On solving the Partition problem

Lasse Lüder[1]

A research project in the Computer Science Masters program

Institute for algorithms and complexity
University of Technology Hamburg

[1]lasse.lueder@tuhh.de

The partition problem is a core NP-complete problem. A list of positive integers is to be partitioned into to sublists, such that both sublists' sums differ as little as possible. A variety of approximation algorithms as well as complete and complete anytime algorithms exist. We show that the partition problem is fixed-parameter tractable with respect to the number of different values in the input. We examine an attempt of approximately solving the partition via bin packing. In its current form, the resulting algorithm does not surpass the best known approximation algorithms.

1 Introduction

The PARTITION problem can be stated as follows: Divide a list $(a_1, a_2, \ldots, a_n)$ of positive integers into two sublists by choosing a subset of indices $A \subset [n]$ to go in one sublist, such that the discrepancy

$$D(A) = \left| \sum_{i \in A} a_i - \sum_{i \notin A} a_i \right|$$

is minimised [1]. A solution to this problem consists of the two sublists and is called a partition. Trivially, we always have $D(A) \geq 0$. If the sum of all numbers is odd, then we have $D(A) \geq 1$. A partition with a discrepancy of 0 or 1 is called a perfect partition. PARTITION can also be formulated as a decision problem: for $a_1, a_2, \ldots, a_n$, does a partition with a discrepancy below a given $k \in \mathbb{N}$ exist?

The PARTITION problem is NP-complete. It is referred to as one of the six "basic core" NP-complete problems, which are often used in reductions to other problems for proofs of NP-completeness. Out of these six, it is the only one that involves numbers [6, 12]. Closely related problems are BIN PACKING, SUBSET SUM and KNAPSACK. PARTITION has been called "the easiest hard problem" [16], because many instances can be solved easily, but other instances can be very hard to solve [9].

PARTITION has gained interest not only from operations research (e.g. [18]), but also from physicists and mathematicians. It has applications for example in multiprocessor scheduling or VLSI. Physicists such as [15, 16] have examined analogies between combinatorial problems and physical phenomenons. It has also been used as an interesting problem for quantum computing [4].

The remainder of this document is structured as follows: Section 2 gives a deeper insight into the computational difficulty of PARTITION. Then, we describe and compare several known algorithms to solve PARTITION, both approximation and exact algorithms. We also present a new kernelization method that allows to solve PARTITION in time $\mathcal{O}(n^k)$, when only k different values are in the instance (section 3). An algorithm to solve PARTITION with BIN PACKING solvers is presented and analysed in section 4, together with a new BIN PACKING approximation algorithm by [10] and considerations about benefits for solving PARTITION. Section 5 gives relevant implementation details. A summary in section 6 concludes this work.

Notation The input and output of PARTITION is often defined as (multi)sets. When discussing PARTITION or BIN PACKING, the inputs and outputs are only used as a collection of numbers, with multiple ocurrences allowed. For simplicity and be consistent with implementations, we will use the term "list" and "sublist" in this document, as they match the intent best.

In BIN PACKING, each bin has a capacity and a size. The bin's size is the sum of all items or item sizes in the bin. A bin's size cannot exceed its capacity.

Nomenclature

$[x]$	$\{1, \ldots, x\}$, $x \in \mathbb{N}$
a_i	the i-th number in a PARTITION input
m	resolution of numbers, measured in bit
n	number of elements in a PARTITION or BIN PACKING instance
v_j	a unique value that occurs in a PARTITION instance
w	number of different unique values that occur in a PARTITION instance

2 The computational difficulty of Partition

It section examines the computational difficulty of PARTITION and describes a phase transition in difficulty with insights in the easy and the hard phase.

While PARTITION is NP-complete, not all instances are practically hard to solve. This section focuses on the properties of the problem itself, but since some properties manifest in the behaviour of algorithms, some knowledge on algorithms is given where needed. All details on algorithms can be found in section 3.

Two remarks should be made first. Note that the size of numeric problems is not only determined by the number of numbers, n in this case, but also the size of each number. If each number in the problem is smaller than M, then there are $\mathcal{O}(nM)$ possible discrepancies, as the sum of all numbers is nM. The size of each number is only $\mathcal{O}(\log M)$, so that the number of possible discrepancies does not indicate a possibility for an algorithm with polynomial runtime in the problem size.

The size of each number will be denoted by m in the rest of this work. In the literature, both problems with m-bit integers or with real numbers in $[0, 1]$, represented with m bit precision, are considered. Also, solutions can be restricted to sublists of equal size or a fixed size difference, yielding the CONSTRAINED NUMBER PARTITIONING problem, see for example [3]. In this work, the two-way UNCONSTRAINED NUMBER PARTITIONING problem with integers will be used.

To get an insight into the properties of PARTITION, let's examine the runtime of the Complete Karmarkar-Karp algorithm (CKK). CKK has a worst-case complexity of $\mathcal{O}(2^n)$, as it searches the entire search space, if necessary. It uses a heuristic to go through the possibilities in a smart way and also applies pruning, so that the average-case complexity is lower. Consider the average runtime of CKK in figure 1, when run on random problems of varying size. The CKK's runtime initially appears to increase exponentially, but then reaches a maximum at $n = 14$, then decreases and until $n = 25$ and then only slowly increases with n. Apparently, something changes drastically at around $n = 14$. The figure also shows how many of the problems have a perfect solution. This ratio transitions from 0 to 1 as CKK reaches its highest run times.

PARTITION shows a typical behaviour of NP-complete problems: a phase transition. A phase transition is a sudden qualitative change in the problem's behaviour. Phase transitions have been observed for example in the random graph model of Erdös and Rényi or other problems such as the random k-SAT problem [2]. Since phase transitions are well known in physics (think of the gaseous, liquid and solid state of water), many of the following insights gained about this phase transition in PARTITION were achieved with methods of statistical physics.

The involved numbers in the experiment in figure 1 have $m = 10\,\text{bit}$. The ratio $\kappa = m/n$ was introduced by Gent and Walsh [7] as the key property for the phase transition. As they have shown by simulation, problems with $\kappa < 0.96$ have no perfect solution and problems with $\kappa > 0.96$ with high probability (i.e., for $n \to \infty$). Later, the threshold value for infinite problem sizes has been more formally determined to be 1 by Mertens [15]. Gent and Walsh had run experiments with finite problem sizes, which lead to their deviating estimate. Borgs et al. [2] have also derived a formula for the threshold in finite-sized problems:

$$\kappa_c = 1 - \frac{\log_2(n)}{2n} + \frac{\lambda}{n}, \tag{1}$$

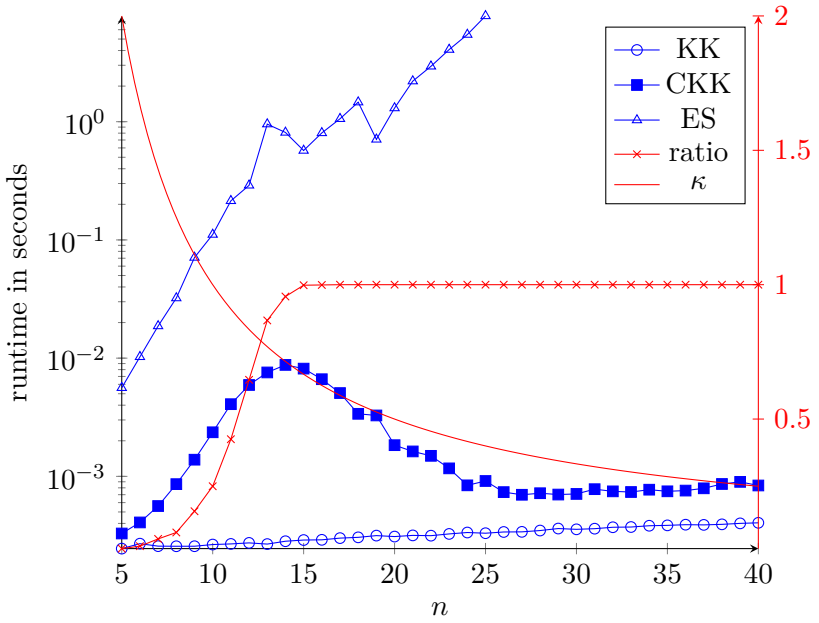


Figure 1: Blue: Average run times of several algorithms for PARTITION. The Karmarkar-Karp-algorithm (KK) [11] is a polynomial-time approximation heuristic. The complete Karmarkar-Karp-algorithm (CKK) [13] is a complete anytime algorithm based on the KK-heuristic. Exhaustive search (ES) searches until it finds a perfect partition and is only shown for comparison. Red: Number ratio of problem instances with a perfect solution and κ.
Each data point for ratio, KK and CKK is the average of 1000 random instances, each data point for ES is the average of 10 random instances with numbers sampled from UNIFORM$(0, 1023)$.

where λ can be used to gain insight into the transition itself. In particular, with

$$r(\lambda) = \exp\left(-\sqrt{\frac{3}{2\pi}}2^{-\lambda}\right)$$

it is possible to calculate the probability that a perfect partition exists.

Theorem 1. [2] *Let $m = \kappa_c n$ with κ_c as in (1), and assume that $\lim_{n\to\infty} \lambda_n = \lambda$ exists. Then*

$$\lim_{n\to\infty} \mathbb{P}\left(\exists a \textit{ perfect partition}\right) = \begin{cases} 1 & \textit{if } \lambda_n = -\infty \\ 1 - \frac{1}{2}r(\lambda)(r(\lambda)+1) & \textit{if } \lambda_n \in (-\infty, \infty) \\ 0 & \textit{if } \lambda = \infty. \end{cases}$$

If we look at $n = 12$ as an example and calculate κ_c to equal $\kappa = 10/12$, we have $\lambda_n \approx -0.21$ and the probability for a perfect partition is 0.67. The experimental value at $n = 12$ is 0.65.

Figure 2 was generated from Borg's formulas and shows how the probability of a perfect partition decreases with increasing κ. Also in this regard a phase transition around $\kappa = 1$ is visible.

Since the two phases behave so differently, they offer different aspects to study. The following insights in the so called easy phase ($\kappa < \kappa_c$) and hard phase ($\kappa > \kappa_c$) are based on [16]. In

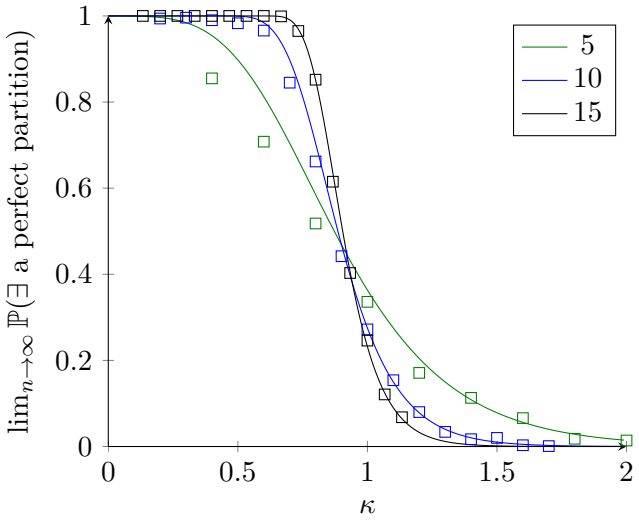


Figure 2: Probability that a PARTITION instance has a perfect partition. The solid lines are based on eq. (1) and (4) from [2], the marker show experimental values from 1000 randomly generated instances.

the easy phase, more and more perfect partition can be expected, as κ_c shrinks. Consequently, the average number of possible solutions that must be searched until the CKK finds a perfect partition also does not increase with larger n after the transition. Figure 1 shows this behaviour in the runtime. Please refer to figure 5 in [16] for a more elaborate examination.

Another interesting observation can be made about heuristic algorithms in the easy phase. The Karmarkar-Karp (KK) heuristic yields a discrepancy in $\mathcal{O}(n^{0.72 \log n})$ [18], and the greedy heuristic yields a discrepancy in $\mathcal{O}(1/n)$. Therefore, the discrepancy bound reaches 0 below a κ-threshold

$$\kappa_{kk} = \mathcal{O}\left(\alpha \frac{\log^2 n}{n \log 2}\right),$$

where $\alpha = 0.72$, for the KK heuristic and

$$\kappa_g = \mathcal{O}\left(\frac{\log_2 n^{-1}}{n}\right)$$

for the greedy heuristic.

In the hard phase, current algorithms need to explore much of the possible search space. Mertens [16] argues that significant improvements over a sequential search should not be expected in the hard phase. If there is a partition with discrepancy in the order of $\mathcal{O}(\sqrt{n}2^{-n})$, moving any number will increase the discrepancy to $\mathcal{O}(1/n)$ and another partition with low discrepancy will not share many assignments with the original one. If good partitions differ so much, they can be seen as independent from each other and there cannot be a way to find them efficiently. Partitions with large discrepancies will correlate more, as a certain small set of numbers can introduce a very large discrepancy that the other numbers must overcome to yield a good overall partitioning. These correlations can be used by heuristics, as it is done by the greedy heuristic and more effectively by the Karmarkar-Karp heuristic. Hence, an algorithm like CKK can rule out bad partitions quickly, but will generally not find a perfect partition quickly.

3 Solvers for Partition

This section describes several relevant algorithms which can be used to solve PARTITION. For each algorithm, we give a short description as well as as python code snippet and a analysis of runtime and results.

Some of the following algorithms are complete algorithms. They guarantee to solve a PARTITION instance optimally. For NP-hard problems, this guarantee comes at the cost of high computational effort. The main evaluation criterion for complete algorithms is their runtime. Other algorithms are approximations, which trade optimality for speed. The presented approximation algorithms run in polynomial time, but give weaker guarantees. The main evaluation criterion for approximation algorithms is their approximation quality, measured as the gap between the optimal solution and the approximate solution. In the best case, they can find the optimal solution. The worst case is relevant for theoretical analysis. For practical uses the average case performance is most important.

3.1 ILP

PARTITION can be formulated as an integer linear programming problem (ILP). Each number a_i is put in one or the other sublist, according to the value of the corresponding binary variable x_i. The ILP to partition n numbers is of the form

$$\min |D(A)| \qquad \text{s.t. } x_i \in \{0,1\} \quad \forall i \in [n],$$

with

$$D(A) = \sum_{i \in [n]} a_i \cdot x_i - \sum_{i \in [n]} a_i \cdot (1 - x_i).$$

The objective is not linear due to the absolute value. To get a true linear program, the absolute value can be incorporated with a helper variable $D'(A)$, two new constraints

$$\begin{aligned} D'(A) &>= D(A) \\ D'(A) &>= -D(A) \end{aligned}$$

and a new, linear objective $\min D'(A)$. Various ILP solvers are available. Our implementation in listing 1 utilizes gurobi [8].

```
x = m.addVars(range(problem.getN()), name="x", vtype=GRB.BINARY)
m.update()

sumA = 0
sumB = 0
for i in range(problem.getN()):
    sumA += problem.getNumbers()[i] * x[i]
    sumB += problem.getNumbers()[i] * (1-x[i])
diff = sumA - sumB
discrepancy = m.addVar(name="discrepancy")
m.addConstr(discrepancy >= diff)
m.addConstr(discrepancy >= -diff)

m.setObjective(discrepancy, GRB.MINIMIZE)
```

