DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD

# Easily searched encodings for number partitioning

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

| | |
|---|---|
| Citation | Wheeler Ruml, J. Thomas Ngo, Joe Marks, and Stuart M. Shieber. Easily searched encodings for number partitioning. Journal of Optimization Theory and Applications, 89(2):251-291, July 1996. The original publication is available at www.springerlink.com. |
| Published Version | doi:10.1007/BF02192530 |
| Accessed | February 17, 2015 12:59:17 PM EST |
| Citable Link | http://nrs.harvard.edu/urn-3:HUL.InstRepos:2031717 |
| Terms of Use | This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA |

*(Article begins on next page)*

# Easily Searched Encodings for Number Partitioning

Wheeler Ruml
ruml@das.harvard.edu
*Harvard University, Cambridge, Massachusetts*

J. Thomas Ngo
ngo@interval.com
*Interval Research Corporation, Palo Alto, California*

Joe Marks
marks@merl.com
*Mitsubishi Electric Research Laboratories, Inc., Cambridge, Massachusetts*

Stuart Shieber
shieber@das.harvard.edu
*Harvard University, Cambridge, Massachusetts*

*Journal of Optimization Theory and Applications,* 1995, to appear.
Final version.

### Abstract

Can stochastic search algorithms outperform existing deterministic heuristics for the NP-hard problem NUMBER PARTITIONING if given a sufficient, but practically realizable amount of time? In a thorough empirical investigation using a straightforward implementation of one such algorithm, simulated annealing, Johnson et al. (1991) concluded tentatively that the answer is "no."

In this paper we show that the answer can be "yes" if attention is devoted to the issue of problem representation (encoding). We present results from empirical tests of several encodings of NUMBER PARTITIONING with problem instances consisting of multiple-precision integers drawn from a uniform probability distribution. With these instances and with an appropriate choice of representation, stochastic and deterministic searches can—routinely and in a practical amount of time—find solutions several orders of magnitude better than those constructed by the best heuristic known (Karmarkar and Karp, 1982), which does not employ searching.

The choice of encoding is found to be more important than the choice of search technique in determining search efficacy. Three alternative explanations for the relative performance of the encodings are tested experimentally. The best encodings tested are found to contain a high proportion of good solutions; moreover, in those encodings, the solutions are organized into a single "bumpy funnel" centered at a known position in the search space. This is likely to be the only relevant structure in the search space because a blind search performs as well as any other search technique tested when the search space is restricted to the funnel tip.

We also show how analogous representations might be designed in a principled manner for other difficult combinatorial optimization problems by applying the principles of parameterized arbitration, parameterized constraint, and parameterized greediness.

**Keywords:** number partitioning, NP-complete, representation, encoding, empirical comparison, stochastic optimization, parameterized arbitration, parameterized constraint, parameterized greediness.

# 1 Introduction

Number Partitioning is a very simple NP-hard problem: given a sequence $A = (a_1, a_2, \ldots, a_n)$ of positive rational numbers, the object is to find a corresponding sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in \{-1, +1\}$, such that the *residue*

$$u \equiv \left| \sum_{i=1}^{n} s_i a_i \right|$$

is minimized. A sample instance, to which we refer repeatedly in examples, is given in Table 1. Intuitively, the task is to divide the given sequence into two subsets of roughly equal sum.[1] Number Partitioning is the optimization problem implied by the decision problem Partition, in which it is asked whether a perfect partitioning ($u = 0$) exists for a given sequence $A$. Partition was shown to be NP-complete by Karp (1972).

Despite the apparent simplicity of Number Partitioning, good heuristics have been difficult to devise. The best heuristic known, due to Karmarkar and Karp (1982), runs in $O(n \log n)$ time, and given a sequence of numbers distributed uniformly between 0 and 1, it returns solutions with expected values of $u = n^{-\alpha \log n}$, where $\alpha$ is a positive constant. Although this is much better than previously reported heuristic techniques, it falls far short of known upper bounds on the expected optimum residue (§7). Another significant drawback is that the Karmarkar-Karp (KK) heuristic is completely deterministic: it always constructs the same solution for a given problem instance. Thus the KK algorithm does not permit a trade-off between computational effort and the value of the solution found.

The term *stochastic optimization* (SO) refers to a family of search techniques—simulated annealing (Kirkpatrick et al., 1983; Černy, 1985) and genetic algorithms (Holland, 1975; Goldberg, 1988; Davis, 1991) being two of the most popular types—that might appear to be worthy candidates for improving on the KK heuristic. SO techniques generally do permit a trade-off between computational effort and the value of the solution found. Therefore, while SO is often valued principally as a "method of first resort" when little is known about an optimization problem, it is worth asking whether SO could be the technique of choice even for a problem that has undergone extensive analysis. However, previous research results have suggested that SO is poorly suited for Number Partitioning relative to the KK heuristic. A thorough empirical investigation of simulated annealing for Number Partitioning by Johnson et al. (1991) led them to report:

> For number partitioning, simulated annealing is not competitive with the differencing algorithm of N. Karmarkar and R. M. Karp, except on relatively small instances.

Nevertheless, Johnson et al. caution:

> [Our results should not be taken to imply that] there is *no* way of successfully adapting simulated annealing to this problem, but at present we can think of no better alternatives than the ones we consider.

[1] Indeed, the task is conventionally cast in terms of dividing a given set of numbers into two partitions such that the difference $u$ between the partitions' sums is as close as possible to zero. It is for clarity later in the paper that we choose to describe the problem in terms of finding a sign sequence $S$ that minimizes the residue $u$.

In this paper, we present some better alternatives. We show that by representing the problem appropriately—choosing a suitable encoding for candidate solutions, and operators for generating new candidate solutions from old ones—a variety of SO techniques can find solutions better than those computed by the KK heuristic. For problem instances with 100 uniformly distributed elements, our techniques match the values of $u$ attained by KK in the same amount of time, begin improving upon the KK solution immediately thereafter, and after a practical amount of time better the KK solution by several orders of magnitude.

We begin by establishing some basic terminology (§2). We summarize previous work: the KK heuristic and previous attempts at using SO for this problem (§3). We describe four encodings for NUMBER PARTITIONING and four "search engines" (§4), and demonstrate that the choice of encoding is far more important than the choice of search engine in determining SO performance (§5). We propose three alternative explanations for the relative performance of the encodings and distinguish among these explanations experimentally (§6). We conclude by discussing how the principles we have used to design effective encodings for NUMBER PARTITIONING might be applied to other optimization problems (§7).

| 6 | 5 | 4 | 4 | 3 | $u$ | Remark |
|---|---|---|---|---|---|---|
| + | + | − | − | − | 0 | Optimal solution |
| + | − | + | − | − | 2 | KK solution |

Table 1: Sample instance of NUMBER PARTITIONING with two candidate solutions. In this sample instance, $A = (6, 5, 4, 4, 3)$.

## 2 Basic terminology

*Stochastic optimization* refers to a family of optimization techniques in which the solution space is searched by generating candidate solutions with the aid of a pseudorandom number generator. As the run proceeds, the probability distribution by which new candidate solutions are generated may change, based on results of trials earlier in the run. (In the simplest and most common case, a new candidate solution is obtained by perturbing another solution from a recent trial.)

Each SO algorithm that we investigate may be divided into two layers. A distinction between these two layers is essential to our research strategy. The bottom layer, the *representation* or *encoding* of the problem, encapsulates details about what independent variables are used to encode a candidate solution, and how the values of those variables might be manipulated during a search. In the context of this paper, an encoding is specified by a list of these independent variables and their possible values (which define the encoding space $\Xi$) and a specification for each of the following operations:

- *Evaluate*—Compute a real-valued cost $u(\xi)$ given a candidate solution $\xi \in \Xi$. The algorithm for decoding $\xi$ may, in general, incorporate heuristics specific to the representation.
- *Randomize*—Draw a candidate solution $\xi \in \Xi$ from some a priori probability distribution.
- *Perturb*—Change the values of the independent variables randomly by a small amount, i.e., given one candidate solution $\xi \in \Xi$, generate another (possibly identical) solution $\xi' \in \Xi$.

The task of the top layer, the *search engine*, is to find $\xi \in \Xi$ that minimizes $u(\xi)$ using only the operations *Evaluate*, *Randomize*, and *Perturb*. A simple example of a search engine is hill climbing (§4.6), in which a randomly generated candidate solution $\xi$ is repeatedly perturbed, but perturbations that increase $u(\xi)$ are rejected. In this paper, attention is restricted to four search engines that employ only the three operations listed above. A fourth operation called *Crossover*, in which two solutions are somehow combined to make a third, can be defined. If an encoding is extended to include such an operation, then several additional search engines—variants of the genetic algorithm (Holland, 1975)—become applicable. We have conducted a more extensive empirical study (Ruml, 1993) that includes genetic algorithms. Because the combinations of genetic algorithm and encoding tested did not present any particular advantage, for brevity we do not present those results here.

The importance of the distinction between the search-engine and encoding layers lies in the principles by which various search engines operate. Although these principles vary greatly in detail depending on the search engine in question, the principles are all generic. That is, they involve minimal assumptions about the distribution of candidate solutions within the search space—assumptions that rely little on information specific to the problem at hand. For example, as Johnson et al. (1991) note, simulated annealing relies for its success on the existence of a "neighborhood structure" in the search space. It is assumed, in essence, that groups of good solutions tend to lie close together, i.e., tend to be mutually accessible via perturbations. If the solution space of a problem lacks neighborhood structure under a given encoding, then simulated annealing may not fare better than a blind search given that encoding.

Thus, the success of the search-engine layer in locating solutions of low cost relies critically on the nature of the solution space presented to it by the representation layer. If the representation produces a candidate-solution space whose structure is incompatible with the assumptions of the search engine, the algorithm as a whole is likely to fail to find low-cost solutions efficiently. In §4 we show by example how an encoding can be designed in a principled manner by choosing its independent variables with respect to a given existing greedy heuristic. In §6 we suggest how the operators *Randomize* and *Perturb* can also be designed in a principled manner.

# 3 Previous Work

## 3.1 Pseudopolynomial algorithm

NUMBER PARTITIONING can be solved in pseudopolynomial time. If $A$ is a sequence of positive integers whose sum is $B$, then the optimal residue $u$ can be determined in time polynomial in $nB$—"pseudopolynomial" time, because $nB$ is not bounded by any polynomial in $n \log B$, a measure of the space required to store $A$ (Garey and Johnson, 1979).

(The pseudopolynomial algorithm is based on the observation that for any subset $A' \subseteq A$, the sum $\sum_{a \in A'} a$ must be an integer between 1 and $B$. By dynamic programming it is asked, in turn, whether there exists $A' \subseteq A$ such that $\sum_{a \in A'} a = 1, 2$, and so on up to $\lfloor B/2 \rfloor$. A value $u$ is an attainable residue if and only if $\frac{1}{2}(B - u)$ is a possible partial sum; therefore, the partial sum closest to $\frac{1}{2}B$ gives the minimal residue $u$.)

This pseudopolynomial algorithm, which requires time and space at least linear in $B$, is of little practical value when $B$ is large. In the runs presented here, we employed effective values of $B$ that exceed $2^n$, the total number of candidate solutions.

## 3.2 Karmarkar-Karp algorithm

Many heuristics that attempt to minimize $u$ even when $B$ is large have been described. Of these, the most successful has been the Karmarkar-Karp (KK) algorithm (Karmarkar and Karp, 1982).

The KK procedure may be understood by means of the following observation. Consider a pair of elements, $a_i$ and $a_j$, in the given sequence $A$. If it is believed, for whatever reason, that all candidate solutions in which $a_i$ and $a_j$ lie in the same partition (i.e., $s_i s_j = 1$) can be eliminated from consideration, then the larger of $a_i$ and $a_j$ can be replaced by their difference $|a_i - a_j|$, and the smaller by zero, without changing the set of attainable values of the residue $u$. This operation, called *differencing*, can be applied recursively to the sequence of remaining nonzero numbers. With each differencing operation the number of nonzero elements in the sequence is reduced by at least one, and when the sequence contains only one nonzero element, that element is the residue. (If no nonzero element remains, then a perfect partitioning exists.) A sequence of signs $s_i$ that produces this residue can be constructed in a straightforward manner from the sequence of differencing choices by two-coloring the graph $(A, E)$, where $E$ is the set of pairs $(a_i, a_j)$ subjected to differencing.

Unfortunately, no known algorithm based on repeated differencing is guaranteed to produce an optimal solution in polynomial time. In particular, at each step of the process, it is not known how to select efficiently a pair of elements $a_i$ and $a_j$ for differencing such that the optimal solution is guaranteed not to be eliminated from consideration. (A polynomial-time algorithm for making this choice with such a guarantee does not exist unless P = NP.) The KK algorithm is a heuristic special case of the repeated-differencing procedure in which the two elements $a_i$ and $a_j$ selected for differencing at any given step are the two largest numbers in the sequence (Figure 1). The solution returned by this procedure is not necessarily optimal (Table 1). In pseudocode, the KK procedure is:

**for** $i = 1$ **to** $n - 1$
  $j \leftarrow$ the index of the largest element of $A$
  $k \leftarrow$ the index of the second largest element of $A$
  $a_j \leftarrow a_j - a_k$
  $a_k \leftarrow 0$
**end for**

When the procedure ends, at least $n-1$ of the elements of $A$ will be equal to zero, and the remaining element will be the value of $u$.

