Stochastic Approximation Algorithms for Number Partitioning

Wheeler Ruml

April 12, 1993

Abstract

This report summarizes research on algorithms for finding particularly good solutions to instances of the NP-complete number-partitioning problem.[1] Our approach is based on stochastic search algorithms, which iteratively improve randomly chosen initial solutions. Instead of searching the space of all 2^{n-1} possible partitionings, however, we use these algorithms to manipulate indirect encodings of candidate solutions. An encoded solution is evaluated by a decoder, which interprets the encoding as instructions for constructing a partitioning of a given problem instance. We present several different solution encodings, including bit strings, permutations, and rule sets, and describe decoding algorithms for them. Our empirical results show that many of these encodings restrict and reshape the solution space in ways that allow relatively generic search methods, such as hill climbing, simulated annealing, and the genetic algorithm, to find solutions that are often as good as those produced by the best known constructive heuristic, and in many cases far superior. For the algorithms and representations we consider, the choice of solution representation plays an even greater role in determining performance than the choice of search algorithm.

[1] This work was undertaken with help from Stuart Shieber, Joe Marks, and Tom Ngo.

Contents

1 Introduction **1**

2 Number Partitioning **2**
2.1 The Problem . . . 2
2.2 Previous Work . . . 3
2.2.1 The Karmarkar-Karp Algorithm . . . 3
2.2.2 Comparisons with Simulated Annealing . . . 5
2.2.3 A Genetic Algorithm for Number Partitioning . . . 6

3 Algorithms **6**
3.1 Random Generate-and-Test . . . 7
3.2 Local Optimization . . . 7
3.3 Simulated Annealing . . . 8
3.4 A Genetic Algorithm . . . 9
3.4.1 Implementation . . . 10
3.4.2 Schema Processing . . . 12
3.5 Parallel Local Optimization . . . 12
3.6 Mixed Algorithms . . . 12

4 Implementation **13**

5 Direct Representation **14**
5.1 Operators . . . 15
5.2 Results . . . 17

6 Permuted Lists with Decoders **21**
6.1 Operators . . . 21
6.2 The Splitting Decoder . . . 23
6.2.1 Results . . . 24
6.3 The Alternating Decoder . . . 27
6.3.1 Results . . . 27
6.4 The Number-Based Splitting Decoder . . . 27
6.4.1 Results . . . 30
6.5 The Greedy Splitting Decoder . . . 33
6.5.1 Results . . . 33
6.6 The Greedy Decoder . . . 36
6.6.1 Results . . . 36
6.6.2 Long Runs . . . 39

6.6.3 Large Instances . . . 40
6.7 Summary of Results . . . 40
6.7.1 Representation Spaces . . . 40
6.7.2 Random Generate-and-Test . . . 42
6.7.3 Local Optimization . . . 44
6.7.4 Parallel Local Optimization . . . 45
6.7.5 Simulated Annealing . . . 46
6.7.6 The Genetic Algorithm . . . 47
6.8 Seeded Algorithms . . . 47
6.8.1 Local Optimization . . . 47
6.8.2 Simulated Annealing . . . 49

7 Difference Rules 51
7.1 Index-Based Difference Rules . . . 51
7.1.1 Operators . . . 51
7.1.2 Results . . . 52
7.1.3 Seeding . . . 54
7.2 Weighted Index-based Difference Rules . . . 56
7.2.1 Results . . . 56
7.2.2 Seeding . . . 56
7.3 Single-Index Difference Rules . . . 59
7.3.1 Results . . . 59
7.3.2 Seeding . . . 59

8 Prepartitioning 59
8.1 Operators . . . 62
8.2 Results . . . 62
8.3 Seeding . . . 62
8.4 Summary of Results . . . 65
8.4.1 Representation Spaces . . . 65
8.4.2 Single Algorithms . . . 65
8.4.3 Seeded Algorithms . . . 70
8.5 Overall Summary . . . 71

9 Conclusion 71

Acknowledgements 73

References 74

1 Introduction

This report describes original research on techniques for finding good solutions to instances of the NP-complete *number-partitioning* problem.[2] In this combinatorial optimization problem, one is given a set of numbers, and asked to partition them into two sets, such that the sums of the numbers in each set are as close as possible to equal. Although there already exists an efficient approximation algorithm, due to Karmarkar and Karp, for constructing solutions that are close to optimal, it is non-stochastic and always yields the same solution for any given problem instance. We have investigated the use of stochastic algorithms, which attempt to find a good solution to a given problem instance by using a search procedure that depends, to a limited extent, on random numbers. Our hope is that, while it may not always find a good solution, a stochastic algorithm may be run many times using different random seed values, possibly yielding a better solution on one of its runs than one could obtain by the non-stochastic Karmarkar-Karp method.