Listing 1: ILP formulation of PARTITION, written in python using gurobi. Taken from `impl/partitioning_algo/ilp.py`

3.2 Greedy algorithm

Probably the simplest approximation algorithm for PARTITION is a simple greedy heuristic. The numbers are sorted. Starting with the largest, each number is assigned to the subset with the smaller sum at that time [13].

```
numbers = sorted(instance.getNumbers(), reverse=True)

list_a = []
list_b = []
sum_a = 0
sum_b = 0

for idx, number in enumerate(numbers):
    if sum_a <= sum_b:
        list_a.append(idx if return_indices else number)
        sum_a += number
    else:
        list_b.append(idx if return_indices else number)
        sum_b += number
```

Listing 2: Greedy algorithm for PARTITION Taken from `impl/partitioning_algo/greedy.py`

The runtime complexity of this algorithm is dominated the sorting step, so that a complexity $\mathcal{O}(n \log n)$ can be achieved. This simple algorithm produces a discrepancy in $\mathcal{O}(1/n)$ for n uniformly distributed numbers.

As a side note, this greedy principle is so simple that a variant is applied by children on a daily basis, as described by [9]. When a group of players (where each player has a numeric ability

estimate) must be split in to subgroups for a sports game, both teams should have a similar total ability. In this application, the teams must also have the same size, so that the underlying problem is BALANCED NUMBER PARTITIONING. The teams choose in turn; each greedily chooses the best unassigned player. When the players are all in a similar value range, this algorithm will yield the same result as listing 2.

3.3 Karmarkar-Karp algorithm

The Karmarkar-Karp algorithm (KK) is a more advanced approximation algorithm for PARTITION. It was presented by Karmarkar and Karp in 1983 [11].

Starting with a list of n numbers, the algorithm replaces the two largest numbers by their difference, until only one number is left. This number is the final difference between the two subsets. Listing 3 shows an implementation in python. Since the algorithm shall not only give the discrepancy, but also a corresponding partition to achieve this discrepancy, it must keep track of the differencing operations it has conducted. The implementation follows [13] and stores this information in a graph. Initially, the graph consists of one node for each number and no edges. When the difference of two numbers is taken, the new value is stored in one of the nodes and an edge is added between the two nodes, indicating that both nodes must go into different sublists. Since n nodes and $n-1$ edges have been added, the resulting graph is a tree and can be coloured with two colours. In the end, the partition is constructed using the colour and the initial number of each node.

```
23  g = nx.Graph()
24  for idx, number in enumerate(instance.getNumbers()):
25      node = KarmarkarKarp.Node(idx, number)
26      g.add_node(node)
27
28  active_nodes_sorted = list(g.nodes)
29  heapq.heapify(active_nodes_sorted)
30
31  while len(active_nodes_sorted) > 1:
32      larger = heapq.heappop(active_nodes_sorted)
33      smaller = heapq.heappop(active_nodes_sorted)
34      larger.value = larger.value - smaller.value
35      heapq.heappush(active_nodes_sorted, larger)
36      g.add_edge(larger, smaller)
37
38  self.color_tree(g)
39  list_a = [n.number if not return_indices else n.idx for n in g.nodes if
    ↪  n.color == 0]
40  list_b = [n.number if not return_indices else n.idx for n in g.nodes if
    ↪  n.color == 1]
```

Listing 3: Implementation of the Karmarkar-Karp algorithm, written in python. Taken from `impl/partitioning_algo/kk.py`

Extracting the two largest nodes and inserting a new node requires $\mathcal{O}(\log n)$ time with a heap,

building the heap requires $\mathcal{O}(n \log n)$ time and colouring a tree requires $\mathcal{O}(n)$ time, so that the algorithm runtime is in $\mathcal{O}(n \log n)$. The Karmarkar-Karp algorithm yields a result with an expected value of $\mathcal{O}(1/n^{\alpha \log n})$ with $\alpha = 0.72$ [18], given an input of uniformly and identically distributed numbers. Figure 3 shows the runtime and result of the greedy and Karmarkar-Karp algorithm. KK clearly outperforms the greedy algorithm and at around $n = 60$ meets the performance of a complete algorithm.

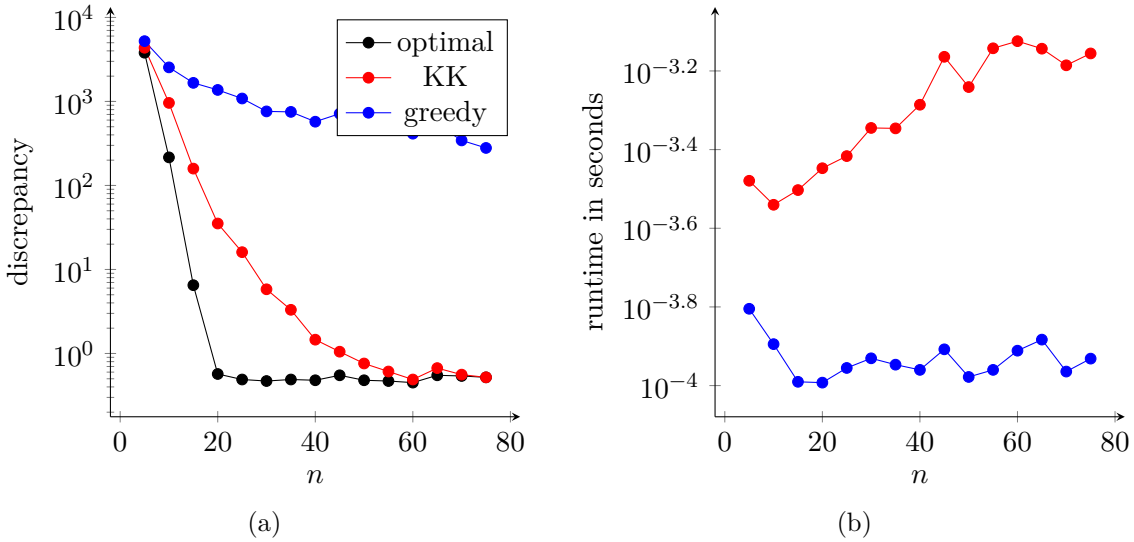


Figure 3: Discrepancy and runtime comparison of the greedy and Karmarkar-Karp algorithm. The discrepancy of optimal partitions is given for reference. Each point is the average of 100 runs with integers in the range from 1 to 2^{15}.

3.4 Complete greedy and KK algorithm

While the two previous algorithms are fast, they do not guarantee an optimal result. If $\mathsf{P} = \mathsf{NP}$, then we cannot find a polynomial-time complete algorithm, i.e. one that always finds an optimal partition. If an optimal solution is required, PARTITION can be solved optimally by searching the entire solution space in an exhaustive search. There are 2^n possible partitions, hence such an algorithm must have an exponential runtime complexity in $\mathcal{O}(2^n)$. For practical uses, the coefficients, which are usually hidden in big O notation, have large relevance.

In such a search algorithm, the search space can be represented as a binary tree. Each level represents one of the input numbers and at each node the number is assigned to both sublists, creating two subtrees. A depth-first search on this tree can check the leaves, which represent possible partitions, to find the best partition.

Korf [13] describes complete versions of the greedy algorithm and the Karmarkar-Karp algorithm: the Complete Greedy Algorithm (CGA) and Complete Karmarkar-Karp Algorithm (CKK). These algorithms still perform an exhaustive search as described above, but apply two techniques in order to search smart:

1. They use the heuristic to search the space around the heuristic solution first.
2. They use pruning, i.e. they omit parts of the search tree if these parts can provably not improve the solution.

