2

# Include smaller items with similar ratios
items += [(7, 14), (6, 12), (5, 10)]
# Add some mid-weight, mid-value items likely excluded in larger sets
items += [(20, 40), (18, 36), (16, 32)]
# Slightly improve some items to create overlapping efficiency
items += [(22, 44), (19, 38)]
```

(b) $F^{(0)}, \overline{\text{Score}}\left(F^{(0)}\right) \approx 1.334$

```
items = [(42 + i, 92 + i) for i in range(15)] * 2

items += [(5 + i, 10 + 2 * i) for i in range(3)]
items += [(16 + 2 * i, 32 + 4 * i) for i in range(3)]
```

(c) $T^{(0)} = I^{(1)}, \overline{\text{Score}}\left(T^{(0)}\right) \approx 1.357$

```
items = [(50 + i, 100 + i) for i in range(12)] * 3  # Created a different range and multipliers

items += [(3 + i, 6 + 2 * i) for i in range(4)]     # Fewer items, different pattern
items += [(20 + 3 * i, 40 + 5 * i) for i in range(4)]  # Larger weights and profits
```

(d) $F^{(1)}, \overline{\text{Score}}\left(F^{(1)}\right) \approx 1.389$

```
items = [(50 + i, 100 + i) for i in range(12)] * 4
items += [(3 + i, 6 + 2 * i) for i in range(4)]
items += [(20 + 3 * i, 40 + 5 * i) for i in range(4)]
```

(e) $T^{(1)} = I^{(2)}, \overline{\text{Score}}\left(T^{(1)}\right) \approx 1.415$

```
items = [(60 + i, 110 + i) for i in range(12)] * 4
items += [(5 + i, 8 + 2 * i) for i in range(4)]
items += [(25 + 3 * i, 45 + 4 * i) for i in range(4)]
```

(f) $F^{(2)}, \overline{\text{Score}}\left(F^{(2)}\right) \approx 1.456$

```
items = [(61 + i, 110 + i) for i in range(11)] * 5
items += [(5 + i, 8 + 2 * i) for i in range(5)]
items += [(25 + 3 * i, 44 + 4 * i) for i in range(3)]
```

(g) $T^{(2)}, \overline{\text{Score}}\left(T^{(2)}\right) \approx 1.480$

Listing 8: Incrementally tuning instances via FunSearch and by hand.

Finally, we also ran local search on instanec $T^{(2)}$, trying to improve the score even further, and obtained:

```
[(62, 111), (62, 111), (64, 113), (64, 113), (65, 114), (66, 115), (67, 116), (68, 117), (69, 118), (70, 119), (71, 120), (61, 110), (64, 113), (63, 112), (66, 115), (65, 114), (66, 115), (67, 116), (66, 115), (69, 118), (70, 119), (69, 118), (64, 113), (63, 112), (63, 112), (65, 114), (65, 114), (66, 115), (67, 116), (68, 117), (68, 117), (70, 119), (71, 120), (62, 111), (62, 111), (63, 112), (65, 114), (67, 116), (66, 115), (67, 116), (68, 117), (68, 117), (70, 119), (69, 118), (61, 110), (63, 112), (64, 113), (64, 113), (65, 114), (66, 115), (67, 116), (68, 117), (69, 118), (70, 119), (71, 120), (4, 7), (5, 9), (7, 12), (9, 15), (7, 13), (26, 45), (29, 50), (31, 52)]
```

Listing 9: The result of running local search on $T^{(2)}$ for 2200 generations. The $\overline{\text{Score}}$ of this instance is ≈ 1.509.

Combined with Theorem 4.2.7, we get:

Corollary 4.2.10: For every $n \in \mathbb{N}$, there exists an instance I of length $\geq n$ such that:

$$\text{Score}(I) \quad = \quad \frac{\max_{1 \leq i \leq n} |P(I_{1:i})|}{|P(I)|} \quad \geq \quad 1.509^n.$$

This is the best bound we found. As the base is upper-bounded by 2, it would be interesting to know what the best-possible base is.

We did the same for $\text{Score}_{\text{dedup}}$, and mark instances with tildes here, to prevent confusion.

```
items = [(4, 4)] * 2 + [(2, 1), (1, 2)] + [(2, 2)]
```

(a) $\tilde{I}^{(0)}, \overline{\text{Score}}_{\text{dedup}}\left(\tilde{I}^{(0)}\right) \approx 1.024$

```
# Introduce a different combination to potentially reduce the pareto set size
items = [(9, 10)] * 4 + [(3, 8), (2, 7), (1, 6)] + [(6, 6)]
```

(b) $\tilde{F}^{(0)}, \overline{\text{Score}}_{\text{dedup}}\left(\tilde{F}^{(0)}\right) \approx 1.046$

We were unable to tune $\tilde{F}^{(0)}$ in a meaningful way. Local search was a lot faster for such a short instance, but did not improve the score significantly more.

```
[(14, 11), (9, 10), (9, 10), (9, 10), (6, 2), (2, 7), (1, 6), (4, 6)]
```

Listing 11: The result of running local search on $\tilde{F}^{(0)}$ for 660000 generations. The $\overline{\text{Score}}_{\text{dedup}}$ of this instance is ≈ 1.060.

Combined with Theorem 4.2.9, we obtain::

Corollary 4.2.11: For every $n \in \mathbb{N}$, there exists an instance I of length $\geq n$ such that:

$$\text{Score}_{\text{dedup}}(I) \quad = \quad \frac{\max_{1 \leq i \leq n} |P_{\text{dedup}}(I_{1:i})|}{|P_{\text{dedup}}(I)|} \quad \geq \quad 1.06^n.$$

In particular, the worst-case runtime of the Nemhauser-Ullmann algorithm on an input of length n is lower-bounded by $\Omega(1.06^n)$.

4.3 k-Median Clustering

We evaluated weighted instances, as unweighted trials usually ended up putting many points in the same places, and evaluating corresponding weighted instances instead would be much more efficient. Going from Listing 12c to Listing 12b, we replaced `np.linspace` with `np.ones`, which yields a more symmetric instance.

```python
def get_weighted_points() -> list[tuple[float,
np.ndarray]]:
    """Return a new weighted clustering-problem, specified
by a list of weighted points.
    The returned tuple consists of the weight of the
point, and the point itself."""
    weighted_points = [(1.0, np.array([0, 0, 0, 0])),
(1e8, np.array([1, 0, 0, 0]))]
    return weighted_points
```

(a) Initial program.

```python
def get_weighted_points() -> list[tuple[float,
np.ndarray]]:
    return [
        (1.0, np.zeros(14)),
        *[(1.0, -np.eye(14)[i]) for i in range(14)],
        (1e10, np.ones(14) / 20),
    ]
```

(b) After tuning Listing 12c by hand.