Stochastic algorithms are typically based on an exploration of the solution space rather than the direct construction of a solution. Unfortunately, the size of the solution space for the number-partitioning problem is exponential in the size of the problem instance, and very few of the possible solutions are good ones. Directly searching the solution space is therefore quite inefficient.

At the center of our approach is the idea of an *encoding* of a solution to a given problem instance (see figure 1 for a schematic diagram). Instead of manipulating partitions directly, many of the algorithms we have tested manipulate an encoding structure, such as a permutation of $(1, \ldots, n)$. An encoding can be interpreted by a decoding algorithm to yield a solution for the given problem instance. A permutation, for instance, can serve as instructions to a greedy decoding algorithm, which would construct a partitioning by considering instance numbers in the order specified by the permutation, and adding each to the partition with the currently lowest sum. In this way, every permutation encoding specifies a particular partitioning. An evaluation of that partitioning is then used by the search algorithm as a score for the original encoding structure. Guided by these scores, the algorithm explores the space of all encodings, which may differ substantially

[2] This work was undertaken with help from Stuart Shieber, Joe Marks, and Tom Ngo; see page 73.

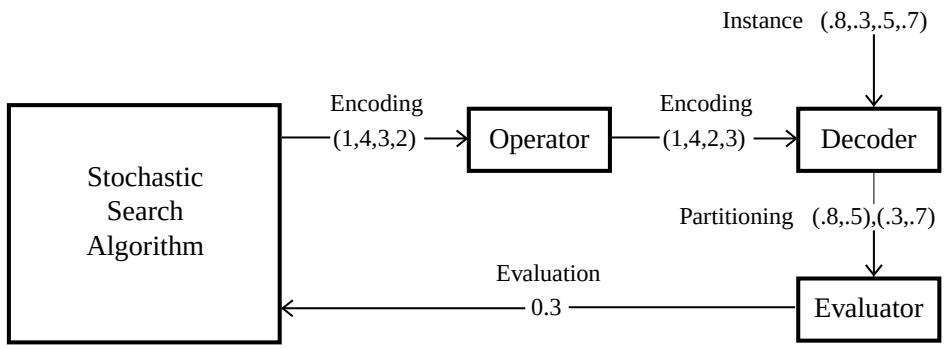

Figure 1: A schematic diagram of our approach to stochastic optimization. A search algorithm manipulates encodings via operators, guided only by evaluations of the partitionings made by decoding the encodings.

in size and structure from the space of all partitionings. By using the encoding structures to construct an especially rich and smoothly structured search space, we hope to enable stochastic algorithms to find solutions that are competitive with those produced by the specialized Karmarkar-Karp algorithm without groping blindly through the desert of all 2^{n-1} possible partitionings.

We have constructed six different encoding structures for the number-partitioning problem, and tested each with at least five different stochastic search algorithms. Note that these algorithms, unlike the Karmarkar-Karp algorithm, are general-purpose methods which explore a solution space, guided only by the quality of the solutions that have been seen so far. As such, they are readily adaptable to different problems.

Our results affirm that algorithm performance is fundamentally based on the representation space being explored, and they show that a good choice of encoding representation can raise the performance of a stochastic algorithm up to or beyond the level of the Karmarkar-Karp algorithm.

2 Number Partitioning

2.1 The Problem

Precisely, an instance of number-partitioning consists in a finite set A and a magnitude $m(a) \in [0, 1]$ for each $a \in A$. The optimal solution s to a given

problem instance (A, m) is a subset $A' \subseteq A$ such that the difference

$$\text{cost}(s) = |\sum_{a \in A'} m(a) - \sum_{a \in A - A'} m(a)|$$

is as small as possible.[3] Karmarkar et al. [10] have shown that the median of the distribution of expected costs of optimal solutions shrinks in $O(\sqrt{n}/2^n)$.