Order Say the two sublists are called A and B. In the simple search, the left edge always assigns a number to A and the right edge to B. The leftmost and rightmost leaf would represent those partitions with all numbers in one sublist. If each number is assigned according to the heuristic result on the left edge and opposite to it on the right edge, the leftmost leaf represents the heuristic result. In the easy phase (see section 2), this can help find the optimal solution faster. This effect looses strength in the hard phase, as the optimal and very good partitions are almost independent.

Pruning

1. After the algorithm has found a perfect partition, it can stop.
2. If at any node the current discrepancy is larger than the sum of all unassigned numbers, the best choice for all of those is to assign them to the smaller sublist. All other choices can be ignored.
3. Additionally to the previous pruning rules from [13], if in the above case the best possible partition is worse than the best currently known, it can also be ignored. Even though this is almost the same as the above, the empirical data in figure 4 shows that there is a slight advantage in terms of time and searched nodes.

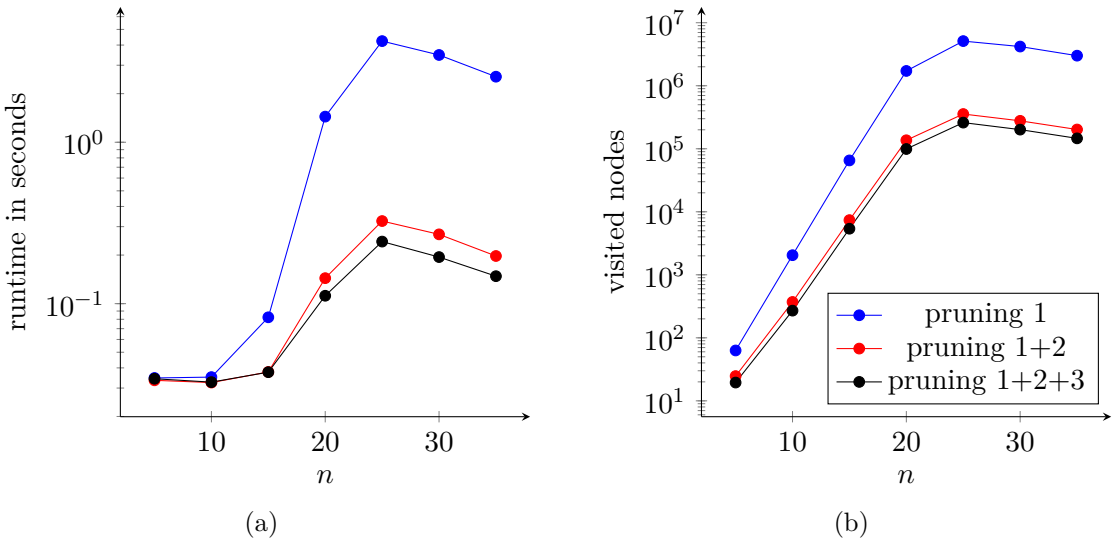


Figure 4: Effect of the pruning methods on the CKK algorithm. Each point is the average of 200 runs with uniformly distributed integers in the range from 1 to 2^{20}.

CGA and CKK are anytime algorithms: they can provide an approximation quickly, and improve the result until optimality, if they are given enough runtime.

```
255 heuristic_result = self.get_heuristics_result(heurstic_instance, log)
256
257 # since the semantics of a branching decision is whether to use the KK
258 # result or not, we have to store the KK decision for each number.
259 for idx in heuristic_result.list_a:
260     direction[idx] = 0
261 for idx in heuristic_result.list_b:
262     direction[idx] = 1
263
264 # A path is a sequence of decisions.
265 # The main information is the decision itself, but for runtime efficiency,
266 # we store the current sum as additional information, so that we don't have
267 # to calculate it again every time.
268 #                path    sums     sum of remaining
269 paths_stack = [([],     [0, 0], sum(numbers))]
270 break_ctr = 0
271 while paths_stack:
272     path, sums, sum_remaining = paths_stack.pop()
273     visited_nodes += 1
274
275     level = len(path)
276     if level != tree_depth:
277         diff = abs(sums[0] - sums[1])
278         if diff > sum_remaining:  # check whether we can prune
279             if best_diff is not None and diff - sum_remaining > best_diff:
280                 continue  # no chance of getting better
281             # put all numbers into the smaller set, this is the best option
282             smaller_idx = 0 if sums[0] < sums[1] else 1
283             new_path = path + [smaller_idx] * (tree_depth-level)
284             sums[smaller_idx] += sum_remaining
285             paths_stack.append((new_path, sums, 0))
286         else:
287             # put both options in a tuple to assign to left and
288             # right branch according to the heuristic.
289             paths = (path + [0], path + [1])
290             sums_0 = [sums[0] + numbers[level], sums[1]]
291             sums_1 = [sums[0], sums[1] + numbers[level]]
292             sums  = [sums_0, sums_1]
293
294             sum_remaining -= numbers[level]
295             paths_stack.append((paths[1-direction[level]],
                ↪  sums[1-direction[level]], sum_remaining))
296             paths_stack.append((paths[  direction[level]], sums[
                ↪  direction[level]], sum_remaining))
297     else:  # we are at a leaf
298         visited_leafs += 1
299         diff = abs(sums[1] - sums[0])
300         if best_diff is None or diff < best_diff:
301             best_diff, best_path = diff, path
302         if best_diff <= 1.1:
303             break_ctr += 1
304             break
```


Listing 4: Implementation of the Complete Karmarkar-Karp algorithm, written in python. Taken from `impl/partitioning_algo/kk.py`

3.5 Kernelized ILP

Even in NP-hard problems some parts of the problem may be easy to decide. It is the goal of kernelization to find those easy parts of an instance and strip them off the hard core. This size of this hard core, called the kernel, can often be bounded polynomially by some parameter.

Etscheid et. al. describe a kernelization for KNAPSACK instances with k different weights that allows to solve them in time $k^{2.5k+o(k)} \cdot \text{poly}(|I|)$, where $|I|$ denotes the encoding length of the instance [5]. When a complete algorithm for KNAPSACK has a runtime exponential in n (or $|I|$), it is now only polynomial in $|I|$ and exponential in the parameter k. The key idea behind this kernelization is to group items with the same weight and reduce the number of variables and constraints. The same idea can be applied to PARTITIONING. It is even more straightforward, because each occurrence of a number is the same, unlike in KNAPSACK, where items with the same weight may have different values.

Given n integers with k different numeric values, let w_j for $j \in [k]$ denote the number of occurrences of value v_j. The kernelized ILP can be stated as

$$\min \left| \sum_{j \in [k]} x_j \cdot v_j - \sum_{j \in [k]} (w_j - x_j) \cdot v_j \right| \qquad \text{s.t. } x_j \in [w_j] \quad \forall j \in [k] \,.$$

3.6 Kernelized CKK

The complete Karmarkar-Karp algorithm (see section 3.4) benefits from the same kernelization as well. If w_j equal numbers v_j are in a problem instance, the traditional CKK will add w_j tree levels to the tree. However, it is also possible to add only one level with $w_j + 1$ branches. As a result the full tree shrinks from $2^{n+1} - 1 = \mathcal{O}(2^n)$ nodes to $\mathcal{O}(n^k)$ nodes.

