Heuristics and Exact Methods for Number Partitioning

João Pedro Pedroso and Mikio Kubo

Technical Report Series: DCC-2008-3

Heuristics and Exact Methods for Number Partitioning

João Pedro Pedroso and Mikio Kubo

February 2008

Abstract

Number partitioning is a classical NP-hard combinatorial optimization problem, whose solution is challenging for both exact and approximative methods. This work presents a new algorithm for number partitioning, based on ideas drawn from branch-and-bound, breadth first search, and beam search. A new set of benchmark instances for this problem is also proposed. The behavior of the new method on this and other test beds is analyzed and compared to other well known heuristics and exact algorithms.

1 Introduction

The number partitioning problem (NPP) is a classical NP-hard problem of combinatorial optimization, with important applications in areas like public key encryption and task scheduling. It has been studied under many perspectives, from the theoretical analysis of "easy-hard" phase transition observed in combinatorial problems, to its approximative solution with heuristics and metaheuristics.

The NPP is the following: given a set (or possibly a multiset) of N positive integers $\mathcal{A} = \{a_1, a_2, \ldots, a_N\}$, find a partition $\mathcal{P} \subseteq \{1, \ldots, N\}$ that minimizes the discrepancy

$$E(\mathcal{P}) = \left| \sum_{i \in \mathcal{P}} a_i - \sum_{i \notin \mathcal{P}} a_i \right| .$$

Partitions such that $E = 0$ or $E = 1$ are called perfect partitions. A decision problem related to the NPP is that of determining if there is a perfect partition. This problem is NP-complete: polynomial transformations from Satisfiability→3-Satisfiability→3-Dimensional Matching→Partition (decision problem), are presented in [7]. Therefore, the NPP is NP-hard.

An important characteristic of this problem is that its computational complexity depends on the type of input numbers $a_1, a_2, \ldots, a_N$. If a_j are all positive integers bound by a constant A, then the discrepancy E can take on at most NA different values, and the size of the search space is $\mathcal{O}(NA)$; a so-called *pseudo-polynomial time algorithm* is presented in [7] for this case. However, a concise representation of the input is of length only $\mathcal{O}(N \log A)$, and NA is not bounded by any polynomial function of that quantity. This property implies that NP-hardness of the NPP depends on exponentially large input numbers (or exponentially high precision, if the a_j's are real numbers). It is also connected to a phenomenon known as *phase transition*, describing an abrupt change in the time required for solving the problem at a particular size of the input. If the numbers a_j are independently and identically distributed random numbers bounded by $A = 2^{\kappa N}$, the solution time abruptly raises at a particular value $\kappa = \kappa_c$; this is due to the fact that there is a high probability of having perfect partitions for $\kappa < \kappa_c$, and this probability is close to 0 for $\kappa > \kappa_c$. See [13] for a short introduction to this property.

Concerning the optimum value of the discrepancy, in [10] it was shown that for a random instance I_N, with N real valued items drawn independently from a uniform distribution in $[0, 1)$, the optimum discrepancy $\mathrm{opt}(I_n)$ has an expected value $\mathcal{O}(\sqrt{N}/2^N)$.

Practical applications of the NPP include multiprocessor scheduling [4], public key cryptography [12], and the assignment of tasks in low-power application-specific integrated circuits [5]. The application of metaheuristics to the solution of this problem was initially analyzed in [8], where the difficulty of finding an appropriate neighborhood structure was reported. Different strategies were tried in [1] (a different formulation of tabu search) and [2] (a memetic algorithm with the recombination operator based on weight-matching); we use their results as a term for comparison.

The contributions of this paper are the following:

- In section 2.3 we describe a new branch-and-bound method, based on breadth-first search, with a variant for controlled incomplete search.
- In section 2.3.1 we propose the use of a diving approach for determining an upper-bound for each node in breadth-first branch-and-bound.
- An extended test for checking if the lower bound was reached in branch-and-bound, allowing in some cases to prove earlier that an optimum was reached, is presented in section 2.3.2.
- Results obtained by two incomplete branch-and-bound methods are presented in section 3, and compared to other heuristics, using convenient metrics described in section 3.1.
- We propose a set of challenging benchmark instances, which can be used as a term for comparison in future works, in section 3.2.

2 Solution approaches

2.1 Simple heuristics

As the NPP is an NP-hard problem, exact solution with known algorithms is only possible for small instances. For other instances, approximative but fast heuristics are commonly used.

One possible greedy heuristic approach, known as the longest processing time heuristics for the multi-processor scheduling problem, consists of placing the largest unassigned number into the subset with the smallest sum thus far, until all the numbers are assigned. For this algorithm, the worst situation occurs when the two subsets are balanced before assigning the last number. If the input numbers a_j are real values uniformly drawn from $[0, 1)$, this heuristics leads to a discrepancy $\mathcal{O}(N^{-1})$. As the algorithm requires sorting the numbers, its complexity is dominated by the sorting algorithm; thus, it runs in $\mathcal{O}(N \log N)$ time.

We will use throughout this paper the example provided in [11], consisting of the instance $\mathcal{A} = \{8, 7, 6, 5, 4\}$. When applied to this set, the greedy heuristics leads to the partitions $\{8, 5, 4\}$ and $\{7, 6\}$ with discrepancy 4.

The best polynomial time heuristics known to date is the differencing method of Karmarkar and Karp (the KK heuristics) [9]. It consists of successively replacing the two largest numbers by the absolute value of their difference and placing those items in separate subsets, but without actually fixing the subset into which each number will go yet. This heuristics is described in Algorithm 1, which determines a tree with indices for elements of $\mathcal{A}$ as vertices. Each edge in this tree indicates that the corresponding vertices should be in separate partitions. The decision concerning fixing *which* partition each number goes is taken at the end, when there is only one number left (which is the discrepancy between the two partitions). The partition is obtained through a 2-coloring of the tree, which can be done in linear time.

Algorithm 1: Karmarkar and Karp heuristics.

```
KK($\mathcal{A}$)
(1)   create a vertex $i$, $\forall a_i \in \mathcal{A}$; add label$_i := a_i$ to vertex $i$
(2)   $\mathcal{E} := \{\}$                                   ←edge set
(3)   while there is more than one labeled vertex:
(4)       $u, v :=$ vertices with the two largest labels
(5)       $\mathcal{E} := \mathcal{E} \cup \{\{u, v\}\}$
(6)       remove label from vertex $v$
(7)       set label$_u$ := label$_u -$ label$_v$
(8)   return discrepancy (last label) and edge set $\mathcal{E}$
```

When applied to the set $\mathcal{A} = \{8, 7, 6, 5, 4\}$, the KK heuristics leads to the partitions $\{8, 6\}$ and $\{7, 5, 4\}$ with discrepancy 2, as can be seen in Figure 1; as occurs in this example, the solution of the KK heuristics is usually better than the greedy solution, but is still suboptimal.