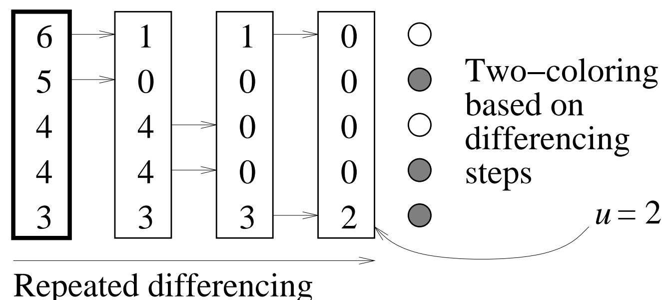


Figure 1: Illustration of the KK heuristic.

Since the publication of the KK algorithm (Karmarkar and Karp, 1982), no superior deterministic heuristic for NUMBER PARTITIONING has been developed. Indeed, Karmarkar et al. (1986) later declared:

> [The expected residue $u$ produced by the KK algorithm] was a great improvement over other results, [but] is still much greater than the optimum difference which is shown in this paper to be likely. It would be very interesting, though possibly very difficult, to improve upon that algorithm.

### 3.3 Previous investigations of SO for NUMBER PARTITIONING

Might stochastic optimization techniques produce solutions closer to optimal than the KK solution, albeit given a larger (but practically realizable) amount of time? This possibility was investigated by Johnson et al. (1991), who empirically compared stochastic algorithms for NUMBER PARTITIONING with the KK heuristic. Johnson et al. tested two search engines with a single encoding—in particular, they tested the combinations that we call SA/direct and HC/direct. (Table 2 explains these and other abbreviations; the encodings and search engines themselves are described in §4.) Simulated annealing was found to be inferior to the KK algorithm, especially with larger problem instances. Indeed, NUMBER PARTITIONING had been chosen as a hard test case for simulated annealing because, at least in what we call the direct representation, "the natural 'neighborhood structures' for it, those in which neighboring solutions differ as to the location of only one or two elements, have exceedingly 'mountainous' terrains, in which neighboring solutions differ widely in cost" (Johnson et al., 1991).

Jones and Beltramo (1991) considered the use of a genetic algorithm for a generalization of the NUMBER PARTITIONING problem that they called the "equal piles problem." In this problem, the object is to divide a given set of numbers into $k$ partitions of equal sum. (NUMBER PARTITIONING is the special case of the equal-piles problem in which $k = 2$.) To test the effectiveness of various encodings with the GA, they compared the combinations that we call GA/jb-greedy, GA/jb-sep, and GA/jb-num, each using several possible types of *Crossover*. Because they used a small problem instance ($n = 34$, $k = 10$) and did not perform comparisons with the KK algorithm, their results do not help answer the question posed above. The results of Ruml (1993) include tests of the best Jones-Beltramo algorithms with larger instances.

# 4 Encodings and Search Engines

### 4.1 Overview

In contrast with previous investigations of SO techniques for NUMBER PARTITIONING, we systematically employed a suite of possible encodings (many of them novel) with a suite of search engines. Our initial purpose was to test, empirically, the proposition that the success of a search engine in finding low-cost solutions for a given instance of NUMBER PARTITIONING depends heavily on the nature of the representation used.

With the exception of the direct encoding (§4.2), every encoding we used was designed with the benefit of insights derived from existing greedy deterministic heuristics such as the KK algorithm. In the remainder of the paper, we will refer to the solution constructed by the given greedy heuristic as

| Search-engine layer | |
|---|---|
| Abbreviation | Explanation |
| **RGT** (§4.6) | Random generate and test |
| **HC** (§4.6) | Hill climbing |
| **PHC** (§4.6) | Parallel hill climbing |
| **SA** (§4.6) | Simulated annealing |
| GA | Genetic algorithm |

| Representation layer | |
|---|---|
| Abbreviation | Explanation |
| **direct** (§4.2) | Direct: The encoding used by Johnson et al. (1991) |
| **jb-greedy** (§4.3) | Greedy decoder: The best encoding used by Jones and Beltramo (1991) |
| jb-sep | Group separators: An encoding used by Jones and Beltramo (1991) |
| jb-num | Group numbers: An encoding used by Jones and Beltramo (1991) |
| split | Splitting decoder |
| split-alt | Alternating splitting decoder |
| split-num | Number-based splitting decoder |
| split-grd | Greedy splitting decoder |
| **prepart** (§4.4) | Prepartitioning |
| index-dbl | Double index rules |
| index-wgt | Weighted double index rules |
| **index** (§4.5) | Single index rules (modified) |

Table 2: Abbreviations for the various search engines and encodings employed in the empirical comparisons of Ruml (1993). Abbreviations for a search engine and an encoding may be combined with an intervening slash (/); for example, "SA/direct" denotes the use of simulated annealing with the direct representation. Boldface abbreviations correspond to the search engine and encodings that are described in detail in this paper.

the *greedy solution*. Our strategy for designing an effective encoding is summarized in the following three steps.

1. Select an existing greedy heuristic for the optimization problem in question.

2. Modify the heuristic so that its operation is not fully determined.

3. Choose, for every underdetermined decision point in the heuristic, a parameter whose value will determine the decision. The set of such parameters will be the independent variables of the encoding.

In the case of the encodings presented here and by Ruml (1993), Step 2 was accomplished by one of the following three principles:

**parameterized arbitration** This design principle can be used only when the given greedy heuristic already requires making arbitrary decisions. For example, Jones and Beltramo (1991) describe a greedy heuristic in which the two partitions are initially empty and the elements $a_i$ from the given instance are added, one at a time, to the smaller of the growing partitions. The

order in which the elements $a_i$ are added is arbitrary. The encoding abbreviated "jb-greedy" (§4.3) was designed from this heuristic by parameterized arbitration: an instance of jb-greedy specifies the order in which elements are to be added to the partitions.

**parameterized constraint** This design principle can be used if the given greedy heuristic is able to respect externally imposed constraints. This condition clearly holds with any heuristic for NUMBER PARTITIONING, since groups of elements can be constrained to lie in the same partition in a preprocessing step; the heuristic itself need not be modified. The prepartitioning encoding (§4.4) is an example of parameterized constraint: an instance of this encoding specifies subsets of $A$ whose elements are to be constrained to lie in the same partition.

Three attractive features are likely to be common among parameterized-constraint encodings. First, the encoding will be complete—all solutions that can be specified in the direct representation will be encodable—if it is legal to specify constraints that fully define a direct solution. Second, the empty set of constraints corresponds to the solution constructed by the unparameterized heuristic. Thus, it is trivial to guarantee that an SO run based on a parameterized-constraint encoding will do at least as well as the original heuristic. Third, the cost of a solution can be anticipated to be positively correlated with the number of constraints in its parameterized-constraint encoding, thus producing the "bumpy-funnel" effect described in §6.3.

**parameterized greediness** A greedy deterministic heuristic generally consists of a sequence of greedy low-level decisions. An encoding can be designed by the principle of parameterized greediness[2] if the greediness of these low-level decisions can be "tuned." Consider, the KK algorithm, in which the two elements $a_i$ and $a_j$ selected for differencing at any given step are the two *largest numbers* in the sequence (§3.2). In the index-rules representation (§4.5), the degree of greediness employed in the selection of $a_j$ is varied: a given solution instance might specify, for example, that $a_i$ and $a_j$ are to be the largest and third-largest elements in the sequence.

Unlike parameterized-constraint encodings, an encoding designed by parameterized greediness cannot generally be expected to be complete. However, the other two attractive features of parameterized-constraint encodings are present: an SO run based on such an encoding can easily be guaranteed to do at least as well as the original heuristic, and the cost of a solution can be expected to be correlated with the "greediness" of the candidate solution (§6.3).

The categories listed above neither are mutually exclusive, nor constitute an exhaustive list of the ways in which principles underlying existing deterministic heuristics can be incorporated into a representation that is to be used with SO. We return to this point in our discussion of how the results we have obtained might aid in the design of representations for other difficult combinatorial-optimization problems (§7).

In the remainder of §4 we describe four search engines and four representative encodings of NUMBER PARTITIONING in detail. A more extensive set of tests analogous to those described in §5 has been conducted by Ruml (1993); see Table 3.[3] In these descriptions, the expression rand$(i, j)$

[2]All three design principles discussed in this section can be described as "parameterized greed." We use the term parameterized *greediness* to describe this case, since only in this case is the *degree* of greediness subject to control.

[3]The encoding abbreviated "index" employed by Ruml (1993) specifies a definition of *Randomize* different from the one used in this paper.

refers to a number drawn from a uniform distribution of integers in $\{i, i+1, \ldots, j-1, j\}$.

| | RGT | HC | PHC | SA | GA |
|---|---|---|---|---|---|
| direct | • | •, JAMC | • | •, JAMC | ∘ |
| jb-greedy | • | • | • | • | ∘, JB |
| jb-sep | | | | | JB |
| jb-num | ∘ | ∘ | ∘ | ∘ | ∘, JB |
| split | ∘ | ∘ | ∘ | ∘ | ∘ |
| split-alt | ∘ | ∘ | ∘ | ∘ | ∘ |
| split-num | ∘ | ∘ | ∘ | ∘ | ∘ |
| split-grd | ∘ | ∘ | ∘ | ∘ | ∘ |
| prepart | • | • | • | • | ∘ |
| index-dbl | ∘ | ∘ | ∘ | ∘ | ∘ |
| index-wgt | ∘ | ∘ | ∘ | ∘ | ∘ |
| index | • | • | • | • | ∘ |

Table 3: Combinations of search engine and encoding of NUMBER PARTITIONING for which empirical tests are described in the following publications: JAMC = Johnson et al. (1991); JB = Jones and Beltramo (1991); ∘ = Ruml (1993) only; • = both here and Ruml (1993).

## 4.2 Direct representation

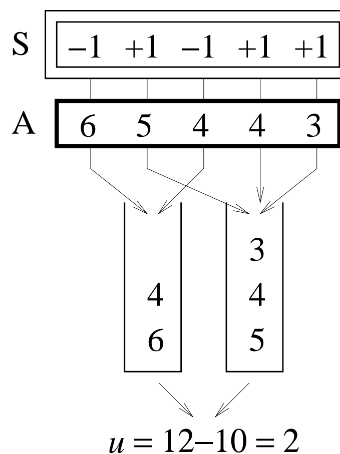


Figure 2: Illustration of the direct representation. The sample solution is $S = (-1, +1, -1, +1, +1)$.

The direct representation (abbreviated "direct" and illustrated in Figure 2) is the obvious encoding for NUMBER PARTITIONING in which the independent variable is the sequence $S$ of signs itself. No decoding is required. The *Randomize* procedure generates one of the $2^n$ possible sign sequences $S$, each with equal probability. The *Perturb* procedure toggles one sign and randomly assigns another: for two indices $i = \text{rand}(1, n)$ and $j = \text{rand}(1, n)$ chosen so that $i \neq j$, the following

assignments are made: $s_i \leftarrow -s_i$ and $s_j \leftarrow 2 \cdot \text{rand}(0,1) - 1$. It is identical to the $SW_2$ operator of Johnson et al. (1991).[4]

The direct representation is clearly *complete*: any of the $2^n$ possible sequences $S$ can be encoded. This is not generally true of the other encodings presented here and by Ruml (1993). Of course, completeness is not necessarily essential to the practicality of an encoding, particularly if the optimal or many low-cost solutions are encodable.

### 4.3 Greedy-decoder representation

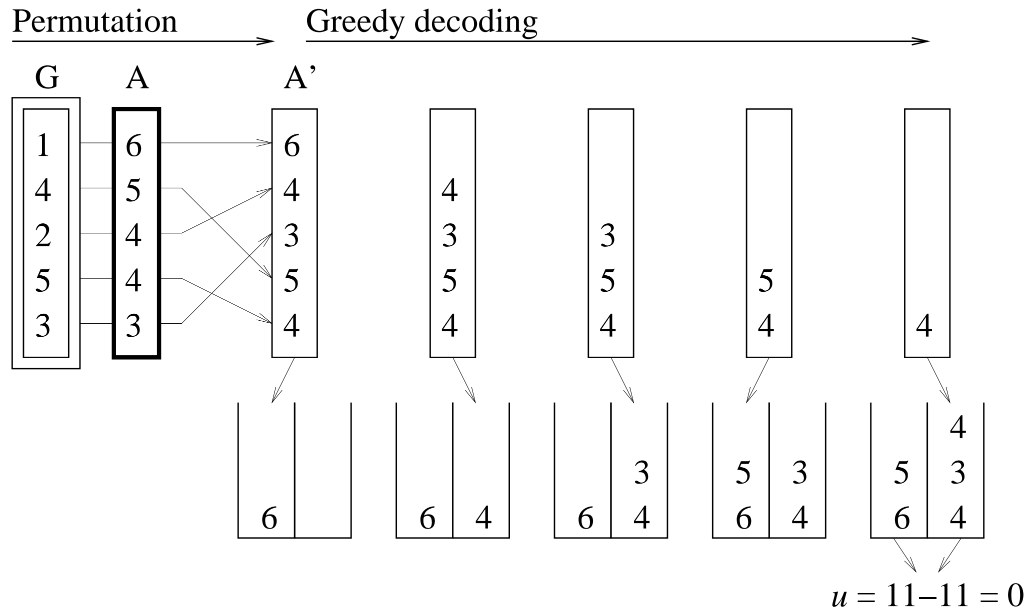


Figure 3: Illustration of the greedy-decoder representation. The sample solution is $G = (1, 4, 2, 5, 3)$.