Number-partitioning can be understood as the optimization problem implied by the classic NP-complete PARTITION decision problem (SP12, p. 223 in Garey and Johnson [2]). PARTITION merely asks if it is possible, given a problem instance (A, m), to form a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} m(a) = \sum_{a \in A - A'} m(a).$$

It was shown to be NP-complete by transformation from three-dimensional matching (3DM) in Karp's seminal 1972 paper on NP-completeness [11], and has become popular as the target of transformations from problems in network design, storage and retrieval, scheduling, and mathematical programming (see table 1 for examples). Garey and Johnson rank PARTITION among the six quintessential NP-complete problems. They also explain how the decision problem can be solved in "pseudo-polynomial time" by dynamic programming (that is, in polynomial time if the input values are bounded). Unfortunately, we do not know of any such algorithm for finding optimal solutions.

Since many problems reduce to PARTITION, a fast approximation algorithm for it has many applications. Besides obvious problems in scheduling and mathematical programming, partitioning finds practical application in cryptography [13]. With a problem instance serving as a lock, a perfect partitioning (i.e., $\text{cost}(s) = 0$) is an easily verifiable key.

2.2 Previous Work

2.2.1 The Karmarkar-Karp Algorithm

The Karmarkar-Karp algorithm [9] is an efficient approximation algorithm for constructing good solutions to instances of number-partitioning. The algorithm, also known as the "difference method," works by constructing a tree, and then coloring it:

[3] Although we have specified the task of dividing the instance numbers into two partitions, all of the algorithms and representations that we consider are easily applied to the problem of partitioning into an arbitrary number of subsets.

Problem	Code
Bin Packing	SR1
Expected Retrieval Cost	SR4
Knapsack	MP9
Continuous Multiple Choice Knapsack	MP11
Subset Sum	SP13
Minimum Sum of Squares	SP19
K^{th} Largest M-tuple	SP21
Shortest Weight-Constrained Path	ND30
Sequencing to Minimize Tardy Task Weight	SS3
Multiprocessor Scheduling	SS8
Scheduling to Minimize Weighted Completion Time	SS13
Open-Shop Scheduling	SS14
Production Planning	SS21
Randomization Test for Matched Pairs	MS10

Table 1: Important NP-complete decision problems which are reducible to PARTITION and which have corresponding optimization problems which are reducible to number-partitioning. The codes identify the problems' entries in Appendix A of Garey and Johnson.

1. (Initialization) Each instance number is assigned a node. Each node is assigned the value of its corresponding instance number, and declared to be "live."

2. (Build tree) While more than one node is "live," repeat the following:

   (a) (Pick nodes) Pick the live node u with the greatest value, and the live node v with the second greatest value. Note that this step resembles the heart of a "greedy" algorithm.

   (b) (Connect nodes) Construct an edge between u and v. This represents the decision to put the two instance numbers in different partitions.

   (c) (Take difference) Declare v "dead," and subtract the value of v from the value of u. This new value for u represents the difference between the partitions specified by the attached tree of dead nodes.

3. (Make partition) The last live node is the root of a tree. Two-color this tree to create the partitions. The value of the last live node is the difference between the partitions.

The algorithm can be implemented to run in $O(n \log n)$ time, and a simpler variant has been shown to construct solutions with an expected difference of $O(1/n^{\alpha \log n})$, $\alpha > 0$. Although this can be much greater than the expected optimum ($O(\sqrt{n}/2^n)$), it is much less than the expected difference of results of any comparable heuristic algorithm.

2.2.2 Comparisons with Simulated Annealing

The Karmarkar-Karp algorithm seems formidable competition for a stochastic algorithm. In their extensive empirical work on simulated annealing, Johnson et al. [5, 6] compare the performance of a stochastic simulated annealing algorithm (see section 3.3) to that of the Karmarkar-Karp algorithm. Johnson et al. choose number-partitioning as a hard test problem for simulated annealing because of the great range of possible solution values, the scarcity of good solutions in the space of all partitionings, and the high performance of the competing Karmarkar-Karp algorithm.

Using a well-designed simulated annealer searching through the space of all 2^{n-1} possible partitionings, Johnson et al. show that both local optimization (see section 3.2) and simulated annealing take about 50,000 times

as long as the Karmarkar-Karp algorithm to find a comparable solution to a given 100-element instance, and much longer for larger instances.