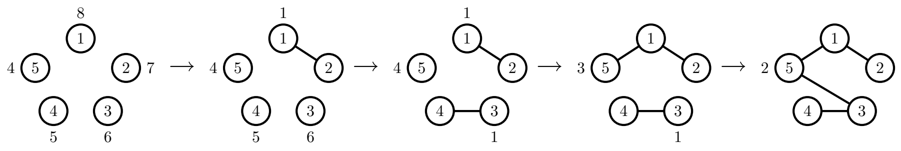


Figure 1: Graph created while applying the KK heuristics to the set $\{8, 7, 6, 5, 4\}$. Each edge connects vertices that will be in different partitions.

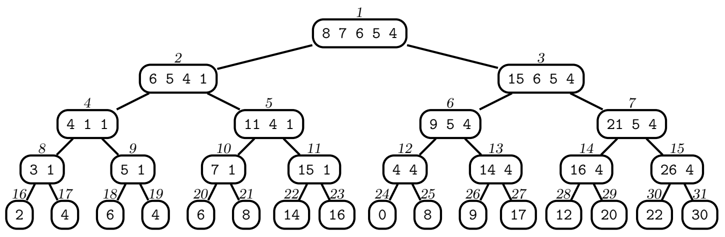


Figure 2: Search tree for the complete differencing method.

If each input number a_j is a real value uniformly drawn from $[0, 1)$ the KK heuristics leads to an expected discrepancy $\mathcal{O}(N^{-\alpha \log N})$, for some positive constant α [16]. This bound is considered as rather unsatisfying [13], comparing to many other NP-hard problems for which good approximative algorithms exist.

2.2 Exact methods

On each step of the KK heuristics the two largest numbers are replaced by their difference. On a complete search algorithm based on the KK heuristics, the alternative of replacing them by their sum must also be considered, as illustrated with the previous example in Figure 2. As in the case of the KK heuristics, the construction of the partition is based on a graph where each vertex corresponds to an element in the initial set. In the case of the exact method, to each leaf in the branch-and-bound tree there corresponds a graph with two kinds of edges: a set $\mathcal{E}$ for which $\{u, v\} \in \mathcal{E}$ if the label of u was replaced by the difference $u - v$, and a set $\mathcal{F}$ such that $\{u, v\} \in \mathcal{F}$ if the label of u was replaced by the sum $u + v$.

As in the case of the KK heuristics, the reconstruction of the partition can be made in linear time, e.g. by means of Algorithm 2. When this algorithm is first called, the parameters $\mathcal{A}, \mathcal{B}$ are empty sets, and u is any vertex in $1, \ldots, N$. Then, u will inserted in set $\mathcal{A}$, and all vertices connected to u by edges in $\mathcal{F}$ will be recursively inserted in $\mathcal{A}$. All vertices connected to u by edges in $\mathcal{E}$ will be recursively inserted in $\mathcal{B}$. At the end, the sets $\mathcal{A}$ and $\mathcal{B}$ they will hold the two partitions.

Algorithm 2: Reconstruction of the partitions based on the graph generated by the complete differencing method.

```
RECONSTRUCT($\mathcal{E}, \mathcal{F}, \mathcal{A}, \mathcal{B}, u$)
(1)     $\mathcal{A} := \mathcal{A} \cup \{u\}$
(2)     foreach $v : \{u, v\} \in \mathcal{F}$:                ←add these nodes to $\mathcal{A}$
(3)         $\mathcal{F} := \mathcal{F} \backslash \{\{u, v\}\}$
(4)         RECONSTRUCT($\mathcal{E}, \mathcal{F}, \mathcal{A}, \mathcal{B}, v$)
(5)     foreach $v : \{u, v\} \in \mathcal{E}$:                ←add these nodes to $\mathcal{B}$
(6)         $\mathcal{E} := \mathcal{E} \backslash \{\{u, v\}\}$
(7)         RECONSTRUCT($\mathcal{E}, \mathcal{F}, \mathcal{B}, \mathcal{A}, v$)
(8)     return partitions $\mathcal{A}, \mathcal{B}$
```

Searching the whole binary tree in the complete differencing algorithm corresponds to exploring the 2^N possible partitions. This exploration can be made *breadth first* (in Figure 2, one possibility for doing this corresponds to visiting the nodes in the order *1, 2, 3, . . . , 31*), or *depth*

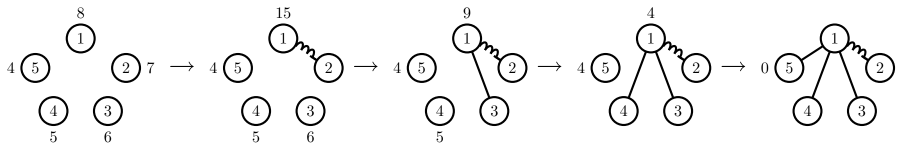


Figure 3: Graph created while applying branch-and-bound to the set $\{8, 7, 6, 5, 4\}$. This picture represents the steps followed for creating the optimal partition, i.e, the path $1 \rightarrow 3 \rightarrow 6 \rightarrow 12 \rightarrow 24$ in Figure 2. Edges ∿∿ connect vertices to be put the same partition, and —— connect vertices to be put in different partitions.

first (which, if we decide to try first differencing and then summing in each node, corresponds to visiting the nodes *1, 2, 4, 8, 16, 17, 9, 18, 19, 5, . . .* in Figure 2).

Figure 3 illustrates the graph corresponding to the optimal solution, as obtained by the branch-and-bound algorithm. Edges with the shape ∿∿ connect vertices that will be in the same partition, and —— edges connect vertices that will be in different partitions. The optimal solution is the partition $\{8, 7\}$ and $\{6, 5, 4\}$, which is a perfect partition.

In the worst case, the complete differencing method has exponential time complexity. However, it is possible to prune parts of the search tree, using the following properties:

1. The KK heuristics is exact for partitioning 4 or less numbers. Thus, when there are only 4 numbers left, only the differencing branch should be taken.
2. If the absolute value of the difference between the largest number and the sum of the others is 0 or 1, a perfect solution was found, and the algorithm can be stopped.
3. If the difference between the largest number and the sum of the others is greater than 1, the best solution in this subtree is to place the largest number in one set and all the other numbers in the other; branching in this subtree can be stopped.

2.3 Proposed methods

Both depth-first and breadth-first branch-and-bound can be used as heuristics, if only a part of the search tree is explored. Although the idea is rather trivial, and was presented for depth-first search in [11], it seems to have been neglected in the more recent literature, even though the results seem to be much better than those of other approximative methods.

The methods we analyze are the following:

DF – depth-first branch-and-bound; in each node, follow first the child corresponding to the difference and then the one corresponding to the sum. Stop when the limit CPU time has been used. This method was initially proposed in [11].

BF – breadth-first branch-and-bound; in each depth of the search tree, sort the nodes according to the number of times there was a sum (i.e., the right side branch was taken) in the branch-and-bound tree until reaching the node. Stop if the limit CPU time has been used.