```python
def get_weighted_points() -> list[tuple[float, np.ndarray]]:
    """Return a new weighted clustering-problem, specified by a list of weighted points.
    The returned tuple consists of the weight of the point, and the point itself."""
    return [
        (1.0, np.zeros(14)),
        (1e10, np.ones(14)),
        *[(1.0, np.eye(14)[i]) for i in range(7)],
        *[(1.0, np.eye(14)[i]*-1) for i in range(7, 13)],
        *[(1e10-i*1e9, np.linspace(i*0.1, (i+1)*0.1, 14, endpoint=False)) for i in range(7)],
        (1e11, np.array([13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0])),
        (1e12, np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])),
        (1e13, np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])*10),
        (1e14, np.array([14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1])*100),
        (1e15, np.array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])*1000),
```

(c) A program found by FunSearch after 10 trials of $2,200$ samples each.

Listing 12: The evolution of programs generating clustering-instances. The model used was open-mistral-nemo with a temperature of 1.5.

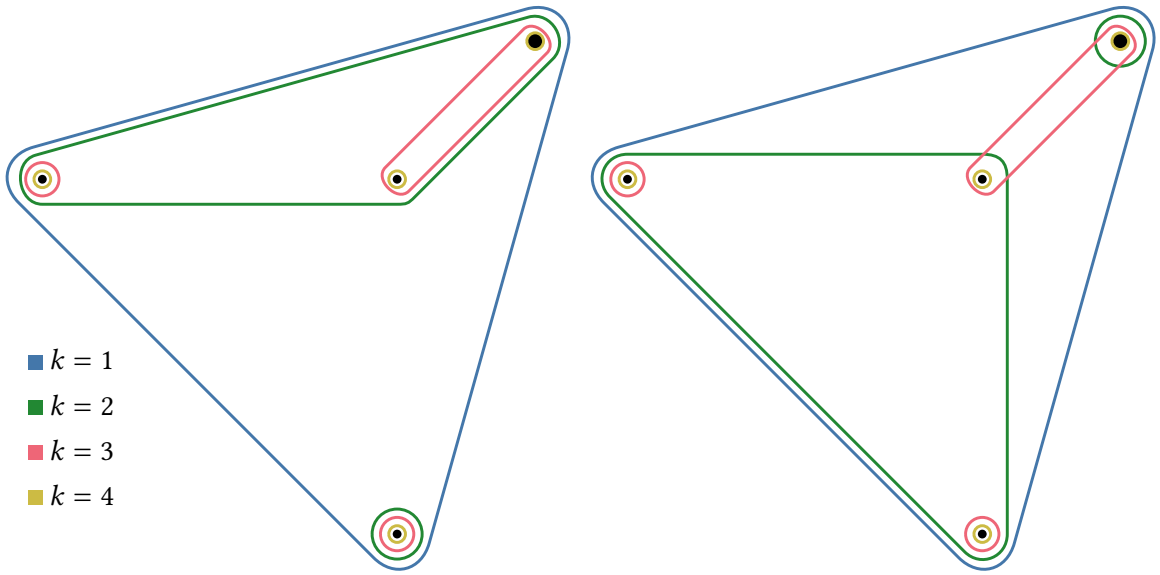


Figure 19: We only defined instances for $d \geq 4$, but this is a depiction of the same instance for $d = 2$ and $c = 2.57$. The large point in the upper right has weight ∞, the others have weight 1. Left: An optimal hierarchical clustering, having approximation-factor ≈ 1.278. Right: Optimal clusterings for each k.

Fix the dimension $d \geq 4$. Put $c := \frac{\sqrt{4d^2+(3-d)^2}+d-3}{2}$, which is one of the two roots of $0 = c^2 - c(d-3) - d^2$. Because $d \geq 4$, we know that $5d^2 - 6d \geq 4d^2$, hence:

$$c = \frac{\sqrt{4d^2 + (d-3)^2} + d - 3}{2} > \frac{2d + d - 3}{2} > d.$$

Let e_i be the ith d-dimensional standard basis vector. Consider the following weighted instance of $d + 2$ points:

$$(1, ..., 1), \quad (0, ..., 0), \quad -ce_1, ..., -ce_d,$$

where the point $(1, ..., 1)$ has weight ∞ and all other points have weight 1.

Lemma 4.3.1: For k-median clustering, this instance's price of hierarchy is at least $\frac{c}{d}$.

Proof: For contradiction, assume there exists a hierarchical clustering $H = (H_1, ..., H_{d+2})$ such that, on every level, the cost of H_k is strictly less than $\frac{c}{d}$ times the cost of the best clustering using k clusters. This enables us to narrow down the structure of H:

- For $k = d + 1$, there is one cluster C containing two points, while all other clusters contain only a single point. Depending on which two points constitute C, we can calculate the total cost of the clustering:
  - If $C = \{(0, ..., 0), (1, ..., 1)\}$, the total cost is:
    $$\|(0, ..., 0) - (1, ..., 1)\|_1 = d.$$
  - If $C = \{(0, ..., 0), -ce_i\}$ for some i, the total cost is c.
  - If $C = \{(1, ..., 1), -ce_i\}$ for some i, the total cost is $d + c$.
  - If $C = \{-ce_i, -ce_j\}$ for some $i \neq j$, the total cost is $2c$.

  Because $d < c$, this constrains H_k to $C = \{(0, ..., 0), (1, ..., 1)\}$, otherwise the total cost of H_k would be at least $\frac{c}{d}$ times the cost of an optimal $(d + 1)$-clustering.
- For $k = 2$: The clustering now contains exactly two clusters. Because H is a hierarchical clustering, we now know that H_2 has a cluster that contains $(0, ..., 0)$, $(1, ..., 1)$ and some number $0 \leq n \leq d - 1$ of the $-ce_i$, while its other cluster contains the remaining $d - 1 - n$ of the $-ce_i$. Due to symmetry, this number n is sufficient for calculating the total cost of H_2. Because $(1, ..., 1)$ has infinite weight, this point must be the center of the first cluster, so this cluster has cost:
  $$\|(1, ..., 1) - (0, ..., 0)\| + n \cdot \|(1, ..., 1) - (-ce_1)\|_1 = d + n \cdot (c + d)$$
  The cluster containing the remaining $d - 1 - n$ of the $-ce_i$ can choose any point as its center. It has cost:
  $$(d - 2 - n) \cdot \|ce_1 - ce_2\|_1 = (d - 2 - n) \cdot 2c$$
  Given n, the total cost of H_2 is $d + c(2d - 4) + n(d - c)$. Because $d - c < 0$, the best choice for n would be $n = d - 1$, resulting in a cost of $c(d - 3) + d^2$. This is only a lower bound on the cost of H_2, because other levels in the hierarchy might put additional constraints on H_2.

  For an *upper* bound on the *optimal* cost of a 2-clustering, consider the clustering that has $(1, ..., 1)$ in its first cluster, and all other points in its second cluster. Assuming the center of the second cluster is $(0, ..., 0)$, we get an upper bound on the total cost of this clustering of:
  $$d \cdot \|(0, ..., 0) - (-ce_1)\|_1 = d \cdot c.$$
  Hence, the ratio between the cost of H_2 and the cost of an optimal 2-clustering is at least:
  $$\frac{c(d - 3) + d^2}{d \cdot c} = \frac{d - 3}{d} + \frac{d}{c}$$

> We defined c as one of the roots of $0 = c^2 - c(d-3) - d^2$. Dividing out cd, we get $\frac{d-3}{d} + \frac{d}{c} = \frac{c}{d}$. However, this contradicts the assumption that the ratio between H_2 and an optimal 2-clustering is strictly less than $\frac{c}{d}$.

Thus, the instance's price of hierarchy is at least $\frac{c}{d}$. □

For large d, this fraction $\frac{c}{d} = \frac{\sqrt{4d^2+(3-d)^2}+d-3}{2d}$ converges to $\frac{1+\sqrt{5}}{2}$, the golden ratio.

Theorem 4.3.2: The Price of Hierarchy for k-median clustering $\mathrm{PoH}_{k\text{-median}}$ is at least $\frac{1+\sqrt{5}}{2} \approx 1.618$.

4.4 Gasoline

The following example is the instance found by Lorieau (2024):

Example 4.4.1: This is a d=1-dimensional instance. Fix some $k \in \mathbb{N}$. For any i, define $u_i := 2^k(1 - 2^{-i})$. Let $\oplus$ denote list-concatenation, e.g. $[1,2] \oplus [3,4] = [1,2,3,4]$. The 1-dimensional instance found by Lorieau (2024) can be written as follows:

$$X = \left(\bigoplus_{i=1}^{k-1} \bigoplus_{1}^{2^i} [u_i]\right) \oplus \left(\bigoplus_{1}^{2^k-1} [2^k]\right) \oplus [0], \qquad Y = \bigoplus_{i=1}^{k} \bigoplus_{1}^{2^i} [u_i].$$

We consider the case $k = 5$. This instance has 62 items, which is too large to write out in full. We plot the permutation-matrices for some optimal solution π_{Opt} and the solution $\pi_{\mathrm{IterRound}}$ found by Algorithm 2. Filled squares represent a 1, empty squares a 0.

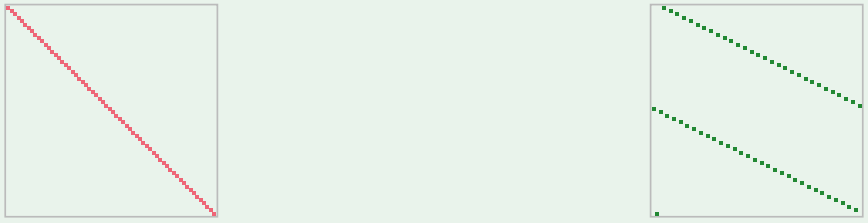


Figure 20: The permutation-matrices for $\pi_{\mathrm{IterRound}}$ (left) and some π_{Opt} (right).

Indeed, $\pi_{\mathrm{IterRound}}$ is the identity. We also plot the progression of BestRowValue over the course of Algorithm 2 for this instance. These values are the result of minimisation-LPs whose set of constraints grows over time, so this plot must be non-decreasing over time.

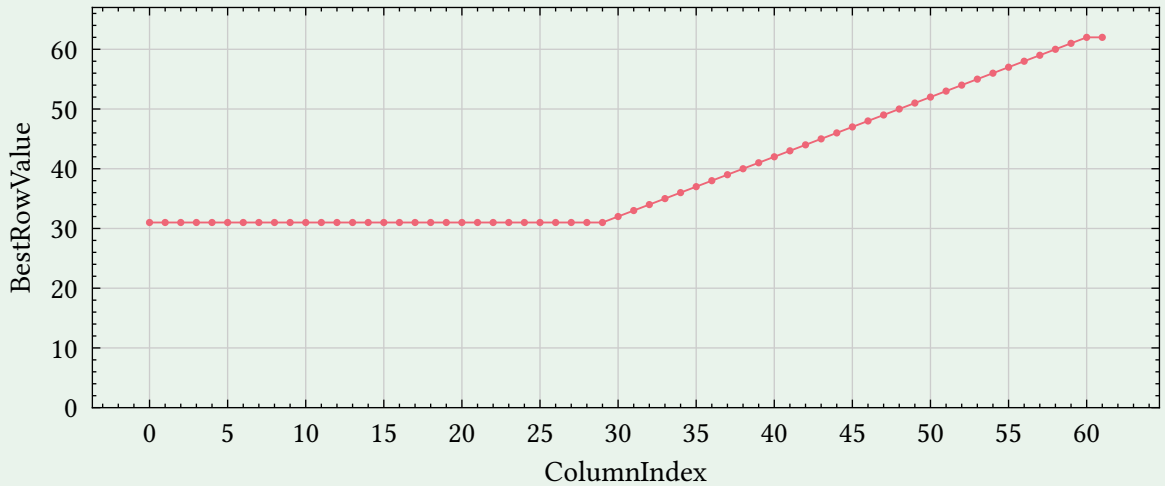


Figure 21: The progression of BestRowValue during Algorithm 2 for this instance.

Plotting bar-charts with annotations about which elements got added / removed from our "warehouse" (like in Example 2.4.1) makes for too wide a plot, so we drop the annotations (they can be inferred from the permutations, if necessary) and instead use a regular line-chart. This instance here is 1-dimensional, so only the stock of one "ingredient" needs to be tracked over time.

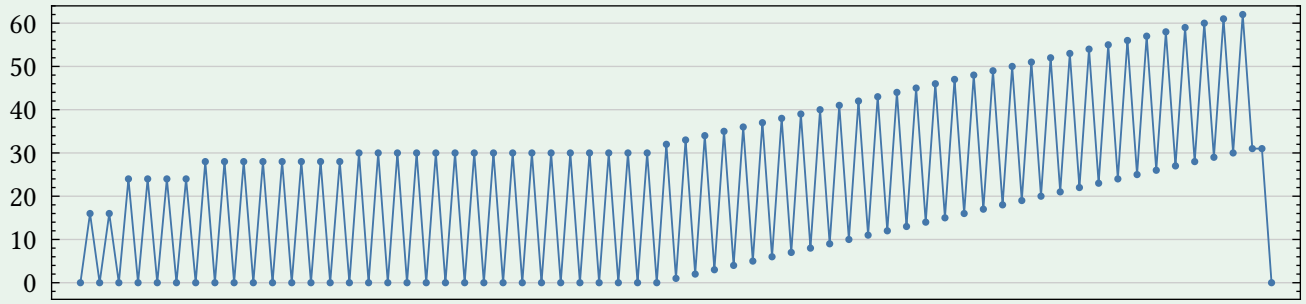


Figure 22: Visualising the "warehouse" for $\pi_{\text{IterRound}}$ over time. The maximum capacity is 62.

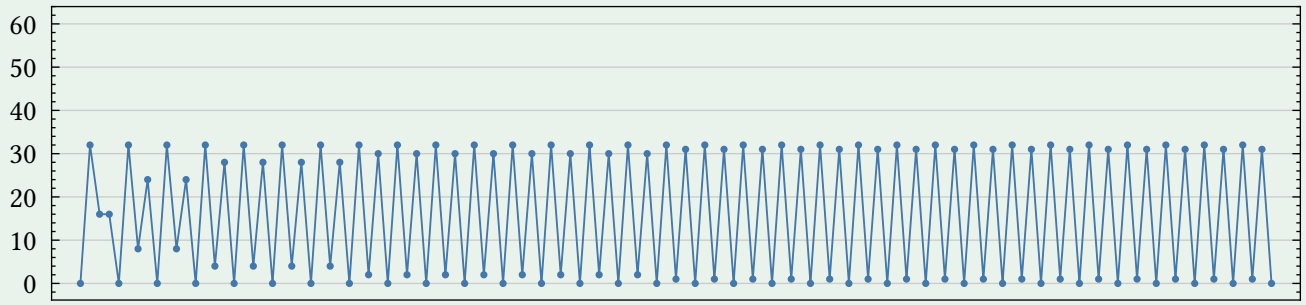


Figure 23: Visualising the "warehouse" for π_{Opt} over time. The maximum capacity is 32.

Thus, for this instance, $\frac{\text{IterRound}(I)}{\text{Opt}(I)} = \frac{62}{32} = 1.9375$. More generally, if I_k is the above instance for some k, Lorieau (2024) showed that $\frac{\text{IterRound}(I_k)}{\text{Opt}(I_k)}$ is at least $2 - 2^{1-k}$.