The greedy-decoder representation was designed from a "best-fit" heuristic by the principle of parameterized arbitration. A candidate solution in the greedy-decoder representation (abbreviated "jb-greedy" and illustrated in Figure 3) is a permutation $G = (g_1, g_2, \ldots, g_n)$ of the sequence $(1, 2, \ldots, n)$. These indices define a permutation $A' = (a'_1, a'_2, \ldots, a'_n)$ of the original sequence $A$:

$$a'_{g_i} \equiv a_i \qquad .$$

The new sequence $A'$ determines the order in which numbers are submitted to a greedy decoder. The decoder creates the sign sequence $S' = (s'_1, s'_2, \ldots, s'_n)$ by the following "best-fit" heuristic:

[4] $SW_2$ is used in preference to $SW_1$ ($i \leftarrow \text{rand}(1,n)$; $s_i \leftarrow -s_i$) because it produces a slightly less rugged search space: the minimum change in $u$ that can be brought about by an invocation of $SW_2$ is expected to be $1/n^2$, compared to $1/n$ in the case of $SW_1$. Tests with $SW_1$ and other operators were performed by Ruml (1993).

```
u ← 0
for i = 1 to n
    if u < 0
        then s'_i ← +1
        else  s'_i ← −1
    u ← u + s'_i a'_i
end for
```

The desired sign sequence $S$ is constructed from $S'$ by inverting the permutation used to construct $A'$ from $A$:

$$s_i \equiv s'_{g_i} \quad .$$

The *Randomize* procedure generates one of the $n!$ possible permutations $G$, each with equal probability. The *Perturb* procedure swaps a pair of indices $g_i$ and $g_j$, where $i = \text{rand}(1, n)$, $j = \text{rand}(1, n)$, and $i \neq j$.

This encoding is not complete. For example, the (maximally non-optimal) sequence $S = (+1, +1, \ldots, +1)$ cannot be generated by any permutation $G$ since the decoder cannot generate a sequence in which $s'_1 s'_2 = 1$. We do not believe that this is a material limitation of the representation because, as we demonstrate constructively in the remainder of this subsection, the optimal sequence can always be encoded.

Given an optimal partitioning $S$ of a given sequence $A$, the following procedure constructs the encoding $G$ of $S$. Let $count(S, s)$ be the number of elements $s_i \in S$ such that $s_i = s$. Let $fetch(S, s)$ be some index $i$ such that $s_i = s$. Let $append(G, i)$ be the sequence formed by appending element $i$ to the sequence $G$.

```
G ← ()
u ← 0
while (count(S, +1) + count(S, −1) > 0)
    if u < 0
        then s ← +1
        else  s ← −1
    i ← fetch(S, s) [◇]
    G ← append(G, i)
    u ← u + s_i a_i
    s_i ← 0
end while
```

If, at some iteration of this procedure, the operation marked [◇] cannot be executed (i.e., no remaining element of $S$ has value $s$), then toggling any nonzero element of $S$ will improve the residue finally attained. The original $S$ therefore could not have been optimal; this contradicts the premise of the proof. Therefore, any optimal partitioning $S$ can be encoded in the greedy-decoder representation.

### 4.4 Prepartitioning representation

A candidate solution in the prepartitioning representation (abbreviated "prepart" and illustrated in Figure 4) is a sequence $P = (p_1, p_2, \ldots, p_n)$ of integers $p_i \in \{1, 2, \ldots, n\}$. It is decoded in two steps: (1) a preprocessing step in which certain elements of $A$, specified by the sequence

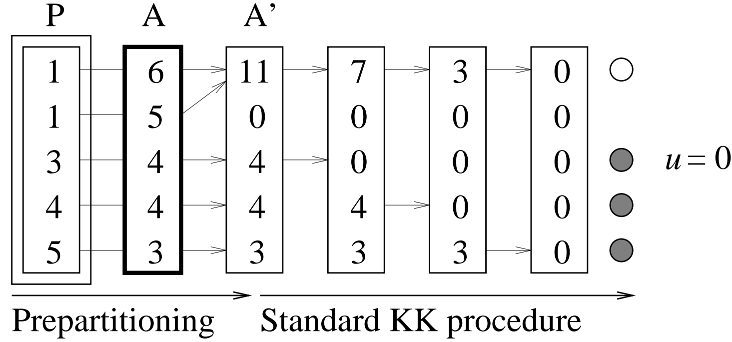


Figure 4: Illustration of the prepartitioning representation. The candidate solution is $P = (1, 1, 3, 4, 5)$.

$P$, are added together; and (2) application of the standard KK heuristic to the resulting modified sequence.

This representation was designed from the KK algorithm by the principle of parameterized constraint: the sequence $P$ specifies subsets of $A$ whose elements are constrained to lie in the same partition. Specifically, the preprocessing step proceeds as follows. (Let $A' = (a'_1, a'_2, \ldots, a'_n)$ be a sequence of accumulators that will store the modified sequence.)

$A' \leftarrow (0, 0, \ldots, 0)$
**for** $j = 1$ **to** $n$
  $a'_{p_j} \leftarrow a'_{p_j} + a_j$
**end for**

The net effect of this procedure is to add together any elements $a_i$ and $a_j$ for which $p_i = p_j$. The modified sequence is then processed by the usual KK procedure. Note that the KK solution may be encoded as $P = (1, 2, \ldots, n)$.

The *Randomize* procedure generates each element of $P$ from a random uniform distribution: $p_1 \leftarrow \text{rand}(1, n)$, $p_2 \leftarrow \text{rand}(1, n)$, and so forth. The *Perturb* procedure assigns $p_i \leftarrow j$, where $i = \text{rand}(1, n)$, $j = \text{rand}(1, n)$, and $j \neq p_i$.

The prepartitioning representation is complete: any given sign sequence $S$ can be constructed by assigning, say, $p_i \leftarrow s_i + 2$ for every $i$. Under this construction, the two nonzero prepartitions are identical to the partitions generated by the given sign sequence $S$, so nothing remains to be done by the subsequent application of the KK heuristic.

## 4.5 Index-rules representation

The index-rules representation (abbreviated "index" and illustrated in Figure 5) was developed from the KK algorithm by the principle of parameterized greediness. The independent variable in the encoding is a sequence $R = (r_0, r_1, \ldots, r_{n-2})$ of rules $r_i \in \{0, 1, \ldots, n-2-i\}$. The decoder is run by executing a sequence of $n-1$ differencing operations:[5]

[5] This decoder is a realization of the suggestion by Karmarkar and Karp (1982) that "differencing operations. . . can

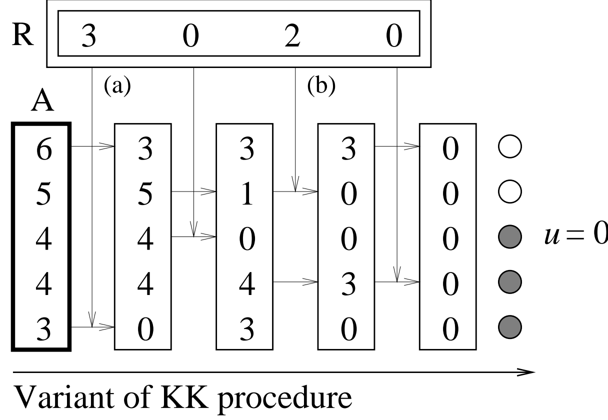


Figure 5: Illustration of the index-rules representation. The candidate solution is $R = (3, 0, 2, 0)$. (a) $r_0 = 3$, so in the first step, the largest and fifth-largest elements are selected for differencing; and (b) $r_2 = 2$, so in the third differencing operation, the largest and fourth-largest elements are selected for differencing.

```
for i = 1 to n − 1
    j ← the index of the largest element of A
    k ← the index of the (2 + r_i)th largest element of A
    a_j ← a_j − a_k
    a_k ← 0
end for
```

As in the usual KK algorithm, a two-coloring procedure generates the sign sequence $S$. Note that the standard KK heuristic (differencing of the two largest elements in $A$) is executed in any step $i$ for which $r_i = 0$; in particular, the KK solution may be encoded as $R = (0, 0, \ldots, 0)$. Note also that the range of values of $r_i$ decreases as $i$ increases; in particular, $r_{n-2} = 0$, which is appropriate because the number of nonzero elements in $A$ after $n - 2$ differencing operations is at most two.

The *Randomize* procedure draws each $r_i$ independently from its own uniform distribution of integers $0 \leq r_i \leq n - 2 - i$.

The *Perturb* procedure is designed to make all distinct, legal changes $r_i \leftarrow j$ (where $r_i \neq j$) equally probable. Merely assigning $i \leftarrow \text{rand}(0, n - 3)$ and $j \leftarrow \text{rand}(0, n - 2 - i)$ would make $r_{n-3} \leftarrow 0$ and $r_{n-3} \leftarrow 1$ the most probable changes. One way to avoid this undesirable effect is to draw $i$ from a probability distribution that involves a square root. The method that we use is as follows:

be used as subroutines in more complicated heuristic algorithms."

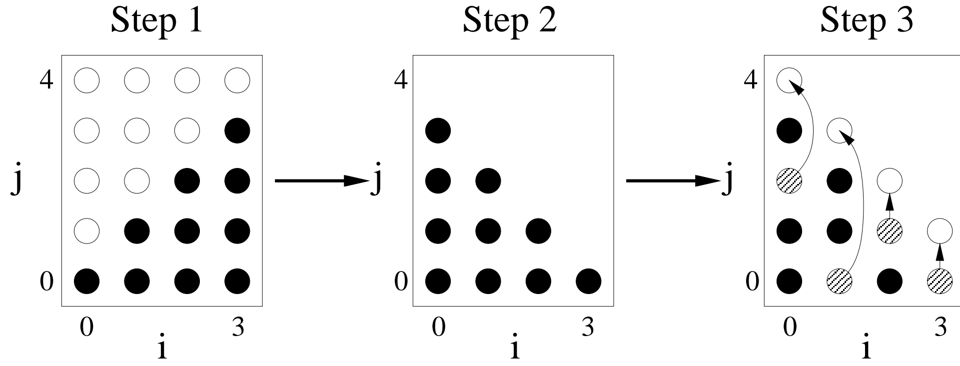


Figure 6: Choosing $i$ and $j$ for use in the *Perturb* operation in the index-rules representation. See text.

$i \leftarrow \text{rand}(0, n-3)$
$j \leftarrow \text{rand}(0, n-2)$
**if** $(i \geq j)$
  **then** $i \leftarrow n-3-i$
  **else** $j \leftarrow n-2-j$
**if** $(j = r_i)$ **then** $j \leftarrow n-2-i$
$r_i \leftarrow j$

Figure 6 demonstrates the procedure for the case $n = 6$, $R = (2, 0, 1, 0, 0)$. A pair of integers $(i, j)$ is chosen from a uniform distribution over the rectangle $(0, 0) \leq (i, j) \leq (3, 4)$. If $i \geq j$ (black circles, Step 1), the point is reflected across the vertical bisector of the rectangle; otherwise, it is reflected across the horizontal bisector. At this point, $(i, j)$ has effectively been chosen from a random distribution chosen over the triangle depicted in Step 2. If $j = r_i$ (hatched circles, Step 3), then the assignment $j \leftarrow n-2-i$ is made (white circles). Thus, every pair $(i, j)$, such that $0 \leq i \leq n-3$, $0 \leq j \leq n-2-i$, and $j \neq r_i$, is equally probable.

Like the greedy-decoder representation, the index-rules representation is not complete; for example, there is no way to encode the sequence $S = (+1, +1, \ldots, +1)$. In this case, the reason is that every differencing operation constrains two elements to lie in different partitions (i.e., $s_j s_k = -1$). As with the greedy-decoder representation, we do not believe that lack of completeness is an important limitation of the index-rules representation because, as we demonstrate constructively in the remainder of this subsection, the optimal sequence can always be encoded.

Given an optimal partitioning $S$ of a given sequence $A$, the following procedure constructs the encoding $G$ of $S$. Let $count(S, s)$ be the number of elements $s_i \in S$ such that $s_i = s$. Let $append(R, i)$ be the sequence formed by appending element $i$ to the sequence $R$. Let $rank(a_i, A)$ be the rank of element $a_i$ in $A$: 1 if $a_i$ is the largest element in $A$, 2 if it is the second largest, and so forth.

```
R ← ()
while (count(S,+1) + count(S,−1) > 0)
    i ← index of largest a_i ∈ A for which s_i ≠ 0
    j ← index of largest a_j ∈ A for which s_j = −s_i [◇]
    R ← append(R, rank(a_j, A) − 2)
    a_i ← a_i − a_j
    s_j ← 0
end while
```

If, at some iteration of this procedure, the operation marked [◇] cannot be executed (i.e., all nonzero elements of $S$ are of the same sign), then toggling any nonzero element of $S$ will improve the residue finally attained. The original $S$ therefore could not have been optimal; this contradicts the premise of the proof. Therefore, any optimal partitioning $S$ can be encoded in the index-rules representation.

### 4.6 Search engines

We now describe the four search engines for which we present results in this paper. Parallel hill climbing, the most successful search engine in most of our tests, is described in detail; the other three search engines are summarized in pseudocode.