Johnson et al. conclude with the observation that the size and structure of the representation space is the critical issue that limits the performance of the stochastic algorithm:

> The major challenge is that of devising a suitable and effective neighborhood structure. ... There remains the question of whether some other neighborhood structure for the problem ... might prove more amenable to annealing. ([6], pp. 400, 405)

It is this question of neighborhood structure that we have investigated.

2.2.3 A Genetic Algorithm for Number Partitioning

Encodings for problem solutions and operations on these encodings form the central focus of research on genetic algorithms, a popular stochastic optimization method (see section 3.4). Jones and Beltramo of General Motors have compared two encodings for number-partitioning and several different operators on these encodings [7]. Using a test problem of length thirty-four partitioned into ten disjoint subsets, one of their representations (permuted lists with a greedy decoder and pmx crossover, see section 6.6) enabled a genetic algorithm to approach the known optimum quickly. When manipulating partitions directly, however, their genetic algorithm converged to very poor solutions.

Although their results are encouraging, Jones and Beltramo do not compare the performance of their genetic algorithm against other competitive search methods, nor do they test its ability on larger problem instances.

3 Algorithms

In order to compare different encoding representations for number-partitioning without bias in favor of one particular stochastic algorithm, we have implemented several. Each algorithm depends upon the notion of moving through a space of solutions, usually starting from a random initial solution and proceeding via an operator that yields a random neighbor of a given solution. Note that since our algorithms manipulate an indirect encoding, an extra decoding step is necessary to form a solution to the given problem instance. When we refer to the cost of an encoded solution, we actually mean the difference between the partitions in the decoded solution. This usage is

sound because all the decoders we have investigated are non-stochastic and yield a unique solution for any given encoding.

3.1 Random Generate-and-Test

The simplest stochastic optimization algorithm is to generate a specified number of randomly chosen solutions and return the best one. We used this algorithm as a benchmark for others. It also gives a good indication of the density of good solutions in the space induced by a given representation.

3.2 Local Optimization

In local optimization, also known as 'hill climbing,' one starts with a random solution and attempts to improve upon it by looking at its immediate neighbors. Our local optimizer can be sketched as:

1. (Initialize) Pick a random solution s.
2. (Loop) For a specified number of iterations, repeat the following:
   (a) (Get neighbor) Apply an operator to s, yielding one of its neighbors, s'.
   (b) (Test and replace) If $\text{cost}(s') < \text{cost}(s)$ (i.e., s' represents a better solution than s), replace s with s'.
3. (End) Return the current solution s.

Note that this differs from some implementations of 'gradient descent' in which all neighbors s' of s are tested, and s is replaced by its best neighbor. Due to the large number of neighbors induced by some of our representations and operators, our algorithm does not follow the 'steepest' route to the optimum, but rather takes the first improvement it can find.

While performing well for many problems in which the evaluation of solutions follows a monotonic path from any point to the optimum, local optimization does notoriously badly on problems that have many local optima. If all neighboring solutions have a higher cost, local optimization will not find any other optimum, even if it is nearby and substantially better.

3.3 Simulated Annealing

Simulated annealing attempts to circumvent the poor behavior of local optimization by allowing occasional moves to neighboring solutions that are worse than the current one. The algorithm takes its name from an analogy with a process in physical chemistry, in which a liquid is more likely to crystallize in its minimum energy configuration if it is cooled, or 'annealed,' very slowly [8]. The cost of a solution is considered to be analogous to the energy level of a particular configuration, and a variable t, representing temperature, controls how likely a move to a worse solution will be. Typically, annealing algorithms follow a 'schedule' of decreasing temperatures. At first, almost any move will be tolerated, but as t decreases, only moves to neighboring solutions that are better or only slightly worse will be allowed. Some implementations of simulated annealing exhibit values for t at which 'phase transitions' occur, when moves to positively bad solutions are prohibited but relatively small uphill climbs are still forgiven, thus promoting rapid improvement while giving the search resistance to local minima.