```python
def gasoline(n: int) → tuple[list[np.ndarray], list[np.ndarray]]:
    """Return a new gasoline-problem, specified by the two lists of 2d-non-negative-integer-points.
    Both lists must have length at most n and consist only of points in ℕ^2.
    """
    k = int(math.log2(n + 2)) - 1
    xs, ys = [], []
    for i in range(1, k):
        rounded = int(2**k * (1 - 2 ** (-i)))
        xs.extend([np.array([rounded, 0]) for _ in range(2**i)])
        ys.extend([np.array([rounded, 0]) for _ in range(2**i)])

    xs.extend([np.array([2**k, 0]) for _ in range(2**k - 1)])
    xs.append(np.array([0, 0]))

    rounded = int(2**k * (1 - 2 ** (-k)))
    ys.extend([np.array([rounded, 0]) for _ in range(2**k)])

    return xs, ys
```

(a) Initial program. This is Example 4.4.1 embedded into $\mathbb{R}^2$ via $x \mapsto (x, 0)$.

```python
 def gasoline(n: int) → tuple[list[np.ndarray], list[np.ndarray]]:
     """Yet another variation of the gasoline-problem generator."""
     k = int(math.log2(n + 2)) - 1
     xs, ys = [], []
     for i in range(1, k):
         rounded = int(2**k * (1 - 2 ** (-i)))
         xs.extend([np.array([rounded, 0]) for _ in range(2**i)])
-        ys.extend([np.array([rounded, 0]) for _ in range(2**i)])
+        ys.extend([np.array([rounded, 2]) for _ in range(2**i)])  # No change

-    xs.extend([np.array([2**k, 0]) for _ in range(2**k - 1)])
+    xs.extend([np.array([2**k, 4]) for _ in range(2**k - 2)])  # No change
-    xs.append(np.array([0, 0]))
+    xs.append(np.array([0, 1]))  # Changed from [0, 2] to [0, 1]
+    xs.append(np.array([2**k, 2]))  # Changed from [2**k, 0] to [2**k, 2]