The order followed for exploring nodes at the same level of the branch-and-bound tree in BF is based on a suggestion found in [11], which proposes an exploration of the search tree in such a way that the minimum deviations from the KK heuristics are tried first. BF can be parameterized, by limiting the number of nodes that are searched in each depth of the tree; this is a variant of *beam search*, proposed for a scheduling problem in [15]. In our implementation, at each depth we sort the nodes in increasing order according to the number of deviations from the KK heuristics (i.e., the number of sums that were done from the root to the current node), and select the first α nodes; the remaining are neglected. Let us consider the example of Figure 2, without pruning, for BF with $\alpha = 3$. Then, at root level there will be one node (*1*), and at depth 1 there will be nodes *2,3.* At depth 2, node 7 will be neglected (as it had 2 sums (i.e., the right side branch has been taken two times), more than the other three at this depth. At depth 3 nodes *8* (zero sums) and nodes *9,10,12* (one sum) are selected, and nodes *11,13,14,15* are neglected.

Algorithm 3 describes the procedure used in the BF branch-and-bound, representing each branch by the corresponding set of elements. Details concerning keeping the graph structure required for reconstructing the solution are omitted for clarity; a complete, functional version in the Python language is provided in [14].

Algorithm 3: BF branch-and-bound.

```
BF($\mathcal{A}, \alpha, T$)
(1)     $E^* = \mathrm{KK}(\mathcal{D})$                              ←best discrepancy found so far
(2)     $B := \{\mathcal{A}\}$                                 ←initial single branch is the input set
(3)     while B != {} and CPU time used $< T$
(4)         $C := \{\}$                                        ←non cut branches
(5)         foreach $\mathcal{B} \in B$                                 ←$\mathcal{B}$ is a sorted list of items
(6)             $b$ = largest element of $\mathcal{B}$
(7)             $r$ = sum of all items in $\mathcal{B}\backslash\{b\}$
(8)             if $|b-r| = 0$ or $|b-r| = 1$
(9)                 return $|b-r|$                             ←found perfect partition
(10)            if $b \geq r$                                  ←best possible partition is $\{b\}, \mathcal{B}\backslash\{b\}$
(11)                if $b - r \leq E^*$                           ←improved best found solution
(12)                    $E^* = b - r$
(13)            else
(14)                $C := C \cup \{\mathcal{B}\}$                          ←keep $\mathcal{B}$ branch alive
(15)        $B := \{\}$                                        ←prepare new set or branches
(16)        $C := \{\alpha$ elements of $C$ with less sums in their path$\}$
(17)        foreach $\mathcal{B} \in C$
(18)            $b_1 :=$ largest element of $\mathcal{B}$
(19)            $b_2 :=$ second-largest element of $\mathcal{B}$
(20)            $\mathcal{C} := \{b_1 - b_2\} \cup \mathcal{B}\backslash\{b_1, b_2\}$            ←differencing branch
(21)            $\mathcal{D} := \{b_1 + b_2\} \cup \mathcal{B}\backslash\{b_1, b_2\}$            ←summing branch
(22)            $B := B \cup \{\mathcal{C}, \mathcal{D}\}$                      ←add the two new branches
(23)            if $E = \mathrm{KK}(\mathcal{D})$                          ←left-side dive (KK heuristics)
(24)                if $E < E^*$                               ←update best found solution
(25)                    $E^* = E$
(26)        return $E^*$
```

2.3.1 Diving

When using BF branch-and-bound on large problems, with large values of α, the leaves of the branch-and-bound tree may not be reached within the allowed CPU time; in this case we still want to have a solution at the moment the process is interrupted. As a solution (and, thus, an upper bound, also used for pruning) is useful at any stage of the search, we implement a variant of *diving* for obtaining it. Diving is common, for example, in mixed-integer programming, where the solution of a linear programming relaxation is used for selecting a path in the branch-and-bound tree (see e.g. [3], for a description of several variants of diving in the context of mixed-integer programming).

In the context of the NPP, an interesting diving variant corresponds to computing the KK heuristics' solution at each node. This stands for selecting the differencing (left-side) branch in each node, until reaching a leaf; we call this strategy *left-side diving.* In general the quality of this heuristic solution is very good, and the computational cost for determining it is reasonable.

In Algorithm 3 left-side diving is done in line 23; this is only necessary for the right side child of each node, as for the left side branch the diving solution is the same as the one of the parent. After a left-side diving, a new upper-bound may have been found, and thus the best found solution may have to be updated (line 24).

2.3.2 Testing optimality

Both branch-and-bound algorithms may provide proven optimal solutions. DF is exact if the whole search tree can be searched within the allowed time. As for BF, it is exact if all the nodes of each depth are explored (excluding those that could be cut) within the allowed time. This is possible only for small instances. On both variants, if a solution with discrepancy equal to the lower bound (i.e., a perfect partition) is found, it is also a proven optimum.

Usually, the test for a perfect solution is done only when, at some point of the search, the largest element of the set is larger than the sum of the others [11]. In this case, the best solution that can be obtained is placing the largest element in one partition and all the others in the other.

If the difference between those values is zero or one, an optimal solution was found. However, an optimum was also found, for the same reason, if the largest element is *smaller in one unit* than the sum of the others; the algorithm can successfully stop in this case too. This extended test is implemented in both DF and BF variants; for the latter, it corresponds to lines 8 and 9 of Algorithm 3.

3 Computational results

3.1 Metrics used for reporting results

As the numbers used in all challenging instances of the NPP are extremely large, they are very difficult to read. For analyzing the results of the algorithms, and comparing them to other algorithms found in the literature, we propose using the following metrics:

$$\eta(n) = \log_2(n+1). \tag{1}$$

The value $\eta(n)$ is a convenient and visually effective way of presenting very large values of n. It also preserves the output for perfect partitions: for discrepancies $E = 0$ and $E = 1$, the respective values $\eta(E)$ are still 0 and 1.

3.2 Benchmark instances

For the analysis of the behavior of the algorithms presented we propose a series of benchmark instances, with numbers generated randomly. We shall start by introducing the formula that determines the phase transition [13], for numbers a_j bounded by $A = 2^{\kappa N}$:

$$\kappa_c = 1 - \frac{\log_2 N - \log_2(\pi/6)}{2N}.$$

According to [6] there is an expected exponential number of perfect partitions for $\kappa < \kappa_c$, and no expected perfect partition for $\kappa > \kappa_c$.

We generated instances with several different sizes, ranging from sets $\mathcal{A}$ with ten elements ($N = 10$) to a thousand ($N = 1000$). The first series of instances, named *easy*, consists of sets with numbers drawn below the critical point: for an instance with N elements, each number was drawn with $N/2$ random bits. The second series, named *hard* consists of numbers with N random bits, thus above the critical point. Therefore, there is a high probability that there exist many perfect partitions in the *easy* instances, and a low probability that there is a perfect partition in the *hard* series.