Random generate and test (RGT), which is used as a control case that does not employ the *Perturb* operation, merely returns the best of many candidate solutions, each generated independently by *Randomize*:

```
ξ ← Randomize()
for iter = 1 to max_iter
    ξ' ← Randomize()
    if u(ξ') < u(ξ) then ξ ← ξ'
end for
return ξ
```

Hill climbing (HC) executes a random walk in the search space, accepting only steps that produce improvement:

```
ξ ← Randomize()
for iter = 1 to max_iter
    ξ' ← Perturb(ξ)
    if u(ξ') < u(ξ) then ξ ← ξ'
end for
return ξ
```

Simulated annealing (SA) is a variant of HC in which steps that produce setbacks (i.e., steps for which $\Delta u > 0$) are accepted with probability exponential in $\Delta u$:

```
ξ ← Randomize()
for iter = 1 to max_iter
    ξ' ← Perturb(ξ)
    Δu ← u(ξ') − u(ξ)
    if Δu ≤ 0
        then ξ ← ξ'
        else  ξ ← ξ' with probability exp(−Δu/T(iter))
end for
return ξ
```

The temperature parameter T(iter), which controls the tolerance with which setbacks $\Delta u > 0$ are accepted, decreases according to a fixed schedule.[6]

In PHC (Figure 7), a population of candidate solutions is maintained. We use a population of 1,000 in the examples in this paper. These are initially generated by the *Randomize* procedure. Each candidate solution is evaluated, i.e., its residue $u$ is calculated. Thereafter, at each iteration of the algorithm, the following steps are executed:

1. A solution, selected at random from the population, is duplicated and the copy is subjected to *Perturb*. The probability distribution for this selection is such that solutions with smaller residues are more likely to be chosen.[7]

2. The worst member of the population, i.e., the candidate solution with the largest value of $u$ (possibly the solution just created by *Perturb*), is deleted.

This procedure is repeated for some predetermined number of iterations.

PHC usually outperformed the other SO variants we tested, in terms of the solution cost attained after 30,000 trial solutions (§5.4).

Results (Ruml, 1993) from tests with a genetic algorithm (GA), which are very similar to those obtained with the other search engines, are omitted to avoid complicating the presentation with comparisons of various crossover operators.

## 4.7 Computational complexity

If, as in our multiple-precision implementation, arithmetic operations on the elements of $A$ dominate the computational cost of decoding, then all four representations presented here are approximately equivalent with respect to the time required per evaluation: each requires $n - 1$ additions and

[6] The schedule used in the runs presented here is $\mathrm{T(iter)} = 0.01(0.8)^{\lfloor \mathrm{iter}/300 \rfloor}$. Runs with other temperature schedules more carefully tailored to the individual representations give essentially the same results with regard to the important comparisons in this paper (Johnson et al., 1991; Ruml, 1993). Although it is possible that further careful tuning of the annealing schedule could improve the performance of SA, we feel that this is unlikely given the results presented in §6.3.

[7] In particular, we employ linear rank-based selection (Davis, 1991), in which the probability that a given solution is selected is related linearly to its rank in the population. The two independent parameters of the linear relation are determined by (1) the constraint that the probabilities sum to unity, and (2) a parameter called the *rank factor*, defined as the ratio between the selection probabilities of the worst and best individuals in the population. In all PHC runs presented here, the rank factor was set to 0.8.

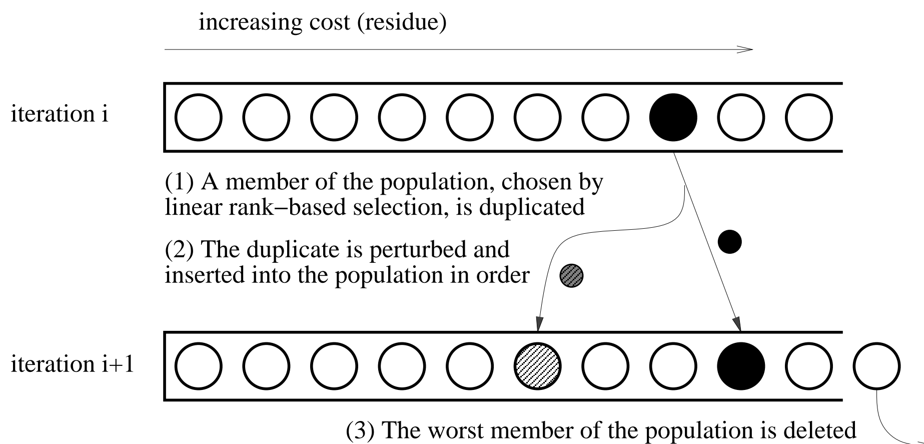


Figure 7: Single iteration of parallel hill climbing.

subtractions. However, when operations associated with accessing elements of $A$ are taken into account, the representations fall into two categories with regard to their asymptotic performance. Whereas the direct and jb-greedy representations can be decoded in $O(n)$ time, decoding in the index-rules[8] and prepartitioning representations requires executing the KK heuristic and therefore has $O(n \log n)$ complexity.

# 5 Empirical comparisons

## 5.1 Experimental strategy

The empirical comparisons described here, like those conducted by Ruml (1993), investigated primarily how the performance of stochastic optimization for a given instance of Number Partitioning is influenced by the following three factors:

1. Choice of search engine.
2. Choice of representation.
3. Use of seeding.

Because of the difficulty of varying all three factors exhaustively, factors were varied one or two at a time.

The first two factors are discussed in detail above. The third factor, *seeding*, refers in this paper to the use of the KK solution wherever a solution generated by *Randomize* is normally called for.

[8] In the index-rules representation, decoding is done using a modified form of the KK procedure that does not always select the two largest numbers for differencing. The $O(n \log n)$ complexity can be brought about by using a data structure in which both fetching of the $i$'th largest element and insertion of the difference can be accomplished in $O(\log n)$ time.

This is particularly natural to do in the case of the prepartitioning and index-rules representations, which are parameterized-constraint and parameterized-greediness encodings, respectively. One motivation for testing the effect of seeding is to develop better algorithms for NUMBER PARTITIONING. A second, equally important motivation is to help distinguish between alternative hypotheses for the relative performance of the various representations (§6).

A *batch* refers to a set of 100 SO runs[9] carried out under identical[10] conditions. We ran such a batch on the same[11] 100-element instance of NUMBER PARTITIONING for each combination of encoding, search engine, and seeding tested. All instances of NUMBER PARTITIONING used in our tests were generated by drawing numbers from a uniform distribution over (0,1]. Curves (as in Figure 8) depict the progress of runs in a given batch. In each case, the ordinate is the number of candidate solutions evaluated in a run, and the abscissa is the mean value of $\log_{10} \hat{u}$ averaged over all runs in the batch, where $\hat{u}$ is the best residue encountered by a given run after the given number of trials, normalized against the KK solution (i.e., $\hat{u} \equiv u/u_{\mathrm{KK}}$). Each error bar represents the standard error of the mean of $\log_{10} \hat{u}$, computed separately above and below the curve. For this instance of NUMBER PARTITIONING, $u_{\mathrm{KK}} = 1.084 \times 10^{-7}$.

Although results are reported as if each element $a_i \in A$ were a rational number in the range $0 \leq a_i \leq 1$, all runs employed multiple-precision integer arithmetic to avoid any dependence of the calculated residue $u$ on the order in which numbers were added or subtracted.[12] The number of digits of precision was chosen such that the probability that the optimal residue would round to zero, were it to be represented in integer form, is less than $10^{-5}$. Table 4 shows the number of digits used in the 100-element instance and with the other problem-instance sizes for which results are presented in §7.

### 5.2 Dependence on algorithm

Figure 8 presents performance curves for all four search engines with the direct representation of a single 100-element instance of NUMBER PARTITIONING. While there appears to be some variation in the performance of the various search engines, it is insignificant in most cases relative to the variation among runs with a single search engine. In the exceptional case, PHC, the geometric mean residue obtained after 30,000 iterations still falls more than three orders of magnitude short of the KK solution. (These results also confirm for four search engines what Johnson et al. (1991) observed with simulated annealing: in our runs, little improvement is observed after 15,000 iterations, and the best solution encountered is (on average) four orders of magnitude worse than the KK solution.) If only the direct representation were to be considered, it would appear that SO techniques are—for practical purposes—far inferior to the KK algorithm.

### 5.3 Dependence on representation

---

[9]The sole exception is in §7, where a batch refers to a set of 30 runs.

[10]A different random-number seed was employed in each run.

[11]Ruml (1993) obtained similar results with other 100-element instances of NUMBER PARTITIONING.

[12]For this reason, our experimental results cannot be compared quantitatively with those of Johnson et al. (1991), which were obtained with floating-point arithmetic, although they agree qualitatively.

| $n$ | $u_{\text{lo}}$ | Digits |
|---|---|---|
| 50 | $7.2 \times 10^{-21}$ | 21 |
| 100 | $9.1 \times 10^{-36}$ | 36 |
| 150 | $9.9 \times 10^{-51}$ | 51 |
| 200 | $1.0 \times 10^{-65}$ | 65 |
| 250 | $1.0 \times 10^{-80}$ | 80 |
| 300 | $9.8 \times 10^{-96}$ | 96 |
| 350 | $9.4 \times 10^{-111}$ | 111 |
| 400 | $8.9 \times 10^{-126}$ | 126 |
| 450 | $8.4 \times 10^{-141}$ | 141 |
| 500 | $7.9 \times 10^{-156}$ | 156 |

Table 4: Number of digits of precision used in each experiment. The number of digits of precision is $\lceil -\log_{10} u_{\text{lo}} \rceil$, where $u_{\text{lo}}$ is a residue value defined such that for large $n$, the probability that $u_{\text{opt}} \leq u_{\text{lo}}$ is less than $10^{-5}$ (see appendix).

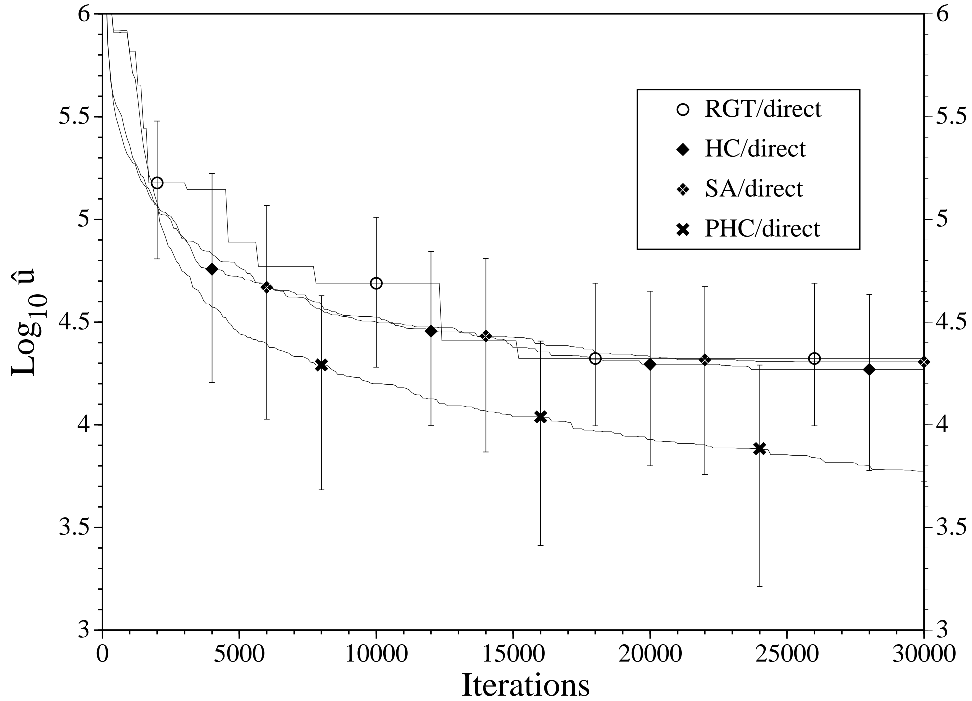


Figure 8: Performance curves for all search engines with direct encoding. Recall that by definition, $log_{10}\hat{u} = 0$ for the KK solution.

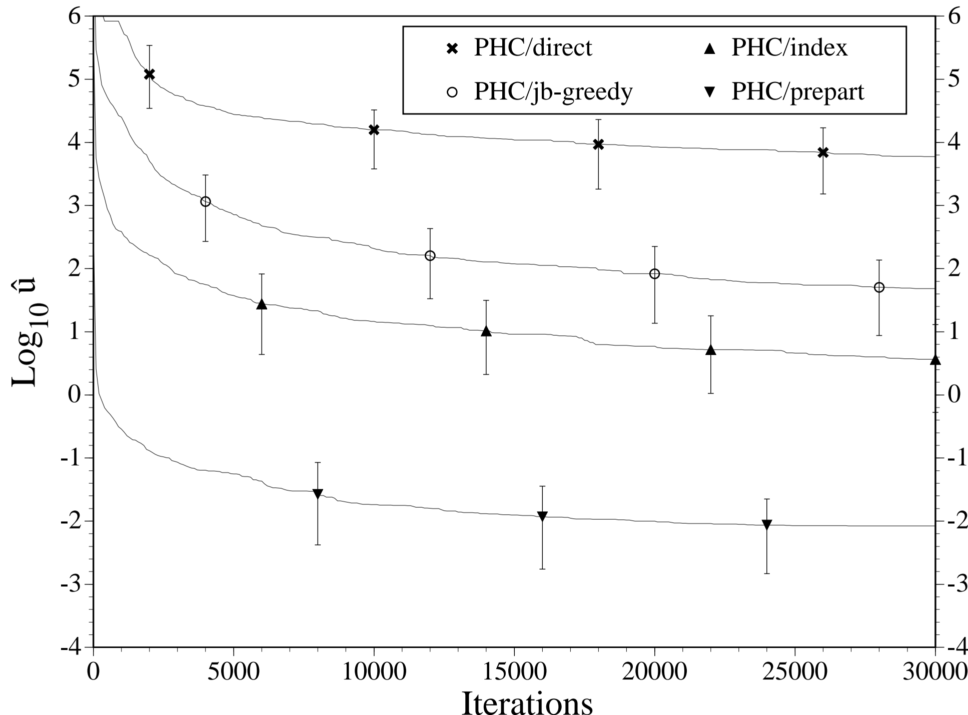


Figure 9: Performance curves for all encodings with PHC search engine.