     rounded = int(2**k * (1 - 2 ** (-k)))
-    ys.extend([np.array([rounded, 0]) for _ in range(2**k)])
+    ys.extend([np.array([rounded, 2]) for _ in range(2**k - 1)])  # No change
+    ys.append(np.array([0, 1]))  # Changed from [0, 2] to [0, 1]
```

(b) The difference between the initial program and a program found by FunSearch after 10 trials of 950 samples each. We only tuned this by discarding the last element of both lists.

Listing 13: The evolution of programs generating 2-dimensional gasoline-instances. The model used was open-mistral-nemo with a temperature of 1.5. Lists were clipped to length n before evaluation, and the final element of `ys` *set such that* `sum(xs) == sum(ys)`.

FunSearch found the following two very similar instances. Fix the dimension $d \geq 2$ and parameter k.

$$X := \left(\bigoplus_{i=1}^{k-1}\bigoplus_{1}^{2^i}\bigoplus_{j=2}^{d}[u_i e_1 + 2e_j]\right) \oplus \left(\bigoplus_{j=2}^{d}\left(\bigoplus_{1}^{2^k-1}[2^k e_1]\right) \oplus [2e_j]\right), \quad Y := \bigoplus_{i=1}^{k}\bigoplus_{1}^{2^i}\bigoplus_{j=2}^{d}[u_i e_1 + 1e_j] \qquad \text{[G-Low]}$$

$$X := \left(\bigoplus_{i=1}^{k-1}\bigoplus_{1}^{2^i}\bigoplus_{j=2}^{d}[u_i e_1 + 4e_j]\right) \oplus \left(\bigoplus_{j=2}^{d}\left(\bigoplus_{1}^{2^k-1}[2^k e_1]\right) \oplus [4e_j]\right), \quad Y := \bigoplus_{i=1}^{k}\bigoplus_{1}^{2^i}\bigoplus_{j=2}^{d}[u_i e_1 + 2e_j] \qquad \text{[G-High]}$$

The two instances only differ in three places, namely in the constant scalars preceding the e_j. Specifically, [G-High] is just [G-Low] multiplied by the diagonal matrix $\mathrm{diag}(1, 2, \ldots, 2)$. Compared to the values u_i preceding the e_1, these constant scalars are quite small.

While [G-High] seems to achieve higher scores, [G-Low] seems better suited for proving asymptotic bounds, because the outputs of Algorithm 2 have more structure there (compare e.g. Figure 24 to Figure 28, and Figure 25 to Figure 29 below). That said, we did not manage to prove any asymptotic bounds for either instance and only note scores for specific parameters, and patterns we spotted in those scores.

Example 4.4.2: Consider an optimal solution π_{Opt} and the solution $\pi_{\text{IterRound}}$ found by Algorithm 2 for [G-Low], with $d = 3$ and $k = 5$. This instance has 124 elements. We draw the same plots as in Example 4.4.1.

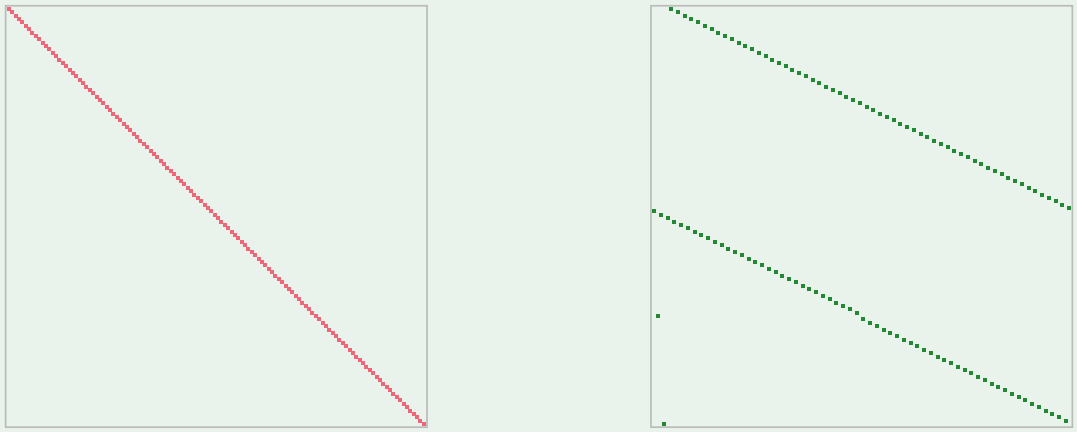


Figure 24: The permutation-matrices for $\pi_{\text{IterRound}}$ (left) and some π_{Opt} (right).

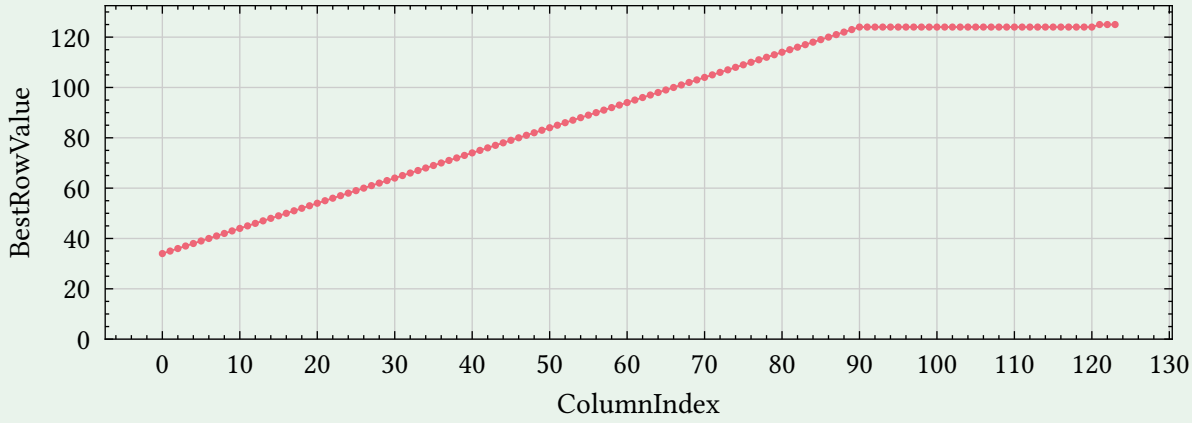


Figure 25: The progression of BestRowValue *during Algorithm 2 for this instance.*

The first component is shown in blue, the second in purple, and the third one in yellow.

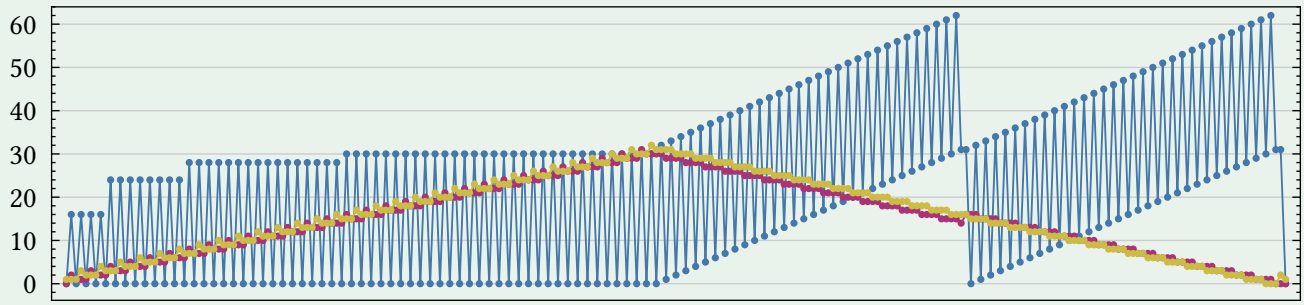


Figure 26: Visualising the "warehouse" for $\pi_{\text{IterRound}}$ over time.
The maximum capacity is $62 + 32 + 31 = 125$.

Figure 27: Visualising the "warehouse" for some π_{Opt} over time.
The maximum capacity is $32 + 2 + 2 = 36$.

Here, $\frac{\text{IterRound}(I)}{\text{Opt}(I)_{(d)}} = \frac{125}{36} \approx 3.47$. This shows $\rho^{(3)}_{\text{IterRound}} \geq 3.46$, disproving one the conjecture of Rajković (2022) that $\rho_{\text{IterRound}} = 2$ for $d > 2$.

As described in Lorieau (2024, Section 2.3.3), attempts at using local search to disprove this conjecture turn out to be ineffective. In our experiments with local search, we were also unable to find instances with $\frac{\text{IterRound}(I)}{\text{Opt}(I)} > 2$ when starting from a *random instance*. However, when starting local search from the instance in Example 4.4.1 instead ($k = 4$, embedded into $\mathbb{R}^2$ via $x \mapsto (x, 0)$) we *did* find instances with $\frac{\text{IterRound}(I)}{\text{Opt}(I)} = 2.1$. We were unable to generalise these instances to higher dimensions, nor could we spot any patterns.

Example 4.4.3: We plot solutions for [G-High] with $d = 3$ and $k = 5$ in the same way as Example 4.4.2.

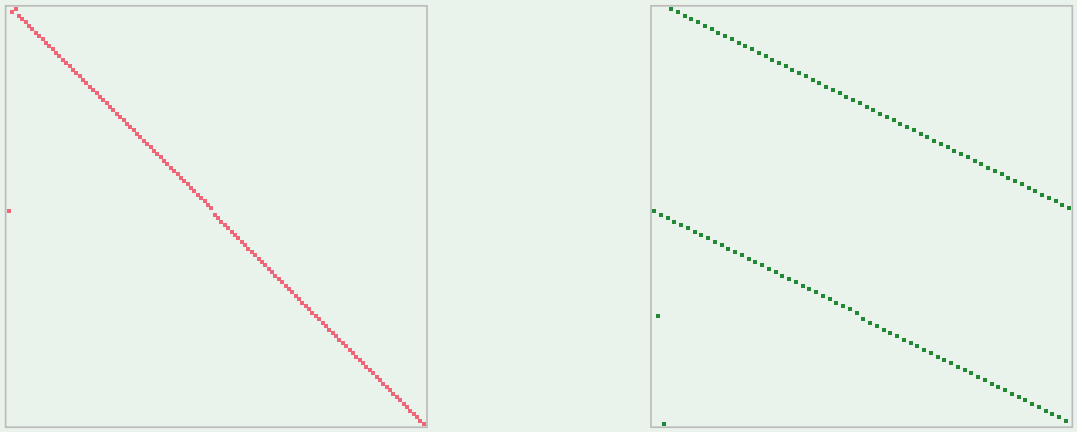


Figure 28: The permutation-matrices for $\pi_{\text{IterRound}}$ (left) and some π_{Opt} (right).

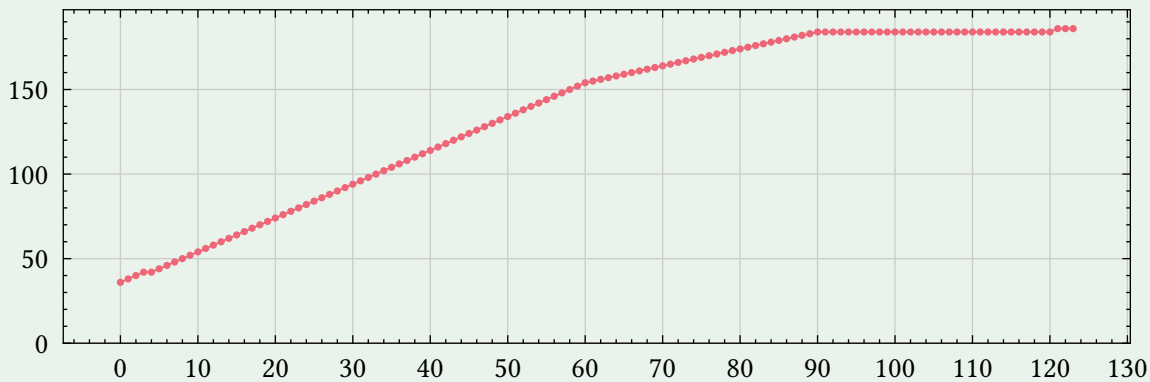


Figure 29: The progression of BestRowValue during Algorithm 2 for this instance.

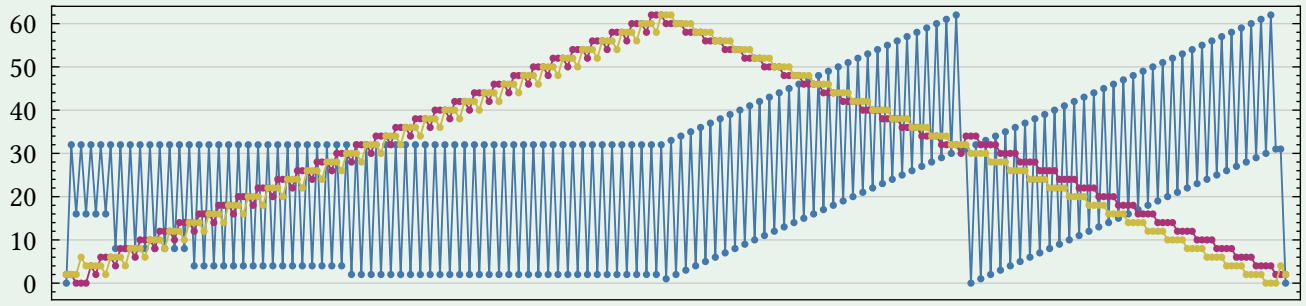


Figure 30: Visualising the "warehouse" for $\pi_{\text{IterRound}}$ over time.
The maximum capacity is $62 + 62 + 62 = 186$.

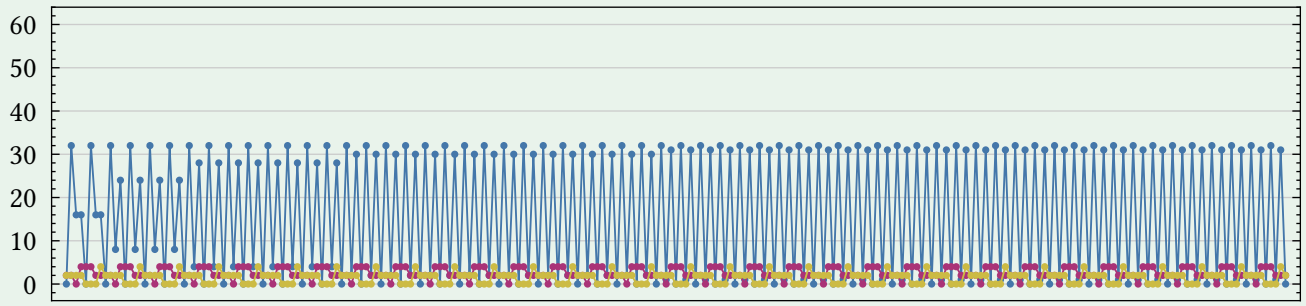