Notice that the numbers in the instances generated are very large; except for the smaller instances, they do not fit in the size of an integer in usual CPUs, and thus require software capable of dealing with very large numbers (we used the Python language, which automatically adapts the precision used in the computations, as required for avoiding integer overflow).

For the purpose of comparison, we also used instances created by independently generating D decimal digits and concatenating them, as proposed in [2]. We call these the D instances.

All the instances used are available in [14]; the algorithms used and the complete output of this computational experience are also available there.

The computer environment used in this experiment is the following: a machine with an AMD Athlon(tm) XP 2800+ at 2 GHz with 512 KB of cache, and 1 GB of RAM, with the Linux Debian operating system version 2.6.22. The algorithms were implemented in the Python language, and were run under version 2.5.2.

3.3 Results for the *easy* and *hard* instances

In Table 1 we report the results obtained by the greedy and the KK heuristics and those of DF branch-and-bound. As expected, KK is better than greedy, and DF is several orders of magnitude better than KK. For the *easy* instances, an optimal solution was found by DF for sizes up to $N = 70$; greater sizes likely have perfect partitions, but they could not be found in the allowed CPU time. For *hard* instances, optimal partitions were found up to instances with 30 elements.

Table 2 reports the results obtained by BF branch-and-bound for several values of the parameter α. If this parameter is small, BF terminates very quickly, but as expected the solution

		Greedy	KK	DF branch-and-bound		
Type	N	η	η	η	CPU	# nodes
easy	10	2.3219	1.5850	1.5850*	0.00	31
	20	3.8074	3.5850	1*	0.00	26
	30	10.691	1.5850	0*	0.00	419
	40	10.438	4.8580	0*	0.08	11209
	50	19.05	6.3750	0*	0.04	5526
	60	21.604	7.8455	1*	0.36	46188
	70	29.214	15.856	0*	120.1	14912636
	80	28.828	18.874	1.5850	3600	436000000
	90	39.593	23.956	2.3219	3600	438000000
	100	38.887	23.570	5.0444	3600	439300000
	200	88.664	74.029	48.140	3600	404900000
	300	142.75	114.26	96.262	3600	378600000
	400	189.66	163.44	141.43	3600	354500000
	500	240.98	211.31	191.10	3600	318600000
	600	288.78	258.86	241.03	3600	300100000
	700	337.79	307.70	286.07	3600	278200000
	800	390.32	355.13	339.05	3600	244900000
	900	437.15	403.69	383.30	3600	224100000
	1000	488.50	449.50	431.33	3600	214000000
hard	10	7.9425	3.8074	1*	0.00	39
	20	13.496	8.0334	4.1699*	0.06	10206
	30	21.962	16.393	3.8074*	27.21	3474759
	40	32.624	25.010	2.3219	3600	459600000
	50	43.953	31.380	15.732	3600	455300000
	60	50.982	40.329	20.758	3600	455600000
	70	62.215	50.729	32.536	3600	446200000
	80	73.694	60.534	37.779	3600	436600000
	90	82.726	70.787	49.325	3600	439600000
	100	91.725	73.369	56.115	3600	444300000
	200	191.58	171.81	151.91	3600	392800000
	300	288.80	265.77	247.07	3600	363700000
	400	391.38	366.18	343.36	3600	335200000
	500	490.52	461.60	438.79	3600	304800000
	600	587.09	557.09	539.13	3600	279500000
	700	687.86	659.50	637.70	3600	254900000
	800	788.01	751.27	731.98	3600	236100000
	900	887.70	853.36	834.95	3600	219700000
	1000	987.81	952.47	932.55	3600	200100000

Table 1: $\eta(E)$ for greedy and the KK heuristics, and DF branch-and-bound. CPU time for branch-and-bound (in seconds) and number of visited nodes (the greedy and KK heuristics run in less than 0.01 seconds of CPU time for all the instances). Instances solved to proven optimality are marked with an asterisk.

is not good. For larger values, the solutions are frequently better than those of DF, but not consistently. It seems that the optimal value of α is large for small instances, and tends to be smaller for larger instances. This probably occurs because for the larger instances too much time is spent on low depths of the tree, and the execution is terminated early, due to reaching the CPU limit. Possibly, fine tuning BF with a dynamic value of α would lead to better performance (we feared over-fitting, and did not attempt fine tuning). Table 3 reports the number of nodes visited when applying BF branch-and-bound.

		BF branch-and-bound									
		$\alpha = 10$		$\alpha = 100$		$\alpha = 1000$		$\alpha = 10000$		$\alpha = 100000$	
Type	N	η	CPU	η	CPU	η	CPU	η	CPU	η	CPU
easy	10	1.5850*	0.01	1.5850*	0.00	1.5850*	0.00	1.5850*	0.00	1.5850*	0.00
	20	1*	0.00	1*	0.00	1*	0.00	1*	0.00	1*	0.00
	30	0*	0.00	0*	0.00	0*	0.00	0*	0.00	0*	0.00
	40	0*	0.01	0*	0.02	0*	0.02	0*	0.02	0*	0.02
	50	1.5850	0.04	0*	0.07	0*	0.31	0*	0.38	0*	0.37
	60	4	0.05	2	0.50	1*	0.93	1*	0.55	1*	0.56
	70	6.4094	0.08	1.5850	0.74	0*	4.79	0*	45.79	0*	24.58
	80	9.5294	0.10	6.1085	0.92	4.6439	11.10	1.5850	131.9	0*	2947.
	90	15.401	0.12	9.0084	1.15	8.3083	14.09	4.0875	169.8	2.3219	3600
	100	19.388	0.14	14.329	1.42	11.018	17.57	7.9484	215.4	7.5925	3600
	200	60.017	0.54	57.685	5.66	53.842	74.89	49.536	987.0	49.322	3600
	300	105.27	1.20	101.23	12.99	98.644	170.91	95.347	2248.	89.487	3600
	400	151.14	2.16	149.29	23.93	146.03	312.73	136.22	3600	141.43	3600
	500	197.36	3.42	193.21	38.51	191.41	505.30	186.85	3600	190.80	3600
	600	247.77	5.04	242.89	57.37	240.70	748.57	239.09	3600	237.99	3600
	700	292.55	7.15	291.52	80.22	288.62	1042.19	283.05	3600	283.05	3600
	800	341.50	9.57	338.70	107.67	334.63	1379.31	334.63	3600	335.16	3600
	900	390.13	12.40	387.88	139.30	383.08	1768.60	384.28	3600	385.40	1536.[a]
	1000	436.50	15.72	434.44	174.47	430.19	2194.57	431.72	3600	431.35	723.2[a]
hard	10	1*	0.00	1*	0.00	1*	0.00	1*	0.00	1*	0.00
	20	4.1699	0.01	4.1699	0.04	4.1699	0.34	4.1699*	1.57	4.1699*	1.62
	30	11.036	0.02	5.6439	0.13	5.1699	1.34	3.8074	12.82	3.8074	240.9
	40	18.399	0.03	13.731	0.25	10.465	2.71	10.259	28.67	7.9366	655.2
	50	25.911	0.04	23.201	0.38	20.011	4.33	18.278	48.02	15.562	1186.
	60	31.402	0.06	28.252	0.52	26.773	6.21	24.771	71.13	21.191	1848.
	70	40.767	0.08	39.614	0.72	33.241	8.54	27.883	100.3	27.883	2680.
	80	49.642	0.10	48.029	0.96	42.904	11.41	42.489	139.4	39.152	3600
	90	62.467	0.12	56.371	1.20	52.050	14.88	52.050	183.2	47.679	3600
	100	66.840	0.14	66.083	1.44	61.353	18.55	59.336	228.9	56.405	3600
	200	160.12	0.54	156.92	5.73	151.98	75.68	149.55	931.9	147.40	3600
	300	254.18	1.21	251.97	13.34	249.20	177.61	246.30	2277.	246.07	3600
	400	351.53	2.24	348.71	24.73	345.96	325.70	336.98	3600	336.98	3600
	500	447.02	3.60	444.80	39.78	439.79	523.94	440.22	3600	440.08	3600
	600	546.98	5.33	539.91	59.18	539.10	775.33	529.36	3600	535.04	3600
	700	645.11	7.62	637.06	83.67	637.06	1081.70	636.66	3600	637.05	3600
	800	741.80	10.23	740.40	112.34	732.60	1431.81	733.63	3600	734.42	1449.[a]
	900	840.31	13.34	837.28	147.31	833.08	1879.33	833.69	3600	835.22	641.2[a]
	1000	938.04	17.08	932.40	185.67	932.40	2380.38	930.16	3600	934.47	156.2[a]