Comparisons of runs with one search engine (PHC) and a variety of representations tell a different story (Figure 9). When the greedy-decoder representation of Jones and Beltramo (1991) is employed, PHC finds solutions that, on average, come within two orders of magnitude of the KK solution. With the index-rules encoding, PHC comes within an order of magnitude of the KK solution on average and, as indicated by the large variance, frequently surpasses the KK solution within a few tens of thousands of iterations. With the prepartitioning representation, PHC improves upon the KK solution by two orders of magnitude on average within 30,000 iterations. Thus, the prepartitioning representation may be said to be "more easily searched" than the direct representation. From these observations, it is clear that an investigation of problem encoding is critical to any attempt to apply SO to NUMBER PARTITIONING.

## 5.4 Relative importance of search engine and representation

Using the same 100-element instance of NUMBER PARTITIONING, we ran a batch of one hundred 30,000-iteration runs for each of the sixteen SO variants that can be constructed by matching one of the four search engines RGT, HC, SA, and PHC with one of the four encodings direct, jb-greedy, index, and prepart. Results are summarized in Figure 10, which depicts the batch-average value of $\log_{10} \hat{u}$ after 30,000 iterations for each SO variant. Switching between search engines as different as RGT and PHC effects a change in final solution cost of one or two orders of magnitude, whereas a change in representation can bring about a change in final residue of seven orders of magnitude. Using SO variants constructed by matching five search engines (including a genetic algorithm) with ten encodings, Ruml (1993) obtained similar results. The results shown in Figure 10 should be compared with those presented in Figure 20, in which seeding (§5.5) and gravitation (§6.3) are

used to advantage.

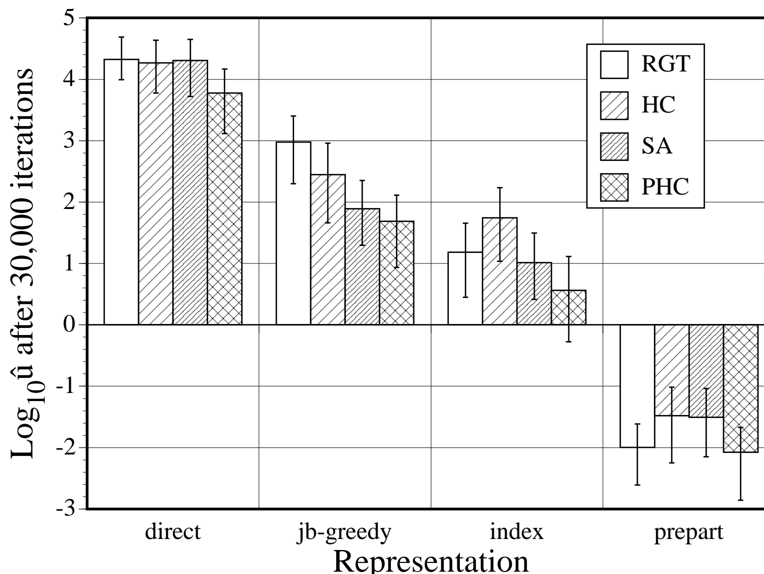


Figure 10: Performance for all search engines and encodings.

### 5.5 Effect of seeding

In parameterized-constraint and parameterized-greediness representations of NUMBER PARTITIONING based on the KK heuristic, the KK solution may be encoded in a straightforward manner. This raises the possibility of *seeding* SO runs, i.e., using the KK solution wherever a solution generated by *Randomize* is normally called for. We investigated the effects of seeding on SO performance for two reasons. First, we were interested in whether seeding would improve SO performance other than at the start of a run. Second, the seeding results help distinguish between alternative hypotheses for the superiority of certain representations (§6).

As demonstrated in Figure 11, seeding significantly improves the performance of PHC/index and PHC/prepart. The effect is greater in the former case: although unseeded PHC/prepart is significantly better than unseeded PHC/index, the seeded batches are within experimental error of each other. Both better the KK solution by three orders of magnitude on average after 30,000 iterations on a 100-element problem instance.

## 6 Analysis

The empirical comparisons presented in §5 raise a poignant question: what accounts for the strong dependence of SO performance on problem representation? To help answer this question, we consider the following characterizations of the search space generated by each given representation:

- "Neighborhood structure"—the tendency of solutions that lie near one another to have similar residue values.

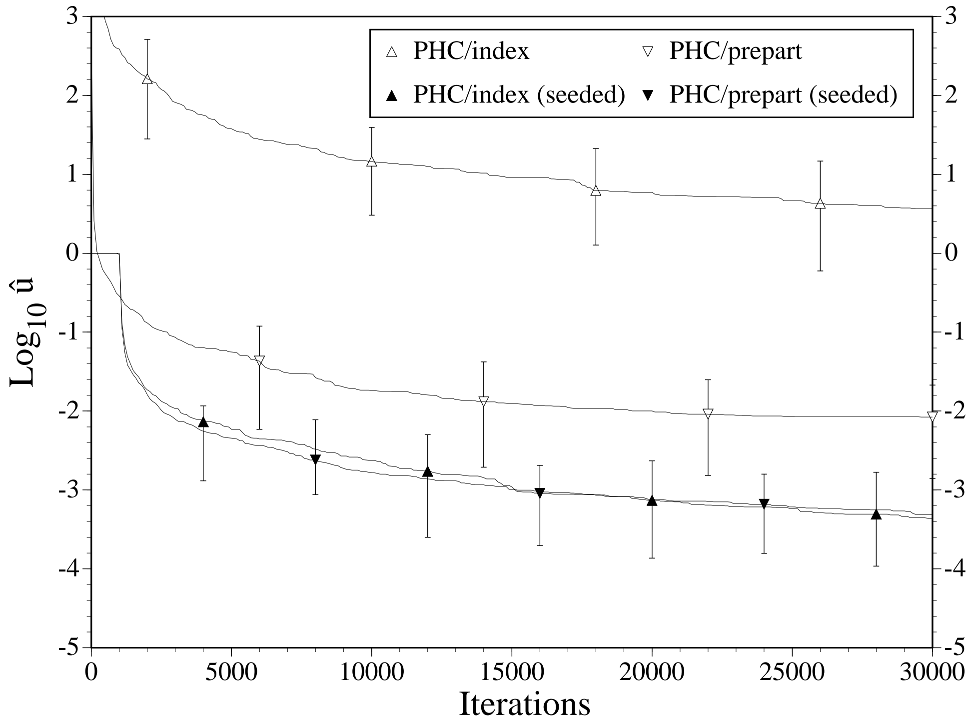


Figure 11: Performance curves for seeded and unseeded PHC with index-rules and prepartitioning encodings. The performance curves for the seeded runs are flat for the first 1,000 iterations because, as we have defined seeding, seeded PHC begins by executing the KK heuristic 1,000 times. In a practical implementation of seeded PHC one would obviously avoid the duplicated computational effort.

- A priori distribution—the distribution of residue values generated by *Randomize*.
- "Gravitation"—the concentration of low-residue solutions around the greedy solution in a parameterized-constraint or parameterized-greediness encoding. In the context of the index-rules and prepartitioning encodings of NUMBER PARTITIONING, gravitation refers to dependence of $\log u$ on the distance between a given solution and the KK solution.

The essence of SO is to exploit these and other similar properties of the search space, when they exist.

## 6.1 Neighborhood structure

A property of a search space commonly associated with efficient stochastic optimization is the presence of "neighborhood structure" (Johnson et al., 1991). To satisfy conflicting goals of intuitive clarity, relevance to SO design, and measurability, we employ three different but interrelated definitions of neighborhood structure.

1. Informally, neighborhood structure refers to the tendency of solutions near each other in the search space to have similar scores. This definition leads to the equally informal notion that

SO looks for good solutions in regions of the search space where it has found other good solutions.

2. Operationally, we say that a search space contains neighborhood structure if evaluating particular candidate solutions can increase the amount of information that the search engine has about where to look for good solutions. Thus, stating that neighborhood structure is absent in a search space is equivalent to stating that no search engine can do better than RGT in that space.

3. Statistically, we say that neighborhood structure is absent if and only if the score of a solution is statistically independent of the scores of its nearest neighbors, next-nearest neighbors, and so forth. A "nearest neighbor" of a given solution $\xi$ is defined as a solution $\xi'$ (possibly the same solution) that can be generated by executing $Perturb(\xi)$.

Of these three, only the statistical definition can be measured directly in a given search space. Moreover, only a subset of the conditions can be applied, given limited computational resources:

- Two random variables $x$ and $y$ are statistically independent if and only if $E[f(x)g(y)] = E[f(x)]E[g(y)]$ for *any* choice of scalar functions $f$ and $g$. The relation must hold for an infinite number of choices of $f$ and $g$—typically $f(x) = x^m$, $0 \leq m < \infty$; $g(y) = y^{m'}$, $0 \leq m' < \infty$. We restrict our investigation to linear $f$ and $g$ by employing only the correlation coefficient (Papoulis, 1990). A similar approach to detecting neighborhood structure was taken by Manderick et al. (1991).

- A search space can, in principle, contain statistical dependence among the scores of next-nearest neighbors but not nearest neighbors. In general, the existence of neighborhood structure in a search space cannot be ruled out without testing for statistical dependence among solution scores at all distances. We restrict our investigation to nearest and next-nearest neighbors.

We define the *neighborhood-structure coefficient* $r_k$ of a given encoding of NUMBER PARTITIONING to be the correlation coefficient of the random variables $\log_{10} \hat{u}_1$ and $\log_{10} \hat{u}_2$, where $u_1$ and $u_2$ are residues of two solutions accessible to each other by $k$ perturbations. Values of $r_k$ close to zero suggest, though they do not unambiguously demonstrate, the absence of neighborhood structure.

The neighborhood-structure hypothesis—the hypothesis that the representations with which SO is most successful are those with the most neighborhood structure—is intuitively attractive. However, for all four NUMBER PARTITIONING encodings investigated here, neighborhood structure turns out to be essentially absent except in irrelevant portions of the search space. For example, let us consider the direct encoding. We estimated $r_k$ by generating 2,000 neighbor pairs with residues below a given limit $u_{\text{lim}}$:

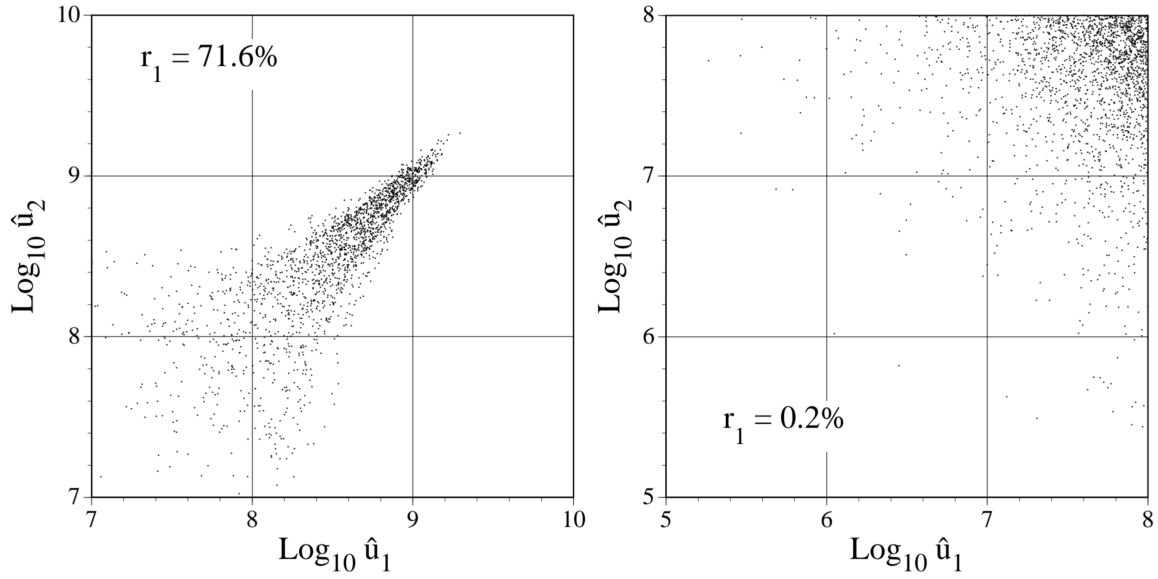


Figure 12: Measurement of $r_1$ in the direct representation, with $\log_{10} \hat{u}_{\rm lim} = 100$ (left panel) and $\log_{10} \hat{u}_{\rm lim} = 8$ (right panel).