Figure 31: Visualising the "warehouse" for π_{Opt} over time.
The maximum capacity is $32 + 4 + 4 = 40$.

Here, $\text{IterRound}(I)/\ \text{Opt}(I) = 186/40 = 4.65$, which shows $\rho^{(3)}_{\text{IterRound}} \geq 4.65$.

Although [G-Low] and [G-High] are similar to the instance in Example 4.4.1, the proof used by Lorieau (2024) to show $\rho^{(1)}_{\text{IterRound}} \geq 2$ does not work for [G-Low] or [G-High]. It relied on using the same optimal solution for LP' for all iterations during the first half of Algorithm 2. This can not apply to [G-Low] or [G-High]: Compare Figure 21 to Figure 25 and Figure 29. In Lorieau (2024)'s instance, the BestRowValue is constant at first (as the same optimal solution can be used for the different LP'). However, for [G-Low] and [G-High], the BestRowValue inrceases immediately, meaning the optimum for LP' must keep changing for the first half of the algorithm. If we wanted to prove asymptotic bounds for either instance, our next step would be to prove properties of optimal LP'-solutions at each iteration of the first half of Algorithm 2. Proving optimality of these LP'-solutions would also be more difficult, as we can't use the same argument as Lorieau (2024) did, and the dual LP is unwieldy.

Lastly, we also show traces in phase-space (like in Figure 10 for Example 2.4.1) for specific 2-dimensional instances.

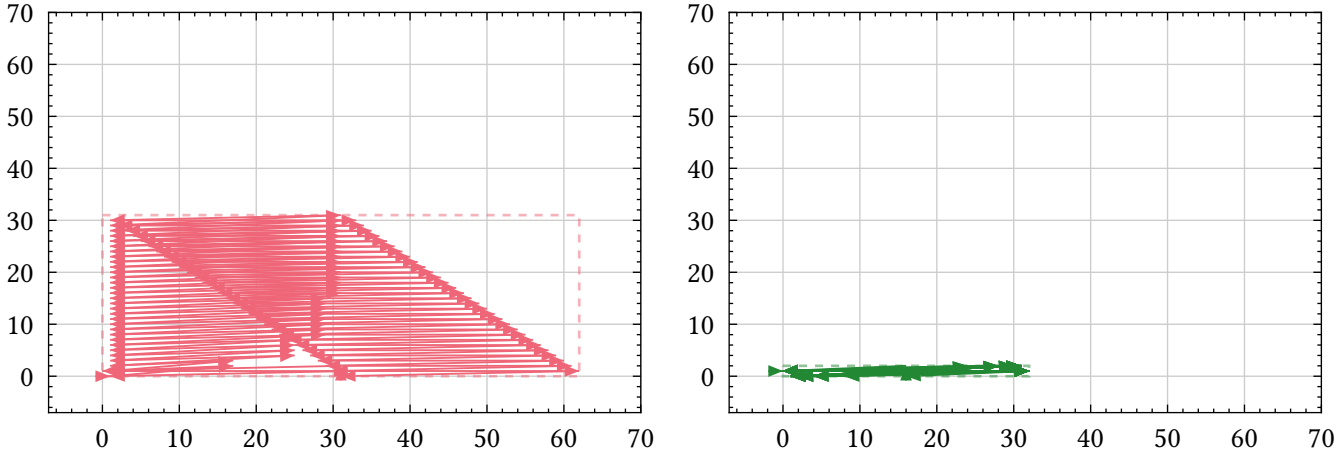


Figure 32: Tracing $\pi_{\text{IterRound}}$ (left) and π_{Opt} (right) in phase-space, for [G-Low] with $d = 2$ and $k = 5$.

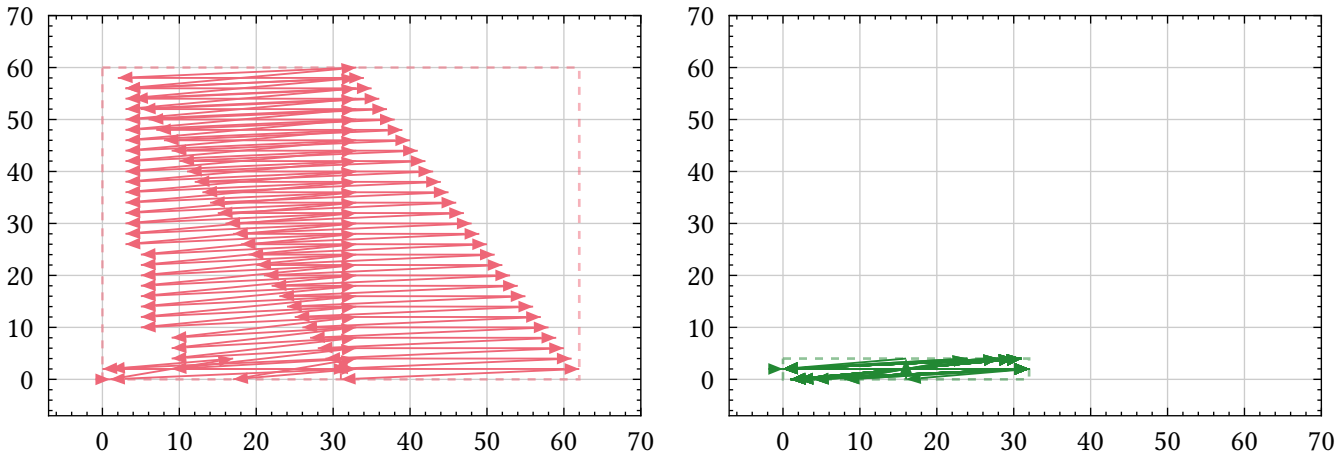


Figure 33: Tracing $\pi_{\text{IterRound}}$ (left) and π_{Opt} (right) in phase-space, for [G-High] with $d = 2$ and $k = 5$.

4.4.1 Empirical Data

As mentioned, [G-High] is the same as [G-Low] scaled by the diagonal-matrix $\text{diag}(1, 2, ..., 2)$, which raises the question: What is the behaviour for diagonal values other than 2? For some rational $p/q =: \alpha \in Q_{\geq 0}$, define I_α as [G-Low] scaled by $\text{diag}(q, p, ..., p)$ (scaling by $\text{diag}(1, \alpha, ..., \alpha)$ would lead to an equivalent instance, but X and Y were required to be integral in the problem-statement).

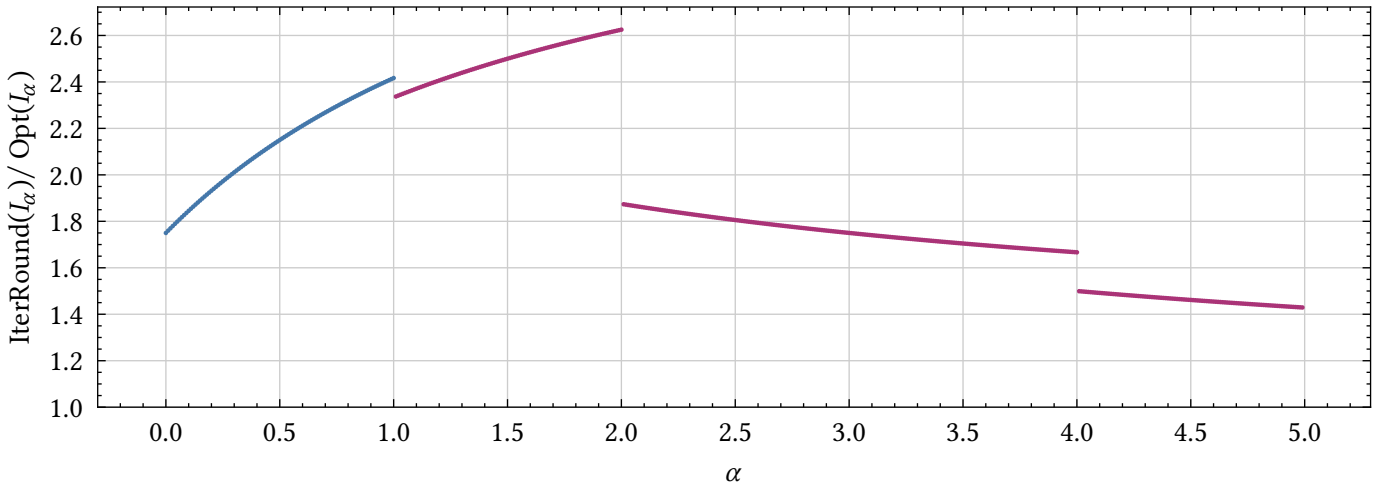
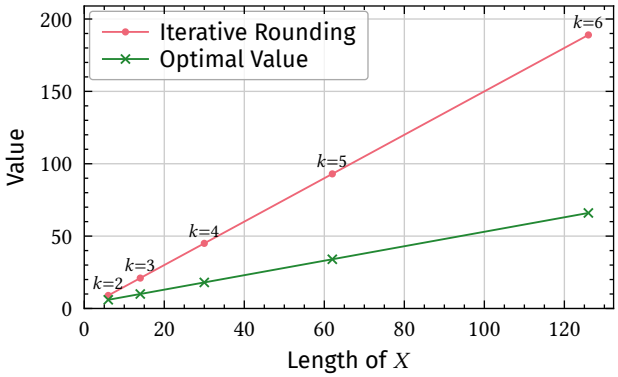


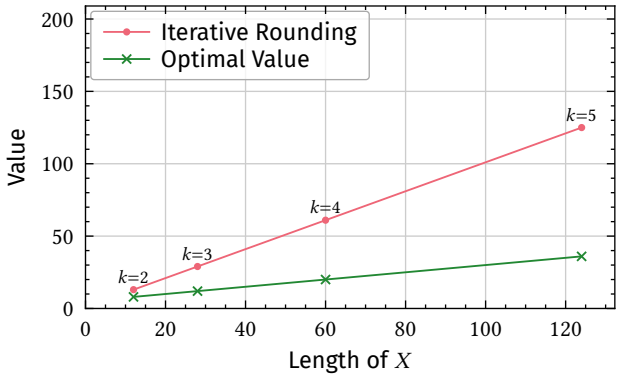
Figure 34: Scores of I_α for different choices of $\alpha \in \left\{\frac{z}{100} \;\middle|\; z \in \mathbb{Z}\right\}$, with $d = k = 3$. A point is coloured blue iff the permutation $\pi_{\text{IterRound}}$ found by Algorithm 2 is the identity (for the shown α, this happens iff $\alpha \leq 1.0$).

This is weak evidence for $I_2 = [\text{G-High}]$ being best-possible among all I_α, and $I_1 = [\text{G-Low}]$ being best-possible among those I_α where the output of Algorithm 2 has simple structure.

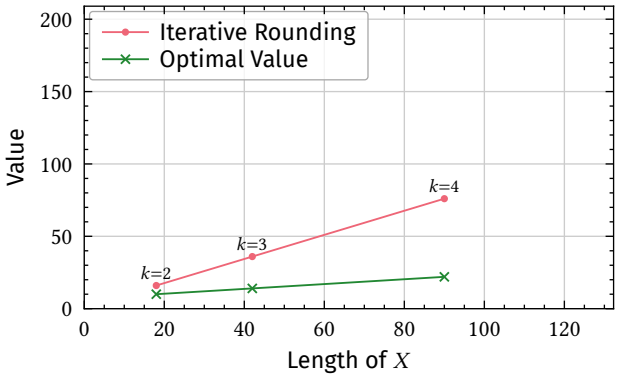
While we could not *prove* asymptotic results, plotting the values Opt and $\mathrm{IterRound}$ against the *size of the instances* showed perfectly straight lines, except for $d = 5$, where the case $k = 2$ broke linearity for $\mathrm{IterRound}$, the actual values being 20 and 24, respectively. Calculating $\mathrm{IterRound}$ and Opt for larger instances is computationally prohibitive. If these linear relationships held true asymptotically, we would obtain respective bounds on $\rho^{(d)}_{\mathrm{IterRound}}$, as noted below.



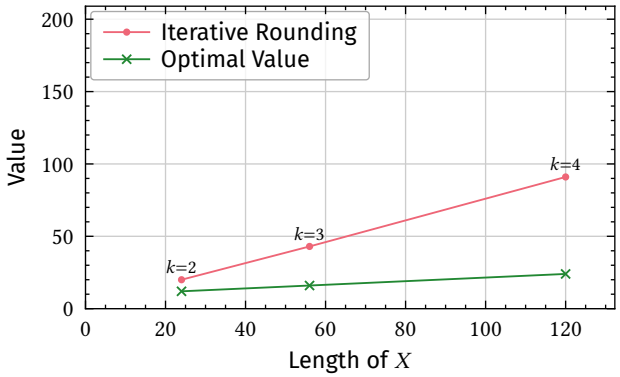
(a) For $d = 2$, $\mathrm{IterRound} \sim 3n/2$ and $\mathrm{Opt} \sim n/2 + 3$. If true asymptotically, it would imply $\rho^{(2)}_{\mathrm{IterRound}} \geq 3$.



(b) For $d = 3$, $\mathrm{IterRound} \sim n + 1$ and $\mathrm{Opt} \sim n/4 + 5$. If true asymptotically, it would imply $\rho^{(3)}_{\mathrm{IterRound}} \geq 4$.



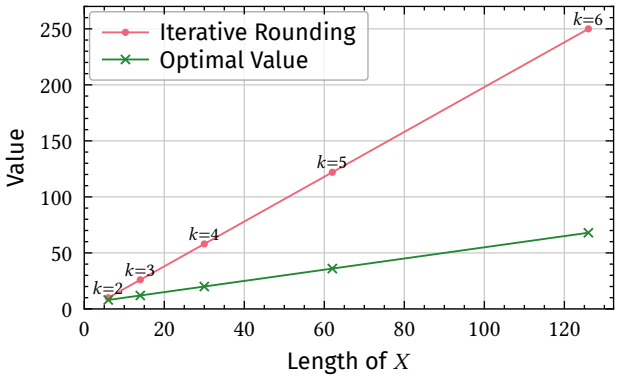
(c) For $d = 4$, $\mathrm{IterRound} \sim 5n/6 + 1$ and $\mathrm{Opt} \sim n/6 + 7$. If true asymptotically, it would imply $\rho^{(4)}_{\mathrm{IterRound}} \geq 5$.



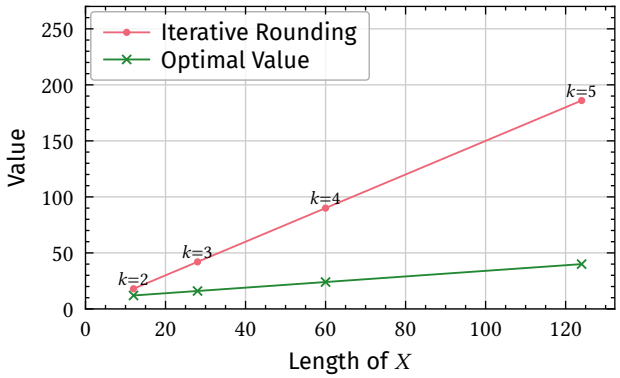
(d) For $d = 5$, ignoring $k = 2$, $\mathrm{IterRound} \sim 3n/4 + 2$ and $\mathrm{Opt} \sim n/8 + 9$. If true asymptotically, it would imply $\rho^{(5)}_{\mathrm{IterRound}} \geq 6$.

Figure 35: Optimal values and $\mathrm{IterRound}$-values on [G-Low] for different choices of d and k (starting at $k = 2$) plotted against the length $n := |X|$, along with linear extrapolations. The asymptotic bounds empirically follow a pattern of $\rho^{(d)}_{\mathrm{IterRound}} \geq d + 1$.