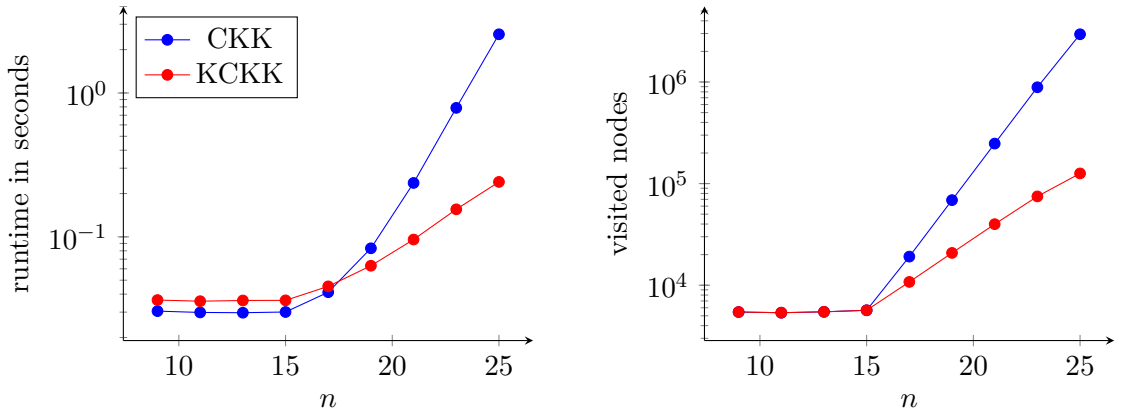


Figure 5: Effect of kernelization on CKK. Each point is the average of 200 runs with integers in the range from 1 to 2^{20}.

Figure 5 shows the algorithm runtime and number of visited nodes for the normal and kernelized version of CKK (KCKK). For small n, each of the $m = 15$ values occurs only once, so that the kernelization has no effect on the number of visited nodes. KCKK has some overhead to build the kernelized instance, and therefore has a longer runtime for small instances. Once $n > w$, KCKK visits less nodes and also has a shorter runtime than CKK.

4 Solving Partition with Bin Packing

BIN PACKING is another NP-complete problem [6] with a variety of applications[2]. The decision variant of BIN PACKING can be stated as follows: Given k bins with capacity b, can a given set S of items with sizes in $(0, b]$ be partitioned into the k bins? Of course, BIN PACKING can be formulated as an optimization problem as well: Given S and b as before, find the packing that minimizes k, the number of used bins.

In 2017, Hoberg and Rothvoss published an approximation algorithm for BIN PACKING based on a method coming from discrepancy theory [10]. Their algorithm has an additive gap of $\log(\mathrm{OPT})$. Can their work be applied to PARTITION as well?

This section starts with a short presentation of Hoberg and Rothvoss' algorithm. We then investigate theoretical bounds for a PARTITION-solver, which uses a BIN PACKING-solver internally. Without strong bounds, we turn to the design and experimental evaluation of such a solver. Finally, the direct applicability of the underlying ideas to PARTITION ´is assessed.

4.1 The Bin Packing algorithm by Hoberg and Rothvoss

The algorithm presented by Hoberg and Rothvoss in [10] will be abbreviated HB. HB is a polynomial-time algorithm that requires $\mathrm{OPT} + \log(\mathrm{OPT})$ bins, where OPT is the optimal solution. It can be sketched as follows:

Formulation Define a pattern (also known as configuration) as a list of items, which fit into one bin. Instead of packing items into bins, the perspective shifts to assigning one pattern to each bin, until all items are covered by the chosen patterns. The goal is of course to minimize the number of bins. This pattern-based view is widely used in the literature.

Relaxation It leads to an ILP, which cannot be solved efficiently for two reasons. ILP itself is NP-hard [6] and the number of variables, i.e. the number of feasible patterns grows exponentially with the number of items. Relaxing the ILP formulation LP yields the so-called Gilmore-Gomory LP relaxation. Despite the exponential number of variables that still exists in the relaxation, it can be solved in practice with column generation and other techniques. Hoberg and Rothvoss approximate the LP solution in polynomial time.

Rounding The main contribution of Hoberg and Rothvoss is the rounding scheme, which is based on an algorithm of Lovett and Meka [14]. Lovett and Meka have presented a randomized algorithm to round a vector $x_{start} \in [0,1]^n$, such that at least half of its entries are in $\{0,1\}$ while constraining how far the rounded vector is away from x_{start}. HB applies this rounding a logarithmic amount of times to round the relaxation result. The key ingredient is to meet the preconditions that the rounding algorithm requires. Small items are removed or rounded up to groups larger items, which are then in turn glued together to be one item in an involved scheme. Afterwards, input for the algorithm of Lovett and Meka is derived. Crucially, each preparation and each rounding adds a gap in $\mathcal{O}(1)$, so that after logarithmically many iterations a gap in $\mathcal{O}(\log n)$ is achieved.

[2] See for example `http://www.ams.org/publicoutreach/feature-column/fcarc-packings3`

4.2 Bounds for Bin Packing-based Partition-solvers

Claim 1. *The decision variant of* PARTITION *can be solved exactly with* BIN PACKING*.*

Proof. Let the bins have size 1. With the sum $S = \sum a_i$, scale the numbers down to a new sum of 2. Now PARTITION is equivalent with BIN PACKING with 2 bins. If and only if a perfect partition exists, each partition has size 1. If and only if a packing is possible, the sum of each bin's content is 1. One case must be excluded: if there is one $a_i > S/2$, there cannot be a packing. But in this case there is clearly no perfect partition. □

Using bin packing to find the optimal partition when there is no perfect partition does not work with the same ease. The idea is the following: if BIN PACKING returns three bins, construct two partitions. One consists of the numbers from one bin, the other partition contains the numbers from the two other bins.

Again, instances in which there is one $a_i > S/2$ are simple to solve. This particular a_i forms one sublist, all other numbers form the other sublist. The remaining observations assume that there is no such a_i.

Claim 2. *The result of* BIN PACKING *in this context, when solved exactly, will never be more than* 3 *bins.*

Proof. The claim is trivial for any instance with $n \leq 3$. Now assume that $n \geq 4$ and that the bin packing solution requires $k \geq 4$ bins. There is a bin with the smallest sum $b_{\min} \leq 1/2$. Then, since the packing is optimal, $k-1$ bins must be filled higher than $1 - b_{\min}$. Then for S, the sum of all bins, we have

$$2 = S > (k-1)(1 - b_{\min}) + b_{\min},$$

which can be transformed to

$$k < \frac{2 - b_{\min}}{1 - b_{\min}} + 1.$$

So with $0 \leq b_{\min} \leq 1/2$, the largest upper bound is 4 at $b_{\min} = 1/2$, and therefore $k < 4$. A contradiction. □

This gives us an upper bound for the discrepancy after transforming the packing into a partition.

Claim 3. *A constructed partition from a* BIN PACKING *solution as above has a maximal discrepancy of* $1/3 \cdot S$*.*

Proof. If the solution uses two bins, the partition is perfect. Constructing the partitions from three bins is a new partition problem with three numbers between 0 and 1 with a total sum of 2, where each number is the sum of one bin.

Choose two numbers as one sublist and the remaining number as the other sublist. Call the sublist with larger sum P and the smaller one p. If $size(P) - size(p) \leq 1/3 \cdot S = 2/3$, then the claim is fulfilled.

Otherwise, $size(P) - size(p) > 1/3 \cdot S = 2/3$. It follows that $size(P) > 4/3$ and $size(p) < 2/3$. As each number is at most 1, P must contain two numbers, of which at least one is larger than $2/3$ and at least one is at most $2/3$. Construct new partitions: The first partition contains the

single element from p and the smaller element from P. Its size is less than $4/3$. The other partition contains the larger element from P. Its size is larger than $2/3$. The new discrepancy is less than $2/3$. □

The discrepancy is usually given as a function of n, not the total sum. Since for any given sequence of n numbers, the sum is in $\mathcal{O}(n)$, the discrepancy is in $\mathcal{O}(n)$. This is a worst case bound. The average performance depends on the properties of the algorithm itself. This bound is tight. Bin packing may distribute the numbers so that the three bins have exactly the same sum. In this case the discrepancy is $2/3 \cdot S$.