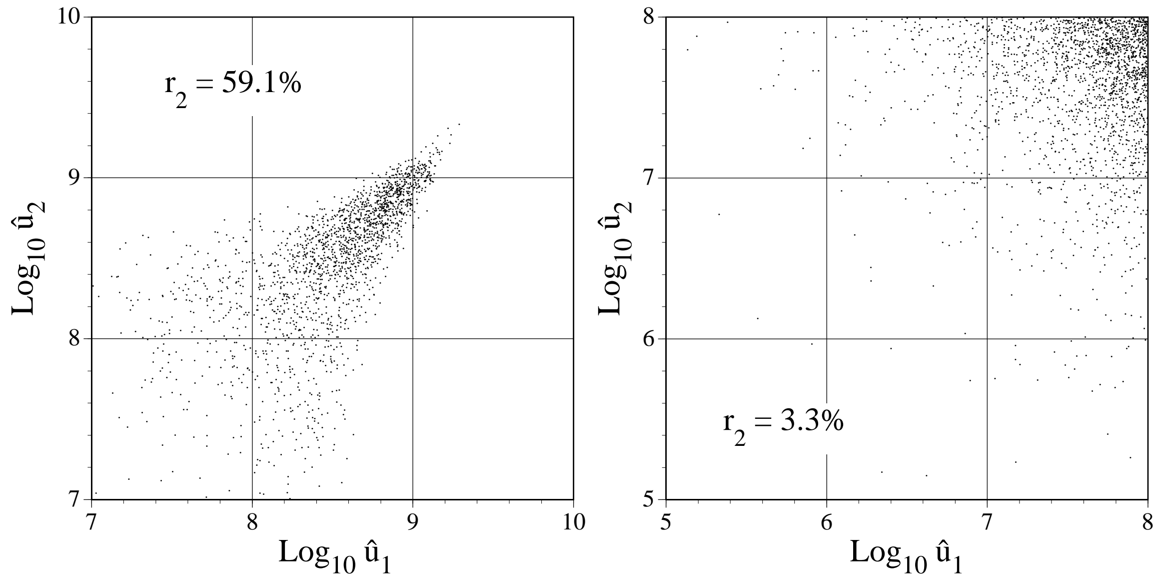


Figure 13: Measurement of $r_2$ in the direct representation, with $\log_{10} \hat{u}_{\rm lim} = 100$ (left panel) and $\log_{10} \hat{u}_{\rm lim} = 8$ (right panel).

npts ← 0
**repeat**
  $\xi \leftarrow$ *Randomize()*
  **if** $(u(\xi) > u_{\text{lim}})$ **then** reject this data point, i.e., skip to "until"
  $\xi' \leftarrow$ *Perturb(ξ)*
  **repeat** $k-1$ **times:** $\xi' \leftarrow$ *Perturb(ξ')*
  **if** $(u(\xi') > u_{\text{lim}})$ **then** reject this data point
  **if** $(u(\xi) = u(\xi'))$ **then** reject this data point
  Use this data point $(\log_{10} u(\xi), \log_{10} u(\xi'))$
  npts ← npts + 1
**until** npts $= 2,000$

When $u_{\text{lim}} = 100$, there is a clear correlation between the scores of neighbors (Figure 12, left panel). This correlation is present because the perturbation operator that we have employed in the direct encoding can bring about a change in $u(\xi)$ of at most 4; this constraint is visible on the log-log plot as a horn-shaped envelope. Thus, neighborhood structure plays a role in reducing $u$ when the trial solutions involved are so imbalanced that transferring any element from the larger partition to the smaller reduces the residue. In the regime $\log_{10} \hat{u}_{\text{lim}} < 8$, there is no correlation between neighbors (Figure 12, right panel), indicating that in the direct representation, neighborhood structure plays no significant role in refining $u$ beyond that level.[13] Correlation between the scores of next-nearest neighbors, which in principle can exist even when $r_1 = 0$, follow a similar pattern (Figure 13). Extending this analysis to higher values of $k$ and the other three encodings yields similar patterns (Figure 14). In each case, $\log_{10} \hat{u}_{\text{lim}}$ is set to the lowest integral value permitted by computational resources.

Because we have tested only a subset of the infinite number of statistical measures of neighborhood structure, these results do not entirely preclude the existence of neighborhood structure among the solutions of interest. The issue of neighborhood structure is revisited in §6.3 using the operational definition.

## 6.2 A priori distribution

Given that the search spaces generated by the representations described here appear to lack neighborhood structure, except among irrelevant high-cost solutions, it is reasonable to ask whether variations in SO performance across representations can be accounted for solely by differences in the probability distribution of the solutions in each search space.

(In this and the next subsection, we use the following notation. Let $f_i(\xi)$ be the probability distribution from which an SO run generates a candidate solution in its $i$th iteration. Let iteration 0 be the process of generating initial random guesses, so that $f_0(\xi)$ is the probability distribution employed by the *Randomize* procedure of the given representation. Let $h_i(\log u)$ be the probability distribution of residue values $\log u$ that results from using $f_i(\xi)$.)

In support of this hypothesis, histograms of $\log_{10} \hat{u}$ in a random sampling of 10,000 solutions generated by *Randomize* (Figure 15) are shifted by several orders of magnitude relative to one another. The performance of the random generate-and-test (RGT) search engine, for which $h_i(\log u) = h_0(\log u)$ in every iteration $i$, follows a similar pattern (Figure 10).

[13] Note that the limit $\log_{10} \hat{u}_{\text{lim}} < 8$ is quite large: it admits solutions with residue values that approach 10.

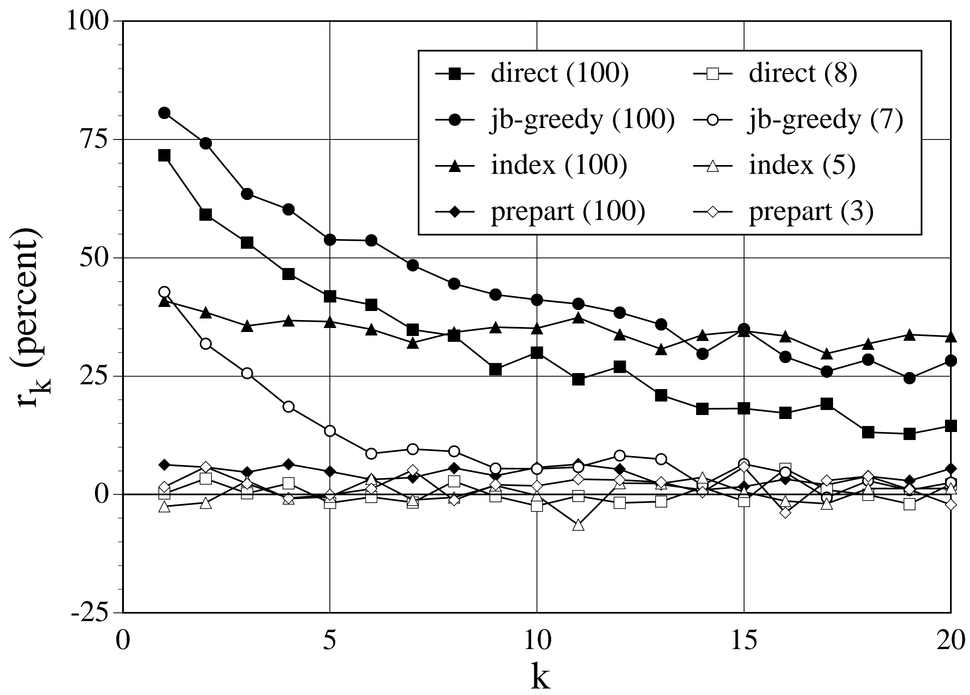


Figure 14: Value of $r_k$ as a function of $k$, for all four encodings. The parenthesized numbers in the legend are values of $\log_{10} \hat{u}_{\mathrm{lim}}$.

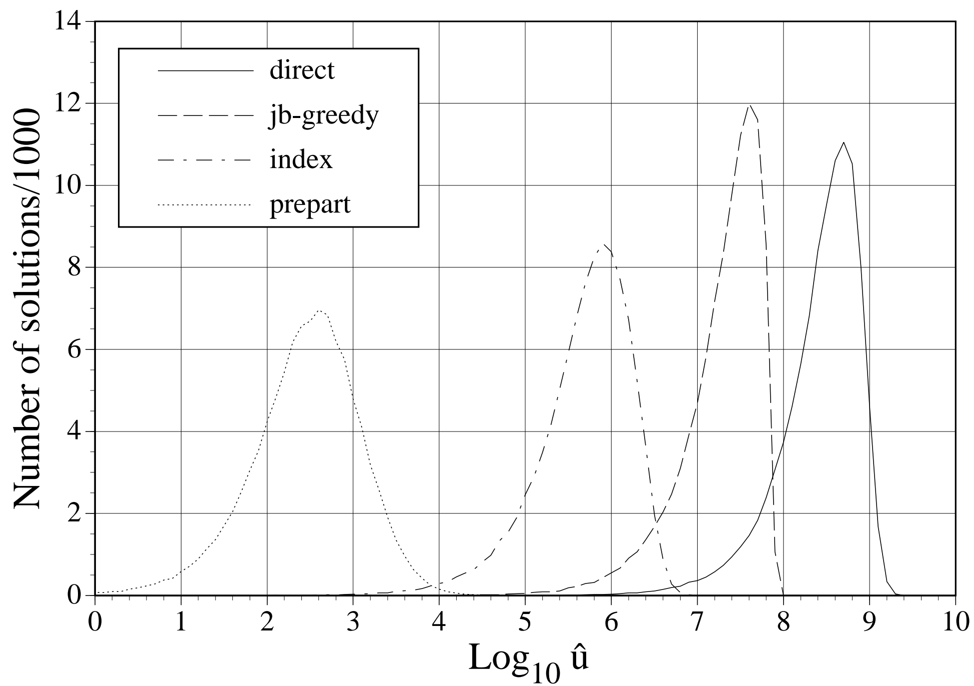


Figure 15: Differences in $h_0$ distribution among four encodings.

### 6.3 Gravitation

Differences in the a priori distribution of $u$ values account for the six-order-of-magnitude disparity in performance between RGT/direct and RGT/prepart, both of which operate independently of structure in the search space. However, such differences do not explain the following observations concerning the index and prepart encodings:

- With the index and prepart representations, RGT is slightly outperformed (§5.4) by PHC, which operates by attempting to exploit neighborhood structure.
- With PHC/index and PHC/prepart, seeding with the KK solution significantly improves performance (§5.5).

Both of these observations appear to demonstrate that in spite of the apparent absence of neighborhood structure among all but the worst solutions in the index and prepart representations, $h_i(\log u)$ does depend on $f_i(\xi)$. Thus, as the run proceeds, the search engine can improve its odds of finding near-optimal solutions by modifying $f_i(\xi)$ to reflect the distribution of residues in the search space.

In this subsection we attempt to explain these observations by testing the following hypothesis: in the index-rules and prepartitioning representations, the expected residue of a solution increases as the decoding procedure departs from the greedy algorithm on which these representations are based. This type of correlation, which we call "gravitation," is defined for representations designed by parameterized constraint and parameterized greediness, but not for direct representations and those designed by paramterized arbitration; the latter do not involve gradual departure from a greedy algorithm.

More precisely, we hypothesize that the $u$ value of a candidate solution in these representations is correlated with its distance from the KK solution, where the metric is encoding-specific. With the index-rules representation, we use $\sum_{j=0}^{n-1} r_j$ as the measure of distance from the KK solution. With the prepartitioning representation, we use the minimum number of constraints of the form $s_j s_k = 1$ required to produce the given constraint set.

Figure 16, which demonstrates that the prepartitioning representation is consistent with the hypothesis, is a plot of $\log_{10} \hat{u}(\xi_j)$ vs. $j$ for 25 independently generated sequences $(\xi_0, \xi_1, \ldots, \xi_{99})$, each defined so that $\xi_0$ is the KK solution, $\text{dist}(\xi_j,\xi_{j-1}) = 1$, and $\text{dist}(\xi_j,\xi_0) = j$. Figure 17, which shows an even stronger correlation in the index-rules representation, is a similar plot for 3 independently generated sequences $(\xi_0, \xi_1, \ldots, \xi_{4000})$. In accord with the observed correlations, the a priori distribution $h_0(\log_{10} u)$ shifts by several orders of magnitude when *Randomize* is restricted to generate candidate solutions in the immediate neighborhood of the KK solution (Figures 18 and 19).[14] Note that the neighborhood of the KK solution in which most of the low-residue solutions reside occupies a negligible portion of the search space, since the solutions are organized into a "bumpy funnel" of high dimensionality.

The distinction between gravitation and the type of neighborhood structure discussed in §6.1 has important implications for algorithm design. Consider the operational definition of neighborhood structure given in §6.1. When neighborhood structure is present, information about where

[14]Figures 18 and 19 were generated using a random sampling of 10,000 solutions in the restricted search space. When max_step = 1 in the index-rules encoding (Figure 19), the distribution has a bumpy appearance because the KK solution has only 98 neighbors.