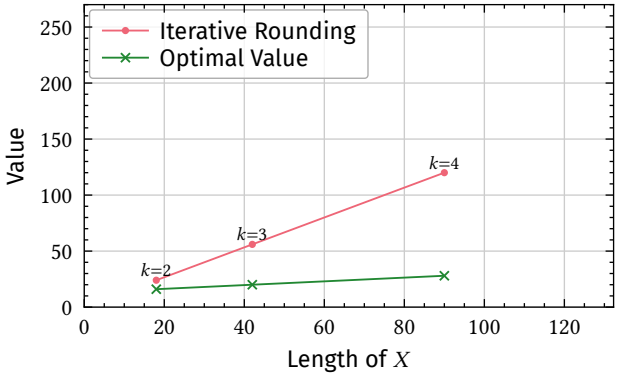
Conjecture 1: For all $d \geq 2 : \lim_{k\to\infty} \frac{\mathrm{IterRound}([\text{G-Low}](d,k))}{\mathrm{Opt}([\text{G-Low}](d,k))} \geq d + 1$.



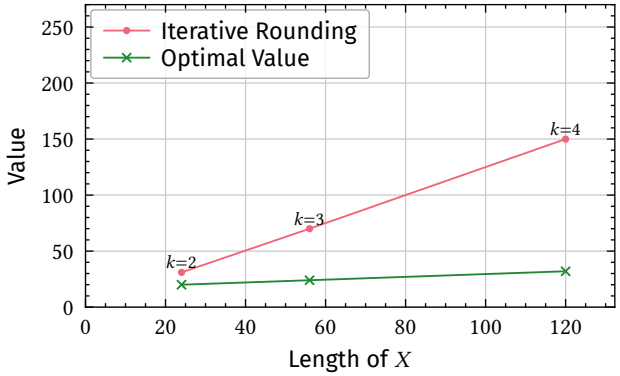
(a) For $d = 2$, IterRound $\sim 2n - 2$ and Opt $\sim n/2 + 5$. If true asymptotically, it would imply $\rho^{(2)}_{\text{IterRound}} \geq 4$.



(b) For $d = 3$, IterRound $\sim 3n/2$ and Opt $\sim n/4 + 9$. If true asymptotically, it would imply $\rho^{(3)}_{\text{IterRound}} \geq 6$.



(c) For $d = 4$, IterRound $\sim 4n/3$ and Opt $\sim n/6 + 13$. If true asymptotically, it would imply $\rho^{(4)}_{\text{IterRound}} \geq 8$.



(d) For $d = 5$, ignoring $k = 2$, IterRound $\sim 5n/4 + 1$ and Opt $\sim n/8 + 17$. If true asymptotically, it would imply $\rho^{(5)}_{\text{IterRound}} \geq 10$.

Figure 36: Optimal values and IterRound*-values on [G-High] for different choices of d and k (starting at $k = 2$) plotted against the length $n := |X|$, along with linear extrapolations. The asymptotic bounds empirically follow a pattern of $\rho^{(d)}_{\text{IterRound}} \geq 2d$.*

Conjecture 2: For all $d \geq 2$: $\lim_{k\to\infty} \frac{\text{IterRound}([\text{G-High}](d,k))}{\text{Opt}([\text{G-High}](d,k))} \geq 2d$.

4.5 Problems We Did Not Make Progress On

In the previous sections, we only presented the results for problems where we applied FunSearch successfully. A-priori, we did not know what problems lent themselves to FunSearch, so we used trial-and-error across several problems, and we briefly talk about said errors here.

- Best-Fit Bin-Packing: Instead of the *absolute* random-order-ratio

$$\text{RR}_{\text{BestFit}} = \sup_{I \in \mathcal{I}} \mathbb{E}_{\pi \in S_{|I|}} \left[\frac{\mathcal{A}(\pi(I))}{\text{Opt}(I)} \right],$$

  there is a different measure that only concerns itself with instances that require a "large" number of bins to pack all items, the **asymptotic random-order-ratio**:

$$\text{RR}^{\infty}_{\text{BestFit}} = \limsup_{M\to\infty} \left(\sup_{I \in \mathcal{I},\, \text{Opt}(I) \geq M} \mathbb{E}_{\pi \in S_{|I|}} \left[\frac{\mathcal{A}(\pi(I))}{\text{Opt}(I)} \right] \right),$$

  The instance I presented in Section 4.1 is no longer viable for this measure, as $\text{Opt}(I) = 2$. In fact, a recent work by Hebbar et al. (2024) showed that $\text{RR}^{\infty}_{\text{BestFit}} \leq 1.5 - \varepsilon$ for a small $\varepsilon > 0$.

    - We attempted to find instances, by adding a parameter `min_sum` to the `get_items`-function, and rejecting any instance whose sum of items was lower than `min_sum`. We then evaluated instances for large values of `min_sum`. This was not successful.
    - A different approach involves, instead of finding lists of items, finding *distributions* of weights, and sampling a random instance by sampling each item iid. from that distribution. This was not successful either.
- Clustering: We were successful in proving a new result for the Price of Hierarchical k-median clustering, $\mathrm{PoH}_{k\text{-median}} \geq \frac{1+\sqrt{5}}{2}$.
    - We also tried finding better lower bounds on the Price of Hierarchy for the following objectives, none of which FunSearch outperformed local search on:
        - k-means
        - k-median, but using the L_2 norm instead of the L_1 norm
        - k-median, but using the squared L_2 norm instead of the L_1 norm
    - We also tried finding lower bounds on the approximation-factor of the hierarchy found by agglomerative clustering, for the following objectives, but were unsuccessful as well:
        - k-means
        - k-median
        - k-median, but using the squared L_2 norm instead of the L_1 norm.
- Gasoline: Though we did find 2-dimensional instances where $\frac{\mathrm{IterRound}(I)}{\mathrm{Opt}(I)}$ was greater than 2, we were unable to find any 1-dimensional instances with that property.
- In the page-replacement-problem, we must make decisions which memory pages to keep in working memory. When a page being requested is currently not loaded (a *page-miss*), we must decide which of the currently-loaded pages to swap out, which is a relatively expensive operation. The objective then is to minimise the number of page-misses, by making smart choices about which pages to keep in working memory. A good heuristic for this is LRU, which discards the page that was Least Recently Used. Using a benchmark-instance of real-world data, we attempted to find better heuristics than this, but were unsuccessful.