Now what happens if we change the scaling, so that the sum of the input to BIN PACKING is larger than 2? The first observation is that the scaling is limited by the largest number in the input, as it must not be scaled to a value > 1. Secondly, we can generalize the bound for k from claim 2.

Claim 4. *A BIN PACKING instance with n numbers $(a_1, \dots a_n) \in (0, 1]$ with $S = \sum_{i=0}^{n} a_i \leq n$ can be packed in $k < 2 \cdot S$ bins.*

Proof. In an optimal packing with k bins, there is a smallest bin with the filled size $b_{\min} \leq S/k$. Since the packing is optimal, all other bins must be filled higher than $1 - b_{\min}$. Then for S we have

$$S > (k-1)(1 - b_{\min}) + b_{\min},$$

which can be transformed to

$$k < \frac{S - b_{\min}}{1 - b_{\min}} + 1.$$

It suffices to check the inequality for $k > S$, since a packing with less than $\lceil S \rceil$ bins is impossible. The inequality holds for

$$S \leq k < 2 \cdot S.$$

□

If we construct a PARTITION result from such a BIN PACKING result, it would be desirable to derive bounds for the PARTITION result. One approach towards such a bound are the minimum and maximum bin size. The closer these bounds are together, the better the discrepancy will be. The maximum bin size is obviously 1. The minimum bin size $b_{\min}$ is at most S/k, for which we have $0.5 < S/k \leq 1$. However, only a lower bound for $b_{\min}$ is useful.

Another approach goes as follows: there is a largest bin $b_{\max}$ with $b_{\max} \leq S/k$. If all bins are at most $b_{\max}$, then for each bin we have a size $b_i \geq S - (k-1)\, b_{\max}$, and therefore $b_{\min} \geq S - (k-1)\, b_{\max}$. Again, this bound turns out to be very weak.

4.3 Solve Partition with Bin Packing solver

The PARTITION solver from the last section perform far above their worst case bounds in most cases. It is therefore valuable to examine the performance of as BIN PACKING-based solver in practice.

Such a solver looks as follows:

1. Input scaling

2. Solving the BIN PACKING instance
3. Transforming the BIN PACKING solution to a partition.

Let's consider the design choices of each step and its influence on the discrepancy. There are tight links between the steps. Discussing them in reverse order eases the explanation.

Solution transform After solving the packing instance, k bins filled with numbers must be transformed into a partition. As this is a smaller instance of PARTITION, the known approximation algorithms or an exact solver can be applied. Of course, with few bins the resulting discrepancy relies on evenly filled bins, which cannot generally be expected from BIN PACKING solvers. Therefore the items from the last two bins are used as stuffing material to cover up the roughness of partitioning just a few bins. The input to the PARTITION solver consists of the sizes of all but the last two bins, and the items from the last two bins. The more items are used for stuffing, the better the result will be, but also additional computational effort is required.

The first design choice is the PARTITION solver. Figure 7 shows the difference in both the result and algorithm runtime of using KK and CKK. While the discrepancy benefits from CKK, the computational effort is quite large. Figure 6 shows the effect of stuffing. Clearly, some stuffing is needed. The graph also shows a variation, where the largest and smallest bin where taken as stuffing in order to reduce the variance of bin sizes for the partitioning, with only minor improvements.

Solving Bin Packing Simple heuristics like the First Fit algorithm from the python library prtpy[3] yield a performance close to the optimum for the instances here and are therefore sufficient. When the number of bins grows slowly with $\log_{10} n$, even at large PARTITION instances of $n > 10^5$, the sum S is $\log_{10}(10^5) = 5$. With an upper bound of $\lfloor 1.7 \cdot OPT \rfloor$ for the First Fit algorithm, there is small room for improvement.

Input scaling Input scaling has a decisive influence on the result. The numbers from the original problem must be scaled down to the range $(0, 1]$. Also, the scaling determines the total sum S and therefore how many bins the packing will need. [4] Having too few bins constrains the partitioning in the last step. Having too many bins constrains the packing, because the numbers grow relative to the bin capacity. We tested several functions to determine the scaling: constants, $\sqrt{n}$, polynomials of n and logarithms of n to various bases. Figure 9a shows that $\log_{10}(n)$ performs best. The three stages must be adjusted to each others behaviour for a good overall performance. In particular, the used algorithms do not perform well if the total sum is just above an integer. In this case, the variance in bin sizes increases and the resulting discrepancy increases as well. A very simplistic remedy, at least for the inputs considered in our experiments, is to use $\lfloor f(n) \rfloor + 0.99$, where $f(n)$ is the function to get a total sum from the problem size. A remnant of this effect is still visible in figure 6 at $n = 10^4$.

The experiments use large numbers and large instances. Large instances are necessary to get a glimpse on the practical runtime behaviour. Large numbers are necessary to use the algorithms in the hard phase of PARTITION. In the easy phase it suffices to use the Karmarkar-Karp algorithm, as it gets very close to the optimal results when $\kappa > 1$.

[3] `https://pypi.org/project/prtpy/`

[4] Note that scaling the bin capacity allows for cleaner code in our experimental code and therefore is utilized in the code.

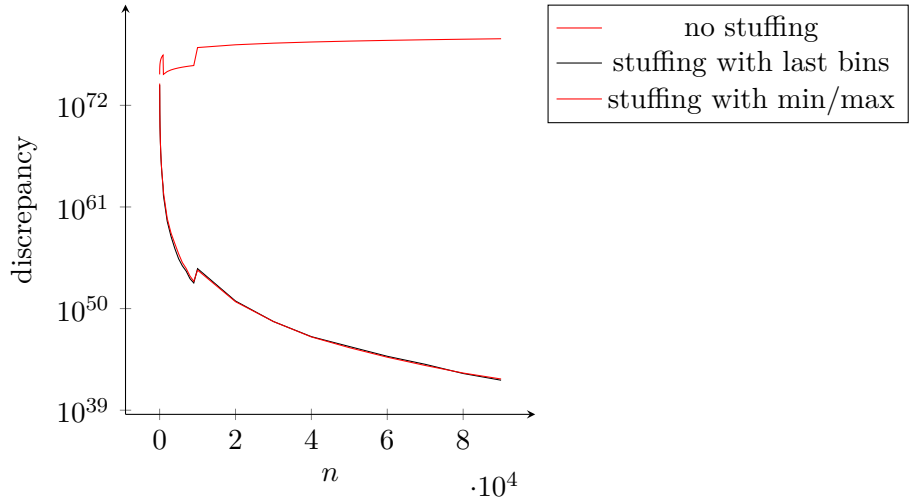


Figure 6: Comparison of different stuffing methods, using $\log_{10}$ for scaling and KK as partition solver. Each point is the average of 100 runs with integers in the range from 1 to 2^{250}.

In total, the best variant shows a comparable performance to the Karmarkar-Karp algorithm. It yields higher discrepancies, but is a bit faster at large instances.

4.4 Using ideas from HB

In the presented algorithm, the bin packing algorithm has only a small impact on the result. But can the ideas behind HB, most notably the algorithm by Lovett and Meka, help for a PARTITION solver? First, an efficiently computable relaxation of PARTITION is required. Unfortunately, the problem has very little structure, what makes it difficult find a relaxation similar to the Gilmore-Gomory LP.