Table 2: Results for BF: $\eta(E)$ and CPU time (in seconds). Instances solved to proven optimality are marked with an asterisk. Runs labelled (a) were interrupted at the reported CPU time, due to insufficient memory.

The evolution of the best solution found with respect to the CPU time used by DF and several

Type	N	BB	BF branch-and-bound				
			$\alpha = 10$	$\alpha = 100$	$\alpha = 1000$	$\alpha = 10000$	$\alpha = 100000$
easy	10	31	35	35	35	35	35
	20	26	3	3	3	3	3
	30	419	6	6	6	6	6
	40	11209	106	360	388	388	388
	50	5526	871	1217	5109	6205	6205
	60	46188	1071	10055	11606	7267	7267
	70	14912636	1271	12055	53765	463485	207642
	80	436000000	1471	14055	134047	1272767	10225868
	90	438000000	1671	16055	154047	1472767	11062143
	100	439300000	1871	18055	174047	1672767	10462143
	200	404900000	3871	38055	374047	3672767	7462143
	300	378600000	5871	58055	574047	5672767	6062143
	400	354500000	7871	78055	774047	6292767	4862143
	500	318600000	9871	98055	974047	4832767	3862143
	600	300100000	11871	118055	1174047	4072767	3262143
	700	278200000	13871	138055	1374047	3452767	2862143
	800	244900000	15871	158055	1574047	2972767	2462143
	900	224100000	17871	178055	1774047	2572767	531071[*]
	1000	214000000	19871	198055	1974047	2232767	231071[*]
hard	10	39	3	3	3	3	3
	20	10206	271	2051	12965	34669	34669
	30	3474759	471	4055	34013	269405	1950223
	40	459600000	669	6055	54047	472667	4055187
	50	455300000	871	8055	74047	672767	6062139
	60	455600000	1071	10055	94047	872767	8062143
	70	446200000	1271	12055	114047	1072767	10062143
	80	436600000	1471	14055	134047	1272767	11062143
	90	439600000	1671	16055	154047	1472767	10062143
	100	444300000	1871	18055	174047	1672767	9662143
	200	392800000	3871	38055	374047	3672767	7862143
	300	363700000	5871	58055	574047	5672767	6062143
	400	335200000	7871	78055	774047	6012767	4862143
	500	304800000	9871	98055	974047	4712767	3862143
	600	279500000	11871	118055	1174047	3992767	3262143
	700	254900000	13871	138055	1374047	3332767	2862143
	800	236100000	15871	158055	1574047	2852767	531071[a]
	900	219700000	17871	178055	1774047	2472767	231071[a]
	1000	200100000	19869	198055	1974047	2112767	65535[a]

Table 3: Number of nodes visited on branch-and-bound. Runs labelled (a) were interrupted due to insufficient memory.

possibilities of BF seem to indicate a slight advantage of BF for small CPU times, as shown in Figures 4 and 5. Hard instances of the NPP are well known for being extremely difficult to solve. Our results corroborate this observation: for the hardest instances, there seems to be no big difference between any of the variants. The best solution is found by different strategies, depending on the instance and on the CPU time allowed to solution. (BF provides several different solutions, for different values of α; in some situations, this is an advantage.)

3.4 Results for the D instances.

Table 4 contains the results for tabu search and memetic algorithms reported in [2], and results obtained by branch-and-bound with the CPU time limited to 600 seconds. The latter are for different random instances, though they were generated with the same distribution. Even though the instances are different, the table shows a clear advantage of limited branch-and-bound over tabu search and the memetic algorithm.

Tabu search was reported to run for 10^9 iterations, and each the memetic algorithms for approximately the same amount of CPU time. For branch-and-bound the maximum (average) number of nodes was 77.4 million (39.4 million) for DF, and 8.0 million (4.5 million) for BF; thus, likely there was much less CPU used by branch-and-bound than by tabu search and the memetic algorithms.

The instances generated for the previous comparison were also used for giving more insight of the behavior of BF branch-and-bound, for different values of α, in Table 5. The results confirm a general tendency for obtaining better solutions when the parameter α in BF search increases. There are some exceptions; the most intriguing one is for B=12, N=95, where for $\alpha = 10000$ the search could find a prove optimum for all instances (by reaching the lower bound of 0 or 1), though a broader search with $\alpha = 100000$ could not find the optimum for one of the instances. Notice that there is a difference between the results for branch-and-bound reported in Tables 4 and 5, as $\eta(E)$ for the average E is different of the average $\eta(E)$; the latter is more meaningful, but we do not have it available for the results reported in the literature.