For the sake of providing a rough estimate, this amounts to 14 attempts, 4 of which ($\approx 29\%$) led to new results. Even now, I do not feel like I have a good understanding of what problems lend themselves well to FunSearch, other than the obvious "Prefer research-questions that are more likely to have low-hanging fruit left". For example, it was better to work on the rather unexplored absolute random-order-ratio of some bin-packing heuristic, rather than working on $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$. It might be better to try FunSearch in a wide variety of contexts, so that one has many chances at finding new results, but also to get a better understanding of which problems FunSearch works well on.

5 Conclusion

Using FunSearch with manual tuning, we obtained new results for different problems:

- We proved that the absolute random-order-ratio of Best-Fit bin-packing is exactly 1.5. Future research may apply similar techniques to improve the bounds on its asymptotic random-order-ratio, which we only know to be between 1.144 and $1.5 - \varepsilon$ (Hebbar et al. 2024).
- Using a sequence of instances of the knapsack-problem, we showed that the size of Pareto-sets of sub-instances can be exponentially larger than the size of the final Pareto-set, with the base of this exponential growth being between 1.509 and 2.0. It would be interesting to identify the largest possible base of this exponential growth.
- An instance found using FunSearch gave a lower bound on the Price of Hierarchy of k-median clustering of $\frac{1+\sqrt{5}}{2} \approx 1.618$, but the best-known upper-bound is 16 (Dai 2014).
- FunSearch yielded a sequence of instances where the iterative-rounding-algorith has an approximation-factor larger than 2, and we conjecture that the approximation-factor on this sequence is linear in the dimension.

The method of FunSearch itself offers many avenues for future research: Applying it to novel circumstances, extending it (see e.g. Novikov et al. (2025)), running large-scale ablations on parameters like LLM-choice, temperature, prompt, and sample-size across different problems, or using LLMs and proof-assistants like Lean (Moura and Ullrich 2021) for automated proofs during the search. FunSearch also seems more promising in areas that local search *can not* be applied to, for instance finding novel heuristics for existing problems, in the form of python-programs.

As LLMs grow more competent and inference becomes faster and less expensive over time, FunSearch becomes an increasingly useful tool for mathematical research. At the same time, the moral status of LLMs is subject of an ongoing debate (Long et al. 2024), and if their wellbeing turns out to be worth contemplating, we would have to consider the effects of FunSearch's queries on their welfare.

Bibliography

Albers, Susanne, Arindam Khan, and Leon Ladewig. 2021. “Best Fit Bin Packing with Random Order Revisited.” *Algorithmica* 83 : 1–26. https://doi.org/10.1007/s00453-021-00844-5.

Arutyunova, Anna, and Heiko Röglin. 2025. “The Price of Hierarchical Clustering.” *Algorithmica*, 1–33. https://doi.org/10.1007/s00453-025-01327-7.

Boyar, Joan, György Dósa, and Leah Epstein. 2012. “On the Absolute Approximation Ratio for First Fit and Related Results.” *Discrete Applied Mathematics* 160 (13): 1914–23. https://doi.org/https://doi.org/10.1016/j.dam.2012.04.012.

Cranmer, Miles. 2023. “Interpretable Machine Learning for Science with Pysr and Symbolicregression.jl.”. arxiv.org/abs/2305.01582.

Dai, WenQiang. 2014. “A 16-Competitive Algorithm for Hierarchical Median Problem.” *Science China Information Sciences* 57 (3): 1–7. https://doi.org/10.1007/s11432-014-5065-0.

Dósa, György, and Jiri Sgall. 2013. “First Fit bin packing: A tight analysis.” In *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, edited by Natacha Portier and Thomas Wilke, vol. 20 of *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*. Leibniz International Proceedings in Informatics (Lipics). Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.STACS.2013.538.

Dósa, György, and Jiří Sgall. 2014. “Optimal Analysis of Best Fit Bin Packing.” In *Automata, Languages, And Programming*, edited by Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, *Automata, Languages, And Programming*. Springer Berlin Heidelberg.