Of course, the ILP from section 3.1 can be relaxed to an LP of the following form:

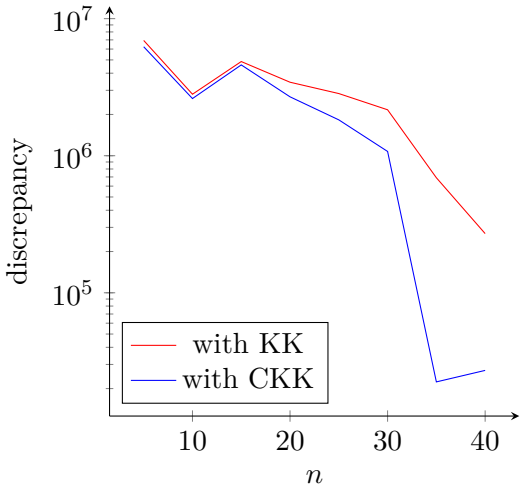
$$\min |D(A)| \qquad \text{s.t. } x_i \in \mathbb{R} \quad \forall i \in [n],$$

with

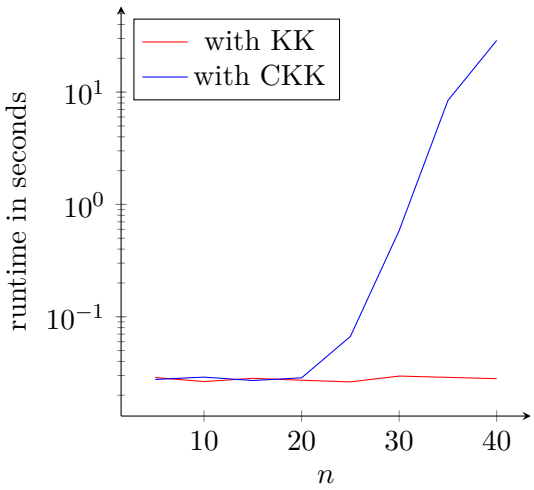
$$D(A) = \sum_{i \in [n]} a_i \cdot x_i - \sum_{i \in [n]} a_i \cdot (1 - x_i).$$

Assign each a_i to one sublist if $x_i < 0.5$ and to the other sublist otherwise. The solver can set $x_i = 0.5$ for $i \in [n]$ and reaches $D(A) = 0$ without giving a useful result.

The gurobi LP solver [8] exhibits this behaviour in our experiments. Interestingly, it must be noted that the numeric range for x_i influences the result. When the variable bounds are shifted to $x_i \in [-0.5, 0.5]$ and the sublist construction is modified accordingly, an obvious solution is possible ($x_i = 0$ for $i \in [n]$), but the solver does approximate a partition with about the same quality (but higher runtime) as the simple greedy algorithm at least in the setting shown in figure 8. A closer examination of this phenomenon exceeds the scope of this work.

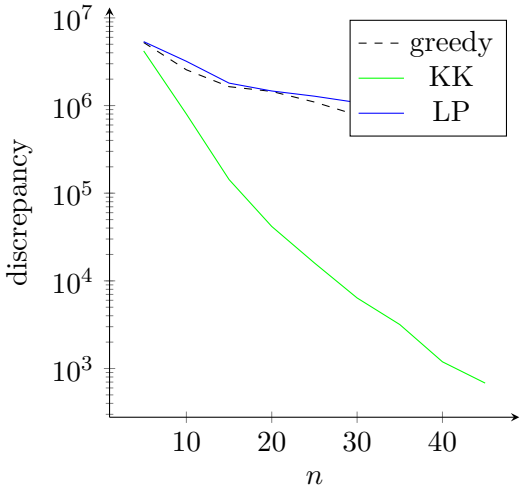


(a) Discrepancy comparison, using $\log_{10}$



(b) Runtime comparison, using $\log_{10}$

Figure 7: Comparison of approximation or exact solving (with KK and CKK) for solution transform. Each point is the average of 100 runs with integers in the range from 1 to 2^{250}.



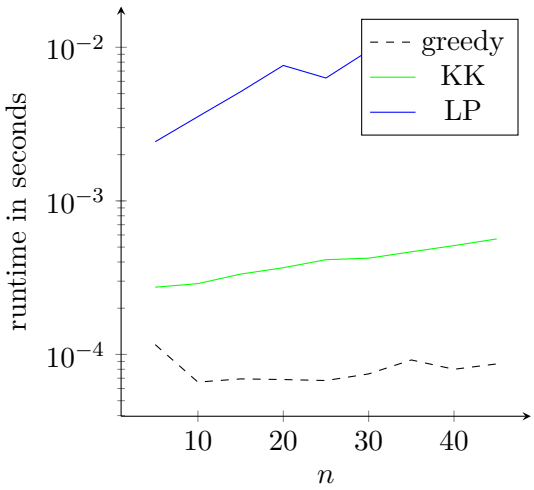


Figure 8: Performance of an LP based PARTITION solver, compared to the greedy and Karmarkar-Karp algorithms. Each point is the average of 100 runs with integers in the range from 1 to 2^{30}.

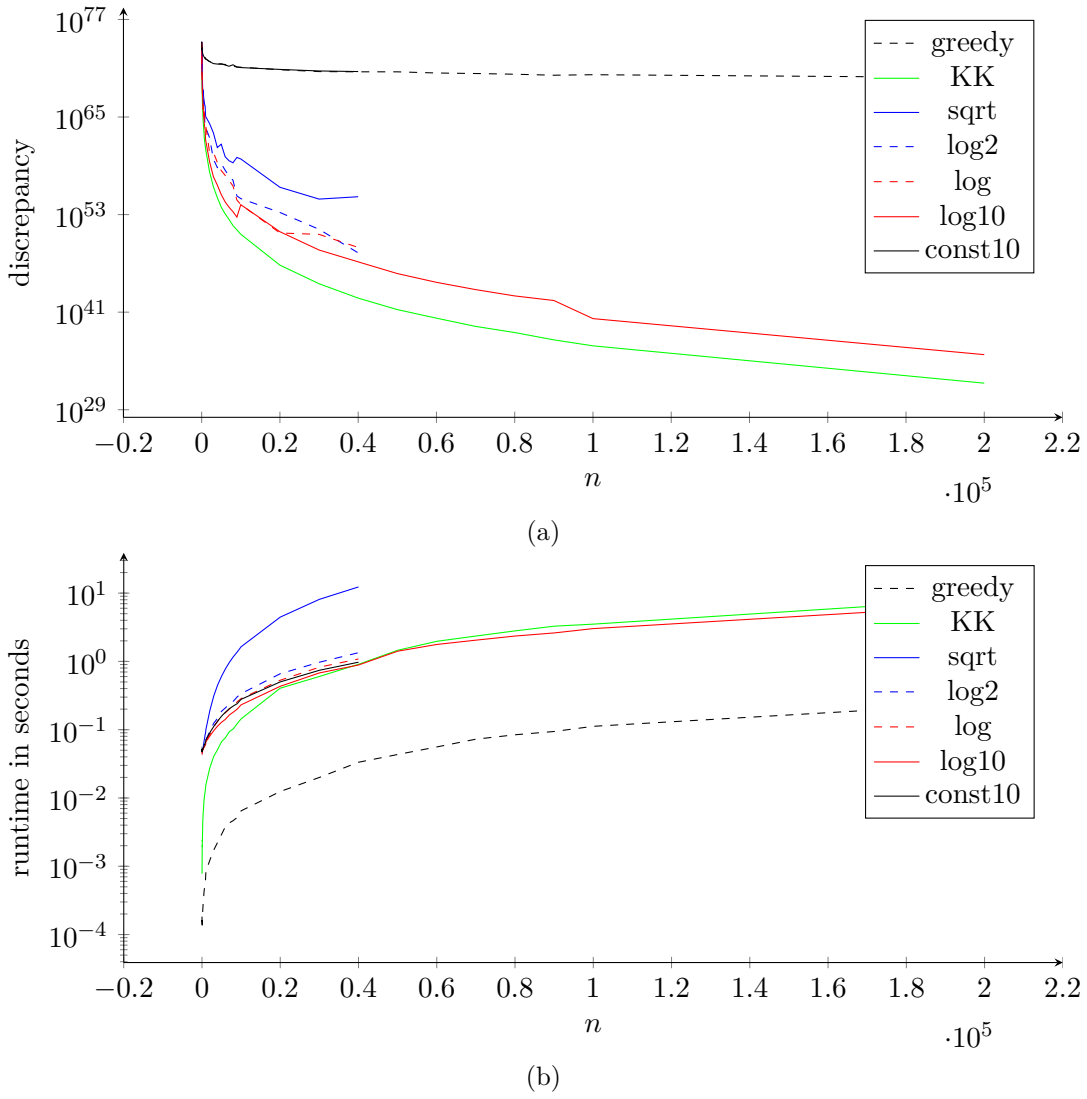


Figure 9: Discrepancy and runtime comparison for different scaling functions. The greedy and Karmarkar-Karp are also given for reference. Each point is the average of 100 runs with integers in the range from 1 to 2^{250}.