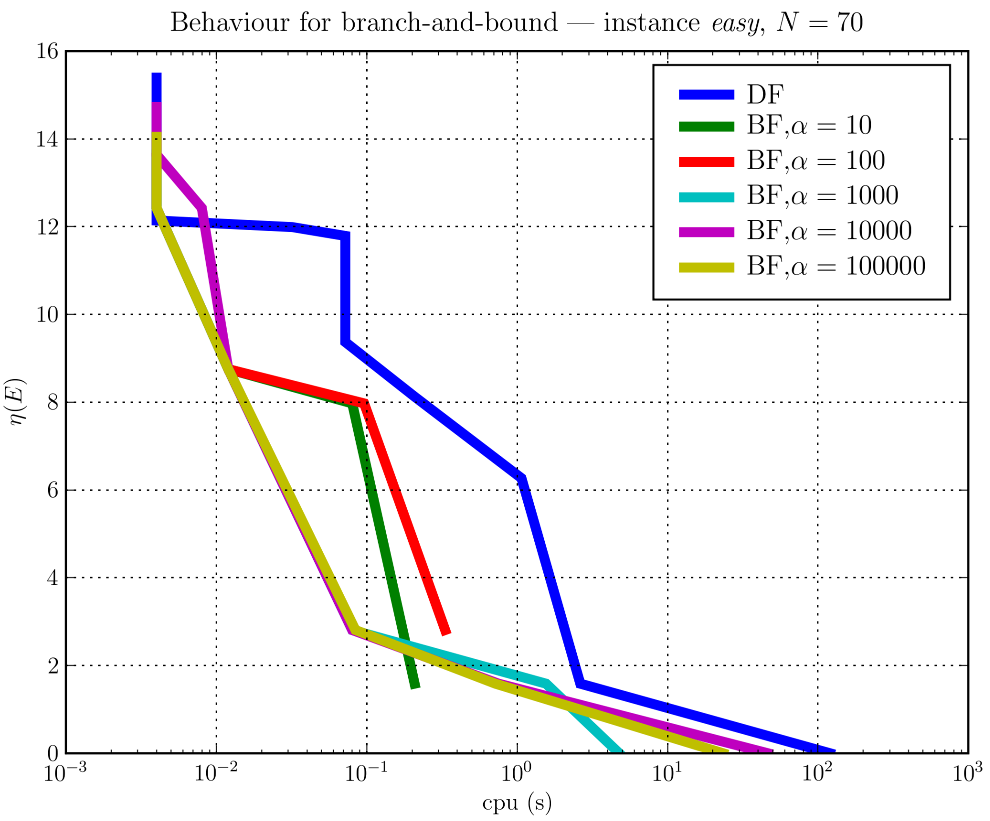
There are also results reported in [1] for a different formulation of tabu search, based on re-casting the NPP as an unconstrained quadratic binary program. However, the results presented there are for instances with 25 and 75 elements drawn randomly from the interval (50,100); we could find no instance generated this way for which branch-and-bound would not find an optimal solution in less than a second (all these instances are easy, and thus are likely to have many perfect solutions).

4 Conclusion

We propose the use of limited branch-and-bound as heuristics for the solution of the number partitioning problem. We present two versions: a depth-first version, as proposed in [11], which is interrupted when the CPU time reaches a specified limit, and a breadth first version, borrowing ideas from beam search [15], with a limit in the CPU used as well as in the number of nodes explored on each depth of the branch-and-bound tree.

For breadth-first branch-and-bound, we implemented a diving method which consists of taking the left-side, differencing branch until reaching a leaf. Left-side diving allows the search to quickly obtain an upper bound for every node; furthermore, as its computation is relatively inexpensive, it can be done systematically. This diving strategy corresponds to applying the Karmarkar and Karp heuristics as a guide until reaching the bottom of the branch-and-bound tree. The concept is general, and it is not sufficiently explored in the literature; we believe that the same idea can be successfully applied for improving branch-and-bound on other problems for which good heuristics exist.

The results for interrupted execution of the two branch-and-bound algorithms are very good, as compared to other approximative methods found in the literature. The idea of combining a good polynomial-time approximation algorithm and making it complete, also introduced in [11], is very promising and seems to be still seldom used. The parameterized breadth-first branch-and-bound has the advantage of allowing a certain diversification of the solutions obtained, by changing the parameter. This frequently leads to better solutions than those obtained by depth-first search.



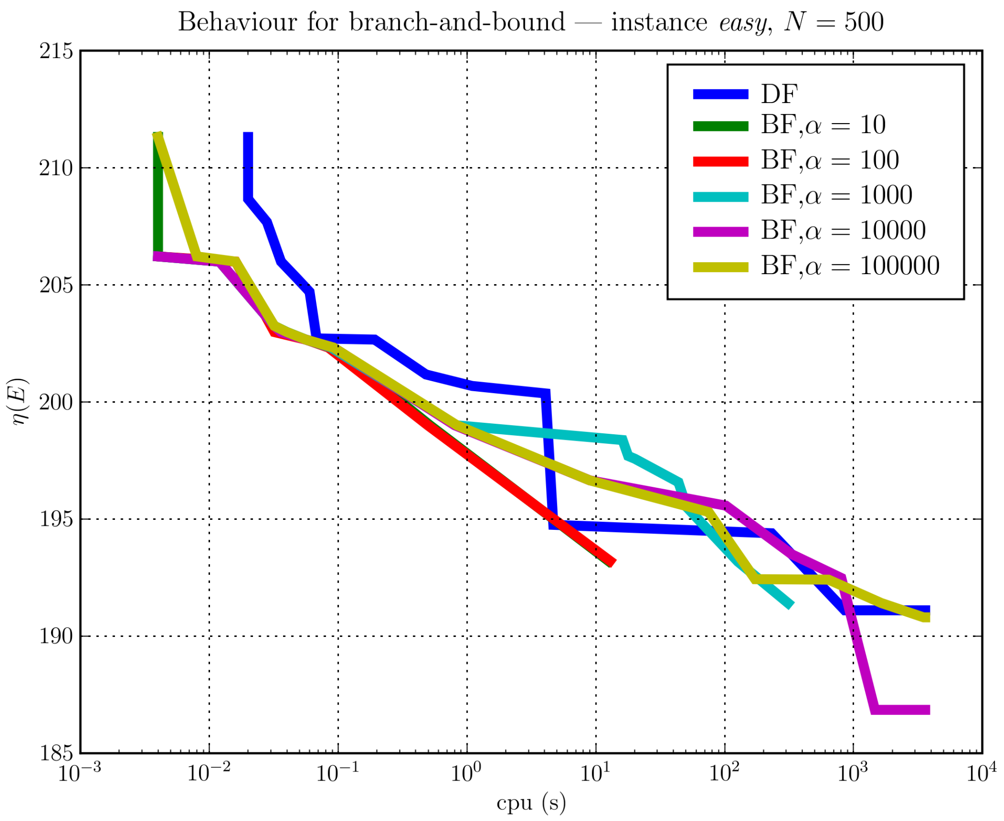
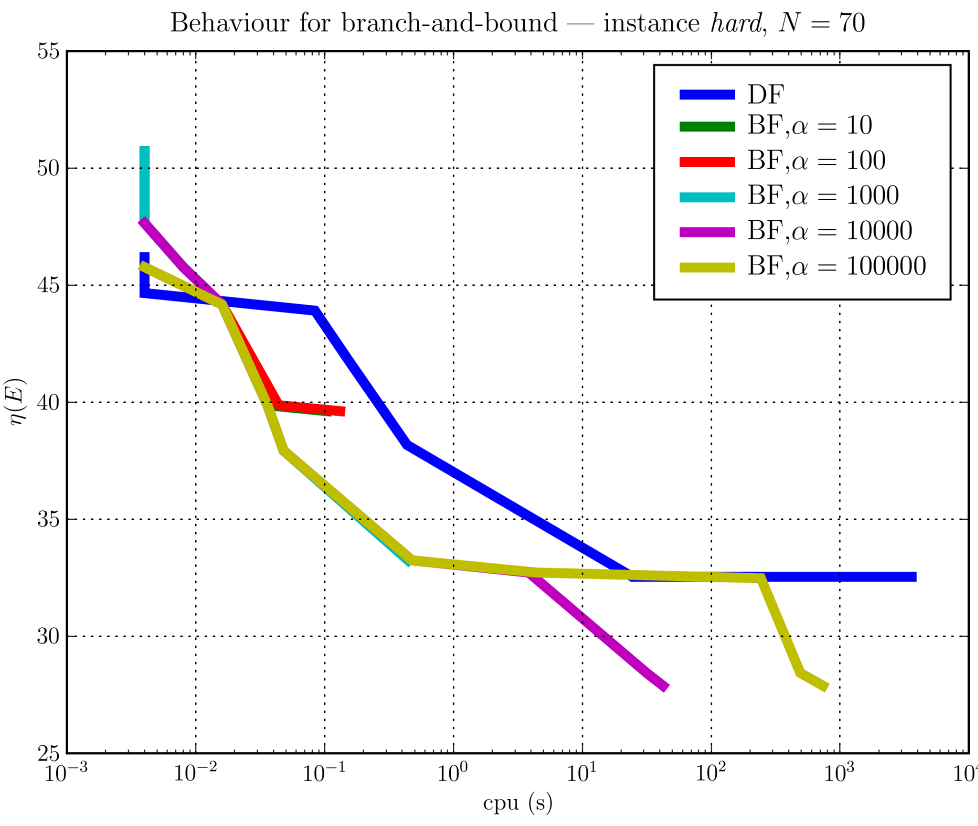