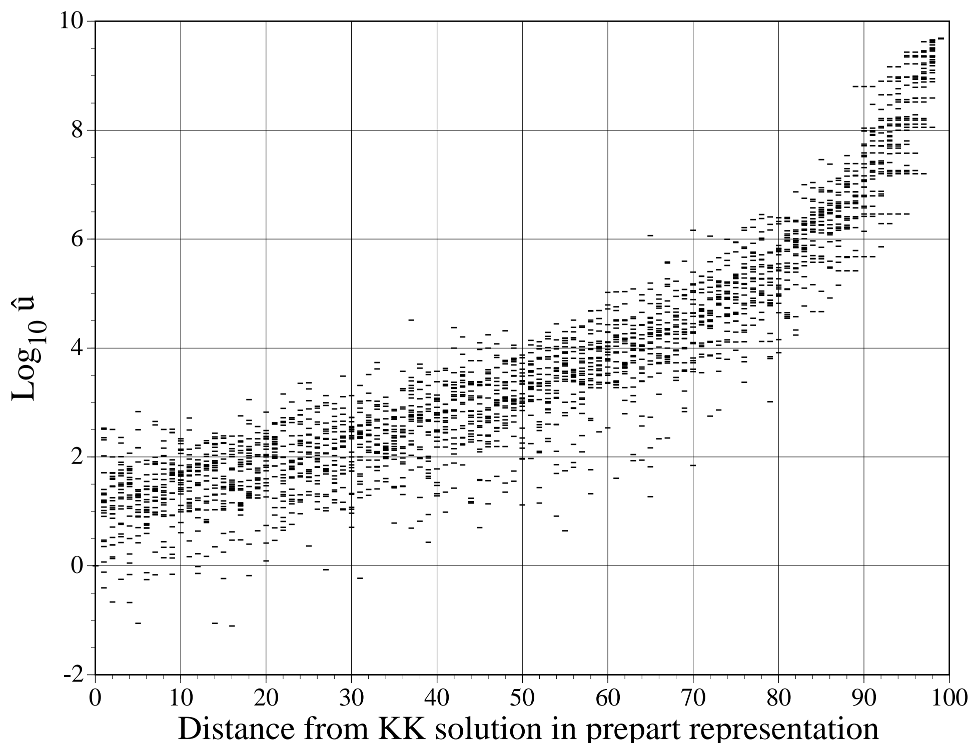


Figure 16: Gravitation in the prepartitioning encoding. The horizontal axis is the radial dimension of a ball, centered at the KK solution, that covers the entire solution space.

to find new low-residue solutions is gained as trial solutions are evaluated. When only gravitation is present, all such information is known at the start of the SO run; no new information is obtained by evaluating trial solutions. In such a case, one would expect to be able to adjust the *Randomize* method of an encoding to exploit gravitation so that RGT, which does not attempt to collect information about the search space as it evaluates candidate solutions, performs as well as perturbation-based forms of SO such as PHC, HC, and SA, whether seeded or not. Figure 20 demonstrates that this is indeed the case for the prepartitioning and index-rules encodings. In both cases, the $f_0(\xi)$ distribution is uniform in a ball surrounding the KK solution and zero elsewhere; in the absence of information about the structure of the search space other than what is presented in Figures 16 and 17, this probability distribution is optimal.[15] Specifically, the *Randomize* procedure in both cases is:

---

[15] To prove that the optimal distribution is uniform in a ball surrounding the KK solution and zero elsewhere when gravitation is the only relevant structure present, consider a non-uniform radial distribution, centered at the KK solution, whose density drops monotonically with distance from KK. This distribution cannot be optimal because shrinking the ball is guaranteed to improve the mean. The performance of a random or systematic search thus improves as the ball shrinks, as long as the number of solutions inside the ball exceeds the number of iterations one can afford to perform. The ball that we employ in our tests has the smallest integral radius that satisfies this condition.

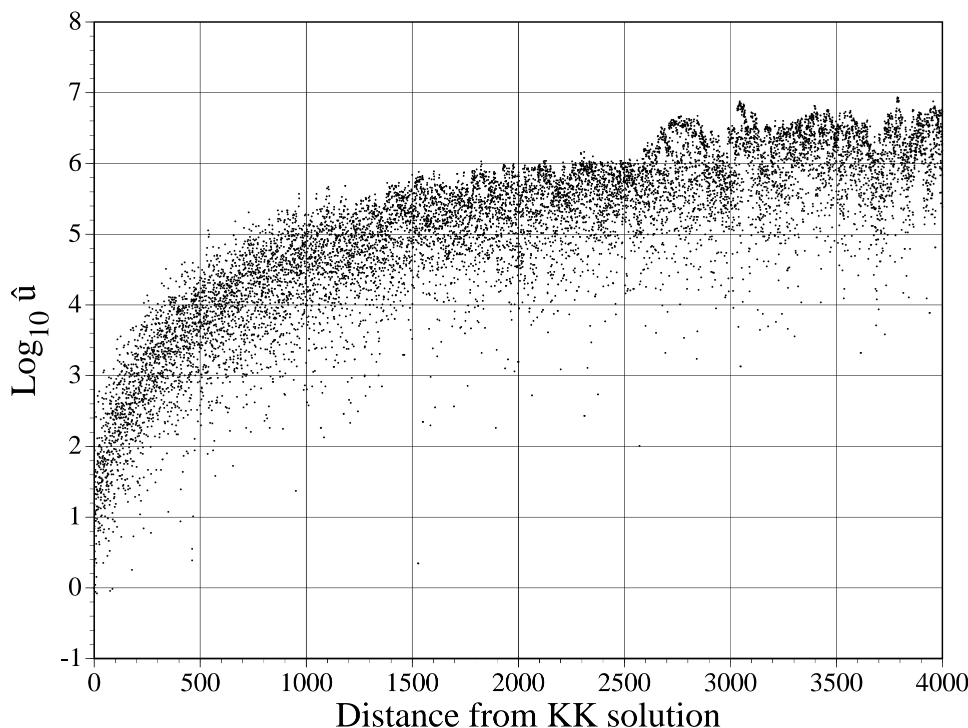


Figure 17: Gravitation in the index-rules encoding. The horizontal axis is the radial dimension of a ball, centered at the KK solution, that covers almost the entire solution space.

$\xi \leftarrow$ KK solution
**for** j = 1 to max_steps
  **if** rand$(0, n) > 0$ **then** $\xi \leftarrow$ *step_away($\xi$)*
**end for**

In the index-rules representation, *step_away($\xi$)* is $r_k \leftarrow r_k + 1$, where $k$ is an integer between 0 and $n-1$ inclusive, a given value of $k$ is chosen with probability proportional to $n-2-j$, subject to the requirement that $r_k + 1$ be a legal value. In the prepartitioning representation, *step_away($\xi$)* adds a new constraint of the form $s_k s_l = 1$. (With both representations, the high probability of executing *step_away($\xi$)* is designed to make $f_0(\xi)$ roughly uniform in the region covered by the search.) Similarly, one would expect a deterministic search that focuses on the region close to the KK solution to be competitive. The progress of a deterministic search in the index-rules representation is in accord with the prediction (Figure 20).[16]

HC, SA, and PHC are expected to be able to outperform RGT and deterministic searches in a search space that exhibits both gravitation and neighborhood structure. In such a case, the *Perturb* operation should be designed, as with the *Randomize* operation, to compensate for the negligible volume of the neighborhood of the greedy solution. Let $f'(\xi';\xi)$ be the conditional probability distribution from which *Perturb* draws solutions $\xi'$ given a solution $\xi$. We suggest that the $f'(\xi';\xi)$

[16]The search tested the KK solution, all solutions one or two steps away, and a small fraction of the solutions three steps away.

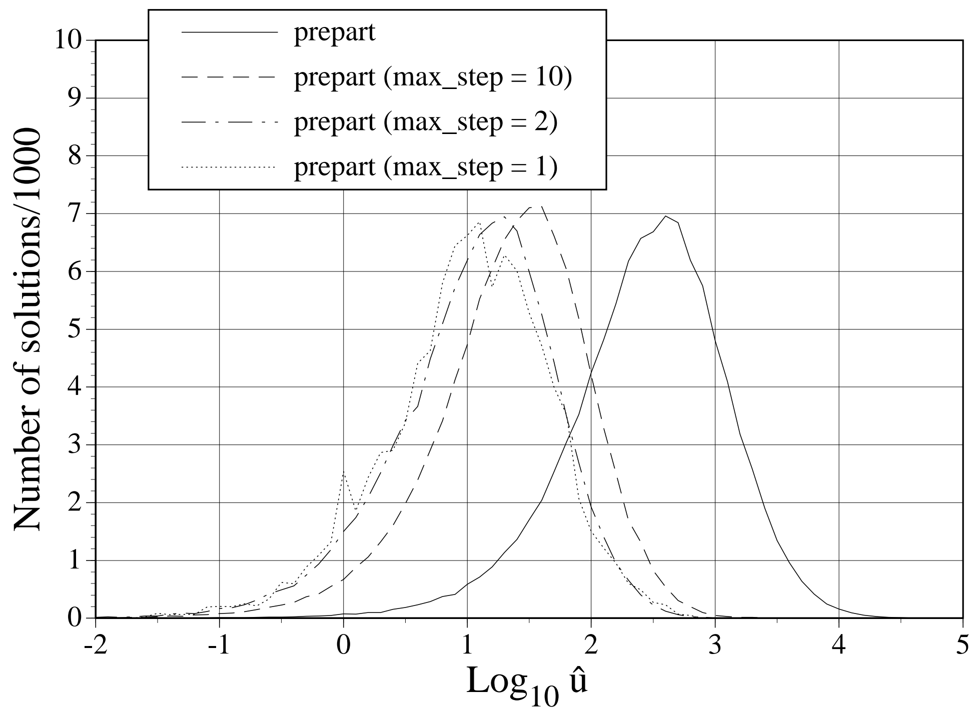


Figure 18: Effect on $h_0$ distribution of restricting the search space (prepartitioning encoding).

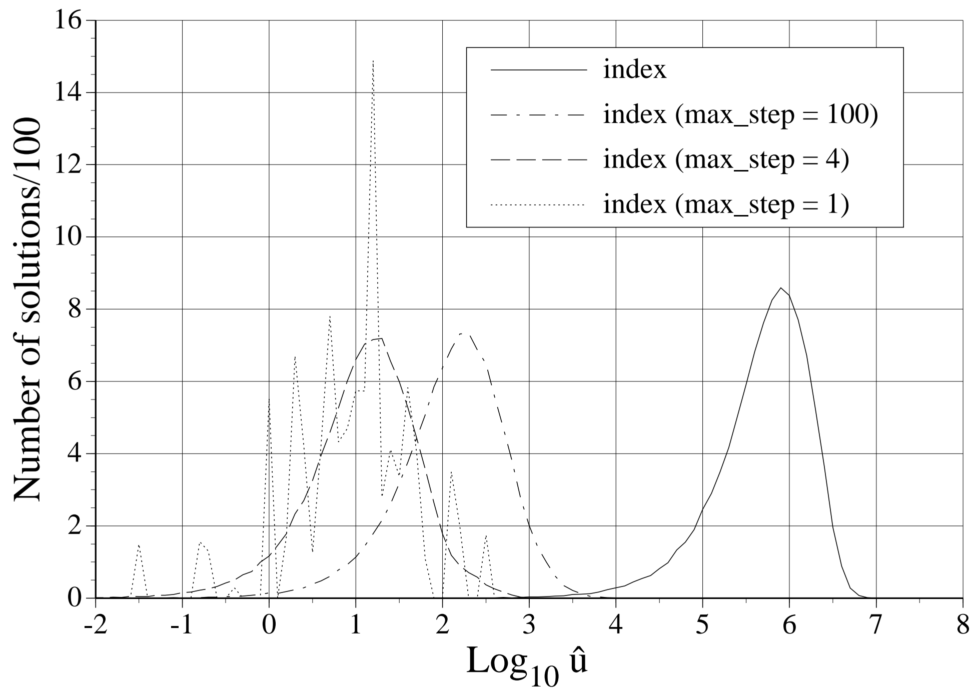


Figure 19: Effect on $h_0$ distribution of restricting the search space (index-rules encoding).

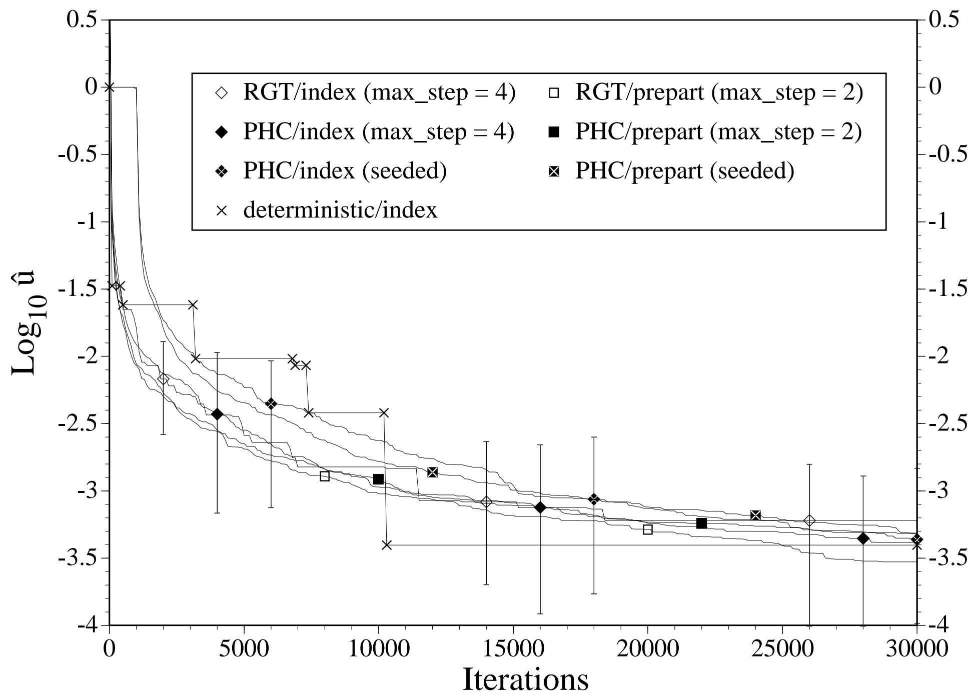


Figure 20: Performance of RGT in restricted search spaces.

distribution approximately obey the relation $\int_\Xi f'(\xi';\xi) f_0(\xi) d\xi = f_0(\xi')$, i.e., that the steady-state distribution that would be produced by repeated executions of *Perturb* be approximately equal to $f_0(\xi)$.