Fontan, Florian, and Luc Libralesso. 2020. “Packingsolver: A Solver for Packing Problems.” *Corr*,. arxiv.org/abs/2004.02603.

Garey, Michael R., and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Großwendt, Anna-Klara. 2020. “Theoretical Analysis of Hierarchical Clustering and the Shadow Vertex Algorithm.” Doctoral dissertation. hdl.handle.net/20.500.11811/8348.

Gurobi Optimization, LLC. 2024. “Gurobi Optimizer Reference Manual.”. www.gurobi.com/.

Hebbar, Anish, Arindam Khan, and K. Sreenivas. 2024. *Bin Packing under Random-Order: Breaking the Barrier of 3/2*. https://doi.org/10.1137/1.9781611977912.145.

Johnson, Stephen C. 1967. “Hierarchical Clustering Schemes.” *Psychometrika* 32 (3): 241–54. https://doi.org/10.1007/BF02289588.

Kenyon, Claire. 1996. “Best-Fit Bin-Packing with Random Order.” In “Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms.” Special issue, *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (Atlanta, Georgia, USA), SODA '96, 359–64.

Koza, John R. 1994. “Genetic Programming as a Means for Programming Computers by Natural Selection.” *Statistics and Computing* 4 (2): 87–112. https://doi.org/10.1007/BF00175355.

Laarhoven, P. J. M., and E. H. L. Aarts. 1987. *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers.

Long, Robert, Jeff Sebo, Patrick Butlin, et al. 2024. *Taking AI Welfare Seriously*.,. https://doi.org/10.48550/arXiv.2411.00986.

Lorieau, Lucas. 2024. “Approximation Algorithm for the Generalised Gasoline Problem.” Master's thesis.

Lovász, László Miklós. 1979. “Combinatorial Problems and Exercises.”. api.semanticscholar.org/CorpusID:117668134.

Moitra, Ankur, and Ryan O'Donnell. 2011. “Pareto Optimal Solutions for Smoothed Analysts.” In “Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing.” Special issue, *Proceedings of the Forty-Third Annual ACM Symposium on Theory of Computing* (San Jose, California, USA), STOC '11, 225–34. https://doi.org/10.1145/1993636.1993667.

Moura, Leonardo de, and Sebastian Ullrich. 2021. “The Lean 4 Theorem Prover and Programming Language.” In “Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings.” Special issue, *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings* (Berlin, Heidelberg), 625–35. https://doi.org/10.1007/978-3-030-79876-5_37.

Nemhauser, George L., and Zev Ullmann. 1969. “Discrete Dynamic Programming and Capital Allocation.” *Management Science* 15 (9): 494–505.

Newman, Alantha, Heiko Röglin, and Johanna Seif. 2018. “The Alternating Stock Size Problem and the Gasoline Puzzle.” *ACM Trans. Algorithms* (New York, NY, USA) 14 (2). https://doi.org/10.1145/3178539.

Nikoleit, Henri, Ankit Anand, Anurag Murty Naredla, and Heiko Röglin. 2025. *Adversarial Examples for Heuristics in Combinatorial Optimization: An LLM Based Approach*.

Novikov, Alexander, Ngân Vũ, Marvin Eisenberger, et al. 2025. “Alphaevolve: A Coding Agent for Scientific and Algorithmic Discovery.” *Arxiv Preprint Arxiv:2506.13131*,.

Parczyk, Olaf, Sebastian Pokutta, Christoph Spiegel, and Tibor Szabó. 2023. “Fully Computer-Assisted Proofs in Extremal Combinatorics.” *Proceedings of the AAAI Conference on Artificial Intelligence* 37 (10): 12482–90. https://doi.org/10.1609/aaai.v37i10.26470.

Petersen, Brenden. 2019. “Deep Symbolic Regression: Recovering Mathematical Expressions from Data via Policy Gradients.”. https://doi.org/10.48550/arXiv.1912.04871.

Rajković, Ivana. 2022. “Approximation Algorithms for the Stock Size Problem and the Gasoline Problem.” Master's thesis.

Rieck, Bastian. 2021. “Basic Analysis of Bin-Packing Heuristics.”. arxiv.org/abs/2104.12235.

Romera-Paredes, Bernardino, Mohammadamin Barekatain, Alexander Novikov, et al. 2024. “Mathematical Discoveries from Program Search with Large Language Models.” *Nature* 625 (7995): 468–75.

Röglin, Heiko. 2020. “Smoothed Analysis of Pareto Curves in Multiobjective Optimization.” In *Beyond the Worst-Case Analysis of Algorithms*, edited by Tim Roughgarden, *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press. https://doi.org/10.1017/9781108637435.020.

Schubert, Erich. 2023. “Stop Using the Elbow Criterion for K-Means and How to Choose the Number of Clusters Instead.” *SIGKDD Explor. Newsl.* (New York, NY, USA) 25 (1): 36–42. https://doi.org/10.1145/3606274.3606278.

Stinson, D. R., and S. A. Vanstone. 1985. “A Few More Balanced Room Squares.” *Journal of the Australian Mathematical Society. Series A. Pure Mathematics and Statistics* 39 (3): 344–52. https://doi.org/10.1017/S1446788700026112.