5 Experimental setup

5.1 Implementation

All algorithms mentioned in the previous sections have been implemented in python[5]. Python is a very widespread interpreted language with a comprehensive standard library and a huge ecosystem of libraries for utilities as well as mathematics. Python is very well suited for rapid prototyping. Using python means accepting a slow execution and high memory overhead especially compared to compiled languages like C++, but easy and fast development is the most important criterion for this thesis.

The algorithm implementations are published at `https://github.com/llueder/partitionpy`. For the original thesis, an examination framework has been developed. It is not in a stage to be published... The framework allows to

- generate random instances with configurable size, value range and number of different values
- quickly choose which algorithms to run on instances
- save solved instances to a file for later analysis
- generate plots of metrics for several algorithms over n or the value range and save the plot data to file.

5.2 Generation of Partition instances

Each instance consists of n integers, sampled from a uniform distribution on $[1, m]$ using the python `randint`[6] function. The instance generation has been designed similar to [13][17] (and [1], where real numbers on the unit interval are used). When a given number of different values is required, first the values are chosen from a uniform distribution on $[1, m]$ and then the required number of numbers is sampled from a uniform distribution on the set of values.

5.3 Algorithm execution

All algorithms in a comparison solve the same problems. For each problem configuration, several instances are used and the results are averaged within each configuration. The algorithms are executed on a hexacore AMD Ryzen 5 1600 processor running linux. Up to four parallel processes were used. No other processes with significant resource usage were active. Memory swapping was disabled. Other side effects (e.g. caching, temperature) are neglected.

Python's standard types are used, including the `float` type. It is a 64 bit IEEE754 floating point representation with a maximum value of about 1.8×10^{308}. With up to 100000 numbers up to 2^{250} involved in an instance, the maximum numbers can be expected in the order of 1×10^{80}, so that no special handling of numerical/precision issues has been implemented.

[5] `https://www.python.org/`
[6] `https://docs.python.org/3/library/random.html#random.randint`

6 Conclusion

We have presented a simple, yet effective kernelization for instances with a limited number of values as well as an attempt to use a recent BIN PACKING solver to solve PARTITION. While this algorithm performs better than the very simple greedy heuristic, it cannot match the Karmarkar-Karp algorithm. While at the input sizes used in this work a simple BIN PACKING heuristic suffices, at very large instances a better BIN PACKING approximation such as the one presented by Hoberg and Rothvoss may have a stronger effect on the result. Time constraints and implementation issues with their involved algorithm made prevented experiments on that matter. PARTITION proves yet again that it is a very hard problem, at least in its hard phase.

References

[1] S. Boettcher and S. Mertens. Analysis of the karmarkar-karp differencing algorithm. *The European Physical Journal B - Condensed Matter and Complex Systems*, 65:131–140, 09 2008. doi:10.1140/epjb/e2008-00320-9.

[2] C. Borgs, J. Chayes, and B. Pittel. Sharp threshold and scaling window for the integer partitioning problem. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, STOC '01, page 330–336, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133499. doi:10.1145/380752.380854.

[3] C. Borgs, J. T. Chayes, S. Mertens, and B. Pittel. Phase diagram for the constrained integer partitioning problem. *Random Structures & Algorithms*, 24(3):315–380, 2004. doi:https://doi.org/10.1002/rsa.20001.

[4] V. S. Denchev, S. Boixo, S. V. Isakov, N. Ding, R. Babbush, V. Smelyanskiy, J. Martinis, and H. Neven. What is the computational value of finite-range tunneling? *Phys. Rev. X*, 6: 031015, Aug 2016. doi:10.1103/PhysRevX.6.031015.

[5] M. Etscheid, S. Kratsch, M. Mnich, and H. Röglin. Polynomial kernels for weighted problems. *J. Comput. System Sci.*, 84:1–10, 2017. ISSN 0022-0000. doi:10.1016/j.jcss.2016.06.004. URL `https://doi.org/10.1016/j.jcss.2016.06.004`.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990. ISBN 0716710455.

[7] I. P. Gent and T. Walsh. Phase transitions and annealed theories: Number partitioning as a case study. In W. Wahlster, editor, *12th European Conference on Artificial Intelligence, Budapest, Hungary, August 11-16, 1996, Proceedings*, pages 170–174. John Wiley and Sons, Chichester, 1996.

[8] Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2023. URL `https://www.gurobi.com`.

[9] B. Hayes. Computing science: The easiest hard problem. *American Scientist*, 90(2):113–117, 2002. ISSN 00030996. URL `http://www.jstor.org/stable/27857621`.

[10] R. Hoberg and T. Rothvoss. A logarithmic additive integrality gap for bin packing. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, page 2616–2625, USA, 2017. Society for Industrial and Applied Mathematics. URL `https://dl.acm.org/doi/10.5555/3458064.3458166`.

[11] N. Karmarkar and R. M. Karp. The differencing method of set partitioning. Technical Report UCB/CSD-83-113, EECS Department, University of California, Berkeley, 1983. URL `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1983/6353.html`.

[12] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972. ISBN 978-1-4684-2001-2. doi:10.1007/978-1-4684-2001-2_9.

[13] R. E. Korf. A complete anytime algorithm for number partitioning. *Artif. Intell.*, 106(2): 181–203, dec 1998. ISSN 0004-3702. doi:10.1016/S0004-3702(98)00086-1.

[14] S. Lovett and R. Meka. Constructive discrepancy minimization by walking on the edges. *SIAM J. Comput.*, 44(5):1573–1582, 2015. ISSN 0097-5397. doi:10.1137/130929400.

[15] S. Mertens. Phase transition in the number partitioning problem. *Phys. Rev. Lett.*, 81: 4281–4284, Nov 1998. doi:10.1103/PhysRevLett.81.4281.

[16] S. Mertens. The easiest hard problem: number partitioning. In *Computational Complexity and Statistical Physics*. Oxford University Press, 12 2005. ISBN 9780195177374. doi:10.1093/oso/9780195177374.003.0012.

[17] E. L. Schreiber, R. E. Korf, and M. D. Moffitt. Optimal multi-way number partitioning. *J. ACM*, 65(4), jul 2018. ISSN 0004-5411. doi:10.1145/3184400.

[18] B. Yakir. The differencing algorithm ldm for partitioning: A proof of a conjecture of karmarkar and karp. *Mathematics of Operations Research*, 21(1):85–99, 1996. doi:10.1287/moor.21.1.85.

Declaration of originality

I hereby confirm that I have written this thesis independently and have not used any resources other than those specified. The parts of the work that are taken from other works (this also includes internet sources) in wording or in spirit have been identified and the source has been indicated.