# 7 Conclusions

We have investigated the suitability of stochastic optimization for the NP-hard optimization problem NUMBER PARTITIONING. Our approach of analyzing the statistical properties of an existing optimization problem is complementary to previous lines of reasoning, notably that of Kauffman (1989), in which the ruggedness of the search space in question is under direct experimental control. We find that stochastic search engines previously thought ill-suited for NUMBER PARTITIONING can match the performance of the KK heuristic in the same amount of time, immediately begin improving upon KK with additional investment of time, and eventually better KK performance by several orders of magnitude. Moreover, the selection and subsequent tuning of the search engine—aspects of SO on whose adjustment great human effort is typically expended—are far less important to effective solution of NUMBER PARTITIONING problems than the choice of underlying problem representation (§5.4).

The most easily searched encodings of NUMBER PARTITIONING that we tested are based on parameterizations of existing deterministic heuristics. The proposition that stochastic search techniques be hybridized with greedy deterministic heuristics is not uncommon (Davis, 1985; Davis, 1991). We have focused particularly on integrating greedy deterministic heuristics into the pro-

cess of designing representations. In contrast with previous work, we have explicitly tested several hypotheses for the large difference in performance between the best and worst encodings. "Neighborhood structure," often believed to be the principal factor in determining the success of stochastic optimization, has little to do with the relative performance of the investigated encodings (§6.1). Instead, some encodings are better largely because they contain, on average, solutions with smaller residues (§6.2). Additional effects, present with the index-rules and prepartitioning encodings, can be accounted for in terms of "gravitation": the empirical observation that, in the index-rules and prepartitioning encodings, the residue of a candidate solution is strongly correlated with its distance from the KK solution (§6.3). Each of these two encodings organizes the candidate solution space into a single "bumpy funnel." Since the funnel is bumpy, i.e., the relationship between residue values and distance from the KK solution is only a statistical trend,[17] a stochastic search confined to the funnel tip readily outdoes the KK algorithm.

The principles by which the encodings were designed (§4.1) might easily be applied to other hard optimization problems for which deterministic heuristics currently exist.[18] Of the three principles, *parameterized arbitration*[19] is expected to be both the weakest and the least widely applicable since, by our definition, it refers to the parameterization of decisions that are otherwise undetermined in the given heuristic. Not all heuristics involve undetermined decisions; and when they do, it is because the decisions involved are "neutral" in the sense that there is no obvious reason to choose one alternative over any other.

Similarly, *parameterized-constraint* encodings can be designed only from greedy heuristics that are capable of respecting prespecified constraints. While this consideration is inconsequential in the case of heuristics for Number Partitioning, it may be a serious obstacle when approaching problems in which constraints are not trivial to apply. In favorable contrast with parameterized arbitration, however, there is reason to expect that a parameterized-constraint encoding will generally exhibit gravitation: in such an encoding, the average cost of solutions that specify a given number of constraints should vary gradually from the cost of the greedy solution (when no constraint is specified) to a value that is likely to exceed the mean of $h_0(u)$ (when the maximum number of constraints are specified).

Unlike the other two design principles, *parameterized greediness* involves an explicit, orderly relaxation of the greedy choices made in the given deterministic heuristic. This, we believe, is responsible for the strong gravitation observed in Figure 17. When strong gravitation is present, the *Randomize* procedure must be designed with care and can be used to advantage. We anticipate that parameterized greediness will usually be the easiest of the three design principles to apply, and often the most effective in terms of performance.

[17] If the funnel were smooth, i.e., the residue value were a single-valued, monotonically increasing function of a solution's distance from the KK solution, the KK solution would be optimal and Number Partitioning would lie in P.

[18] The XRLF algorithm for graph coloring (Johnson et al., 1991) is one important example of a parameterized greedy heuristic that has previously been described, though it uses a parameterization strategy that differs from the three strategies that we describe here.

[19] One important special case of parameterized arbitration might be called *parameterized initialization*. This design principle is applicable when the greedy heuristic in question is to perform local optimization from some starting point. The parameters would specify the starting point. In initial work with an NP-hard problem of great practical significance, Graph Partitioning, we have developed an encoding based on parameterized initialization (Marks et al., 1994).

Our results imply neither that we have found the best possible encodings of Number Partitioning for use with SO, nor that deterministic heuristics better than KK do not exist. Rather, if a deterministic heuristic better than KK is developed, the aforementioned design principles might be used to define encodings that outperform those presented here.

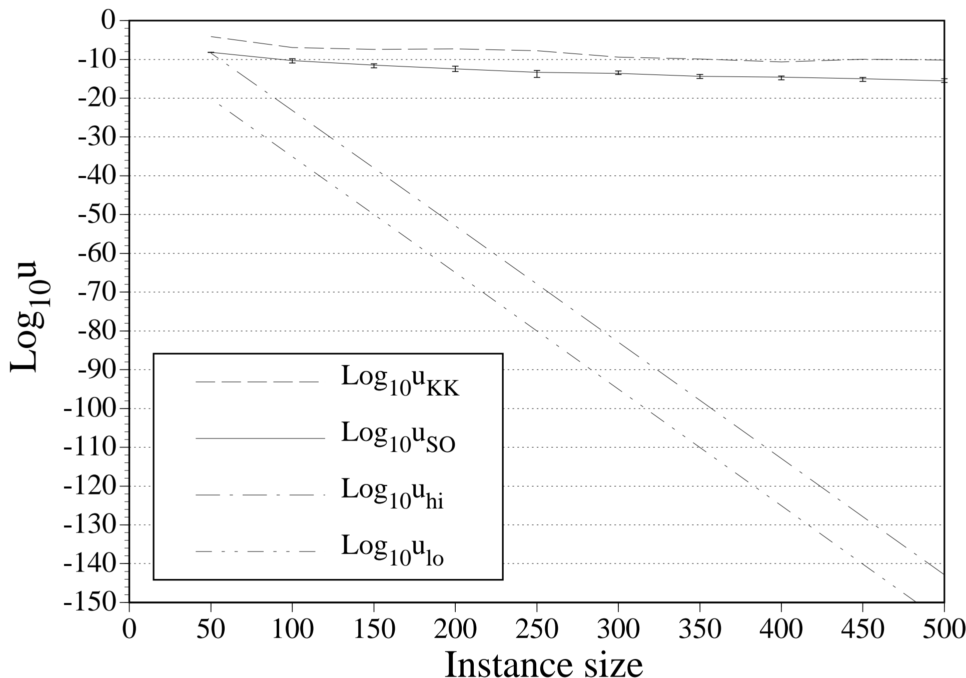


Figure 21: Performance of RGT/index (max_step = 4) with varying problem-instance sizes.

Does an SO variant that can find solutions better than KK by several orders of magnitude with a 100-element instance of Number Partitioning still find such solutions in a reasonable amount of time when used with larger problem instances? To help answer this question, we constructed instances of Number Partitioning that vary in size from 50 to 500 elements, in increments of 50. For each instance we ran a batch of 30 independent 50,000-iteration runs of RGT/index (max_step = 4), and recorded the final value of $\log_{10} u$. The results are plotted in Figure 21, with empirical results from running the KK algorithm, and upper and lower bounds for the optimal solution (see appendix; $p_{\rm lo} = p_{\rm hi} = 10^{-5}$). Although the solutions found by the best SO algorithms tested here are significantly better than those constructed by the KK algorithm, they are far from the median optimum (Figure 21).

## Acknowledgements

We thank David S. Johnson of AT&T Bell Labs for generously and promptly sharing his test instances. For stimulating discussions we thank members of the Harvard Animation/Optimization Group (especially Jon Christensen), the computer science department at the University of New Mexico, the Santa Fe Institute, and the Berkeley CAD group. The anonymous referees made numerous constructive suggestions. We thank Rebecca Hayes for feedback concerning our fig-

ures. JTN is grateful for a Graduate Fellowship from the Fannie and John Hertz Foundation. The research described in this paper was conducted mostly while JWM was at Digital Equipment Corporation's Cambridge Research Lab. This work was supported in part by the National Science Foundation, principally under grants IRI-9157996 and IRI-9350192 to Stuart Shieber, and by matching grants from Digital Equipment Corporation and Xerox Corporation. We thank the Free Software Foundation for making the GNU Multiple Precision package available.

## Appendix

For a given problem-instance size $n$ and probabilities $p_{\mathrm{lo}} \ll 1$ and $p_{\mathrm{hi}} \ll 1$, we aim to find some $u_{\mathrm{lo}}$ and $u_{\mathrm{hi}}$ such that for large $n$, $\Pr\{u_{\mathrm{opt}} \leq u_{\mathrm{lo}}\} \leq p_{\mathrm{lo}}$ and $\Pr\{u_{\mathrm{opt}} \geq u_{\mathrm{hi}}\} \leq p_{\mathrm{hi}}$, where $u_{\mathrm{opt}}$ is the optimum residue. Both are easily derived from asymptotic probability bounds derived by Karmarkar et al. (1986). The first of these bounds, $p_{\mathrm{lo}}$, is an upper bound on the probability that $u_{\mathrm{opt}} \leq u_{\mathrm{lo}}$. (It is assumed that the elements of $A$ are uniformly distributed between 0 and 1.)

$$p_{\mathrm{lo}} \equiv 2^n u_{\mathrm{lo}} \sqrt{\frac{24\pi}{n}} \quad .$$

The other bound, $p_{\mathrm{hi}}$, is an upper bound on the probability that $u'_{\mathrm{opt}} > u_{\mathrm{hi}}$, where $u'_{\mathrm{opt}}$ is the smallest attainable residue when the two partitions are constrained to contain the same number of elements ($\sum_{i=1}^{n} s_i = 0$):

$$\begin{aligned} p_{\mathrm{hi}} &\equiv \frac{1}{1+z} \quad , \\ \text{where} \qquad z &\equiv 2^n u_{\mathrm{hi}} \frac{\sqrt{12}}{\pi n} \quad . \end{aligned}$$

Clearly $p_{\mathrm{hi}}$ is also an upper bound on the probability that $u_{\mathrm{opt}} > u_{\mathrm{hi}}$, since $u_{\mathrm{opt}} \leq u'_{\mathrm{opt}}$.

To obtain given values of $p_{\mathrm{lo}}$ and $p_{\mathrm{hi}}$, we set $u_{\mathrm{lo}}$ and $u_{\mathrm{hi}}$ as follows:

$$\begin{aligned} u_{\mathrm{lo}} &= 2^{-n} p_{\mathrm{lo}} \sqrt{\frac{n}{24\pi}} \quad ; \\ u_{\mathrm{hi}} &= 2^{-n} \left(\frac{1-p_{\mathrm{hi}}}{p_{\mathrm{hi}}}\right) \left(\frac{\pi n}{\sqrt{12}}\right) \quad . \end{aligned}$$

## References

Davis, L. (1985). Job shop scheduling with genetic algorithms. In Grefenstette, J. J., editor, *Proceedings of An International Conference on Genetic Algorithms and Their Applications*, pages 136–140.

Davis, L. (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, NY.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, CA.

Goldberg, D. E. (1988). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI.

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. (1991). Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406.

Jones, D. R. and Beltramo, M. A. (1991). Solving partitioning problems with genetic algorithms. In Belew, R. K. and Booker, L. B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 442–449, San Mateo, CA. Morgan Kaufmann.

Karmarkar, N. and Karp, R. M. (1982). The differencing method of set partitioning. Report UCB/CSD 82/113, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA.

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

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. and Thatcher, J. W., editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, NY.

Kauffman, S. A. (1989). Adaptation on rugged fitness landscapes. In Stein, D. L., editor, *Lectures in the Sciences of Complexity*, pages 527–618. Addison-Wesley, Reading, MA.

Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671–680.

Manderick, B., de Weger, M., and Spiessens, P. (1991). The genetic algorithm and the structure of the fitness landscape. In Belew, R. K. and Booker, L. B., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 143–150, San Mateo, CA. Morgan Kaufmann.

Marks, J., Shieber, S., and Ngo, J. T. (1994). A stochastic search technique for graph bisection. Technical Report TR-94-18, Mitsubishi Electric Research Laboratories, Inc., Cambridge, MA.

Papoulis, A. (1990). *Probability and Statistics.* Prentice Hall, Englewood Cliffs, NJ.

Ruml, W. (1993). Stochastic approximation algorithms for number partitioning. Technical Report TR-17-93, Harvard University, Cambridge, MA.

Černy, V. (1985). Thermodynamical approach to the traveling salesman problem: An efficient simulation approach. *Journal of Optimization Theory and Applications*, 45(1):41–51.