Figure 4: Evolution of the best found solution with respect to the CPU time for several variants of branch-and-bound — *easy* instances.



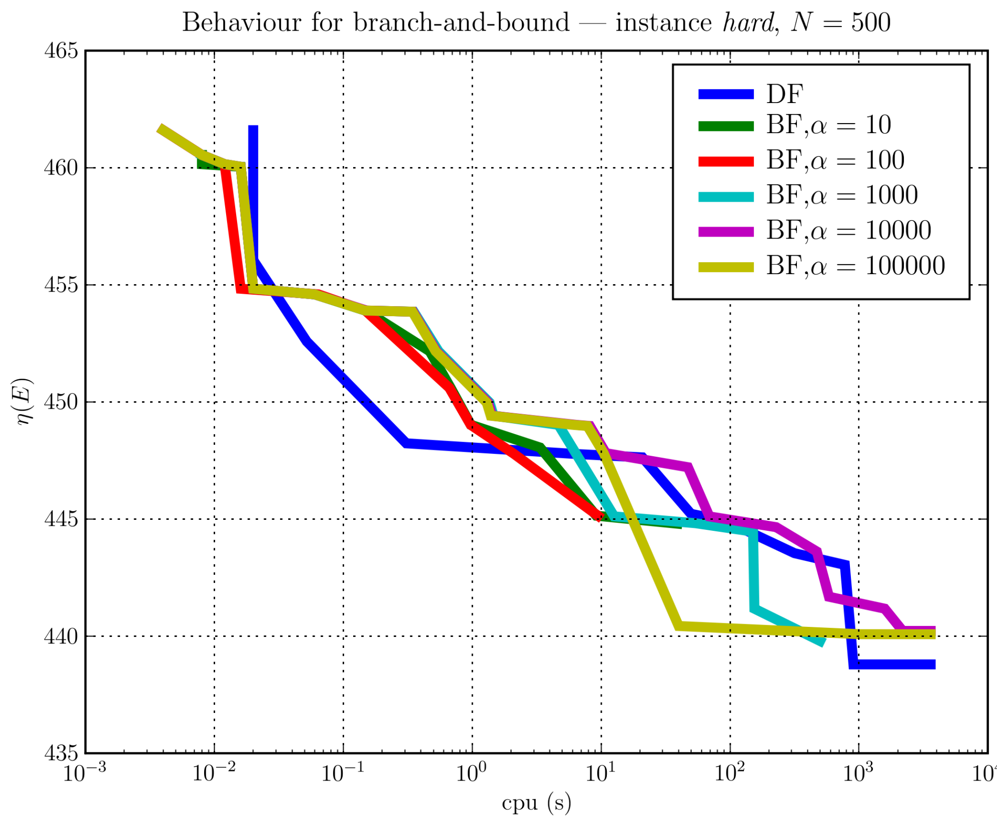


Figure 5: Evolution of the best found solution with respect to the CPU time for several variants of branch-and-bound — *hard* instances.

	N	Tabu	Memetic	DF		BF	
D=10	15	20.768	20.768	20.157	(10)	20.157	(10)
	25	11.627	11.627	10.474	(10)	10.474	(0)
	35	3.585	3.322	2.5110	(5)	3.6439	(1)
	45	3.585	2.585	1.2016	(7)	0.58496	(10)
	55	2.807	1.585	0.76553	(9)	0.58496	(10)
	65	3.700	2.322	0.58496	(10)	0.58496	(10)
	75	2.807	1.000	0.58496	(10)	0.58496	(10)
	85	2.807	1.585	0.48543	(10)	0.48543	(10)
	95	2.807	2.000	0.84800	(10)	0.84800	(10)
	105	2.322	1.585	0.76553	(10)	0.76553	(10)
D=12	15	27.508	27.508	27.795	(10)	27.795	(10)
	25	17.896	17.896	17.627	(10)	17.627	(0)
	35	9.856	9.775	8.5137	(1)	9.9427	(0)
	45	9.592	8.155	6.3291	(0)	5.6088	(0)
	55	10.006	8.214	5.2327	(1)	3.2928	(2)
	65	9.690	8.077	4.6949	(1)	2.2928	(3)
	75	8.904	8.622	2.9635	(4)	1.8480	(4)
	85	9.236	8.443	1.9635	(4)	0.76553	(8)
	95	8.238	7.658	1.0704	(8)	0.92600	(9)
	105	8.362	8.087	0.92600	(8)	0.58496	(10)
D=14	15	34.075	34.075	34.299	(10)	34.299	(10)
	25	24.548	24.396	25.201	(10)	25.201	(0)
	35	17.674	15.837	14.513	(2)	16.706	(0)
	45	15.799	14.914	13.301	(0)	12.600	(0)
	55	15.362	15.164	10.359	(0)	10.592	(0)
	65	15.614	15.472	10.122	(0)	8.4179	(0)
	75	16.250	15.066	8.4128	(0)	6.6381	(0)
	85	15.693	14.645	7.9606	(0)	6.3092	(0)
	95	15.080	14.405	6.7184	(0)	4.9495	(0)
	105	14.698	14.268	6.4611	(0)	5.3327	(0)

Table 4: Results for tabu search and a memetic algorithm as reported in [2]: $\eta(E)$, where E is the average result obtained on 10 random instances; (for the memetic algorithms, in each entry (N, D) we report the best of the solutions obtained in four proposed alternatives). The same measure for DF and BF branch-and-bound, with the CPU limited to 600 seconds, on 10 similar (but not identical) random instances. BF was run with $\alpha = 100000$. The number in parenthesis indicates the number of instances for which the solution was proven optimal.

		DF	BF		
	N		$\alpha = 1000$	$\alpha = 10000$	$\alpha = 100000$
D=10	15	19.363 (10)	19.363 (10)	19.363 (10)	19.363 (10)
	25	9.8359 (10)	10.541 (0)	9.8359 (0)	9.8359 (0)
	35	2.1099 (5)	6.6540 (0)	5.2543 (0)	3.3242 (1)
	45	0.91699 (7)	3.8599 (1)	1.9901 (3)	0.50000 (10)
	55	0.65850 (9)	2.6402 (0)	0.75850 (8)	0.50000 (10)
	65	0.50000 (10)	1.3077 (5)	0.50000 (10)	0.50000 (10)
	75	0.50000 (10)	0.50000 (10)	0.50000 (10)	0.50000 (10)
	85	0.40000 (10)	0.40000 (10)	0.40000 (10)	0.40000 (10)
	95	0.80000 (10)	0.80000 (10)	0.80000 (10)	0.80000 (10)
	105	0.70000 (10)	0.70000 (10)	0.70000 (10)	0.70000 (10)
D=12	15	25.896 (10)	25.896 (10)	25.896 (10)	25.896 (10)
	25	17.313 (10)	18.117 (0)	17.562 (0)	17.313 (0)
	35	7.4071 (1)	12.651 (0)	11.465 (0)	9.5176 (0)
	45	5.4617 (0)	9.6355 (0)	7.7449 (0)	5.4163 (0)
	55	4.7706 (1)	8.212 (0)	5.7195 (0)	2.8388 (2)
	65	4.0577 (1)	5.9499 (0)	3.9299 (1)	1.9747 (3)
	75	2.2791 (4)	3.8214 (1)	2.1206 (4)	1.6884 (4)
	85	1.5858 (4)	3.7136 (1)	0.77549 (7)	0.61699 (8)
	95	0.9585 (8)	1.6615 (4)	0.70000 (10)	0.85850 (9)
	105	0.7585 (8)	1.1907 (5)	0.50000 (10)	0.50000 (10)
D=14	15	34.011 (10)	34.011 (10)	34.011 (10)	34.011 (10)
	25	24.807 (10)	25.235 (0)	24.932 (0)	24.807 (0)
	35	13.347 (2)	20.382 (0)	18.131 (0)	15.932 (0)
	45	12.275 (0)	16.114 (0)	13.828 (0)	11.703 (0)
	55	9.2294 (0)	14.32 (0)	10.37 (0)	10.158 (0)
	65	8.7968 (0)	11.703 (0)	9.3579 (0)	7.5600 (0)
	75	7.7004 (0)	11.85 (0)	8.4935 (0)	5.5553 (0)
	85	6.8629 (0)	9.2131 (0)	6.4178 (0)	5.5963 (0)
	95	6.0543 (0)	8.4292 (0)	5.734 (0)	4.5529 (0)
	105	5.5494 (0)	6.6276 (0)	4.7675 (0)	4.9086 (0)

Table 5: Average $\eta(E)$ for DF and BF branch-and-bound, with the CPU limited to 600 seconds, for the 10 random D instances. The number in parenthesis indicates the number of instances (out of 10) for which the solution was proven optimal.

For sorting the nodes in each level of the BF search tree we used the number of times that a summing-branch was followed, from the root to the current node. There are other possibilities; we did try to use the upper bound computed with the KK heuristics for sorting, but the results did not improve. Finding a better measure for sorting the nodes at the same level is an interesting subject open to further research.

The exact solution of the large instances presented in this paper, of both the *easy* and *hard* series, is challenging; we provide the instances' data, and an upper bound that can be used as a term for comparison in future works.

References

[1] Bahram Alidaee, Fred Glover, Gary A. Kochenberger, and César Rego. A new modeling and solution approach for the number partitioning problem. *JAMDS*, 9(2):113–121, 2005.

[2] Regina Berretta, Carlos Cotta, and Pablo Moscato. Enhancing the performance of memetic algorithms by using a matching-based recombination algorithm. In Jorge P. Sousa and Mauricio G. C. Resende, editors, *METAHEURISTICS: Computer Decision-Making*, pages 65–90, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[3] Production Planning by Mixed Integer Programming. *Yves Pochet and Laurence A. Wolsey.* Springer, 2006.

[4] E. G. Coffman Jr. and G. S. Lueker. *Probabilistic Analysis of Packing and Partitioning Algorithms.* John Wiley & Sons, 1991.

[5] Milos Ercegovac, Darko Kirovski, and Miodrag Potkonjak. Low-power behavioral synthesis optimization using multiple precision arithmetic. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 568–573, New York, NY, USA, 1999. ACM.

[6] F. F. Ferreira and J. F. Fontanari. Probabilistic analysis of the number partitioning problem. *Journal of Physics A*, 31:3417–3428, 1998.

[7] M. R. Garey and D. S. Johnson. *Computers and intractability.* W. H. Freeman, New York, 1979.

[8] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[9] N. Karmarkar and R. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, University of California - Berkeley, Computer Science Division, 1982.

[10] N. Karmarkar, R. M. Karp, G. S. Lueker, and A. M. Odlyzko. Probabilistic analysis of optimum partitioning. *Journal of Applied Probability*, 23:626–645, 1986.

[11] Richard E. Korf. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181–203, 1998.

[12] R.C. Merkle and M.E. Hellman. Hiding information and signatures in trapdoor knapsack. *IEEE Transactions on Information Theory*, 24(5):525–530, 1978.

[13] Stephan Mertens. The easiest hard problem: Number partitioning. In A.G. Percus, G. Istrate, and C. Moore, editors, *Computational Complexity and Statistical Physics*, pages 125–139, New York, 2006. Oxford University Press.

[14] João P. Pedroso. Branch-and-bound for number partitioning: an implementation in the Python language. Internet repository, version 1.0, 2008. http://www.dcc.fc.up.pt/~jpp/partition and http://modern-heuristics.com.

[15] M. Pinedo. *Scheduling: theory, algorithms, and systems.* Prentice-Hall, Englewood Cliffs, 1995.

[16] Benjamin Yakir. The differencing algorithm LDM for partitioning: a proof of a conjecture of Karmarkar and Karp. *Math. Oper. Res.*, 21(1):85–99, 1996.