There are many variations of this basic annealing method. Our implementation follows that of Johnson et al., and depends mainly on three parameters: *init-prob*, the desired probability of accepting a random move at the starting temperature, *temp-length*, the number of moves to attempt at any given temperature, and *temp-factor*, which controls how fast the temperature should be lowered. The algorithm proceeds like this:

1. (Initialization) Pick a random solution s.
2. (Find starting t) Set t to a value that yields acceptance of approximately *init-prob*% of the neighbors of a randomly chosen solution. (This step can easily be performed separately, and is more accurate if computed using several random starting solutions.)
3. (Loop) While *frozen* $<$ *max-frozen*, repeat the following:
   (a) (Loop at this t) Repeat the following *temp-length* times:
      i. (Find neighbor) Apply a specified operator to s, yielding s'.
      ii. (Test) If $\text{cost}(s') < \text{cost}(s)$, replace s with s' and reset *frozen* to zero.
      iii. (Stochastic move) If $\text{cost}(s') > \text{cost}(s)$, accept it anyway with probability $e^{-\Delta/t}$ where Δ is $\text{cost}(s') - \text{cost}(s)$.
   (b) (Decrease t) Set $t = \textit{temp-factor} \cdot t$.

(c) (Check progress) If fewer than *min-percent*% of moves attempted at this temperature have been accepted, increment *frozen*.

4. (End) Return the best solution seen during the run (which may be different from the solution s at convergence).

During our experiments, the variable *max-frozen* was not used. Instead, the algorithm was run for a specified number of iterations. For the representations that we have used, those that allow a low *init-prob* setting of 20% usually perform better at that low starting temperature than at Johnson et al.'s recommended setting of 50% (for some representations, including the one used by Johnson et al., the range of costs of constructible solutions is so great that the test in step ii succeeds half of the time). Step 2 was performed beforehand, using a converging binary search evaluating the percentage of moves accepted at each temperature during 300 iterations of the inner loop (i–iii) for 25 randomly chosen solutions. We used a *temp-factor* of 0.9 and a *temp-length* of between two and sixteen times the number n of elements to be partitioned.

Simulated annealing has been shown to offer qualitatively better performance than local optimization for many problems [5]. It has been proven that, given an annealing schedule that calls for lowering the temperature extremely slowly, simulated annealing will find the globally optimum solution. Unfortunately, such a slow schedule can mean taking longer than a direct branch-and-bound computation of the optimum [12]. To speed up simulated annealing, Johnson et al. recommend spending less time at each temperature (by lowering *temp-length*), rather than lowering the temperature faster (decreasing *temp-factor*).

3.4 A Genetic Algorithm

Local optimization and simulated annealing both start with a single random solution and attempt to improve upon it by looking at its neighbors. In contrast, a genetic algorithm [4, 3] considers many solutions at the same time, and allows the construction not only of new solutions that are neighbors of a particular current solution, but also of new solutions that incorporate information from two different current solutions. By analogy to genetics, 'neighbor' operators are referred to as 'mutators,' and operators that take information from two solutions are called 'recombination' operators, or 'crossovers.' A typical crossover operator will swap portions of two encodings, and return the two new hybrids.

3.4.1 Implementation

The performance of a genetic algorithm depends on many small design decisions, including the process of selecting solutions to operate on and the policy for deletion of poor solutions. Our genetic algorithm proceeds like this:

1. (Initialize) Initialize the population set P to contain a specified number of random solutions.

2. (Loop) For a given number of iterations, repeat:

   (a) (Select operator) Choose an operator o from a specified non-empty set O.

   (b) ('Crossover') If o takes two arguments, do the following:

      i. (Select 'parents') Select two solutions s_1 and s_2 from P. The probability of selecting a particular solution s_i is inversely related to $\text{cost}(s_i)$.

      ii. (Apply operator) Apply o to s_1 and s_2, producing new solutions s_1' and s_2'.

   (c) ('Mutation') If the operator o instead takes only one argument, do the following:

      i. (Select parent) Select a solution s from P. As with crossover, the probability of selecting a particular solution is inversely related to its cost.

      ii. (Apply operator) Apply o to s, producing a new solution s'.

   (d) (Add solutions) For each new solution, if its cost is less than the cost of the worst member of P, delete that worst member and add that new solution to P.

3. (End) Return the best solution in P.

This is a 'steady-state' genetic algorithm, which guarantees that only the worst solutions will be replaced, as opposed to a 'generational' algorithm, in which all solutions are replaced with every iteration.

In our implementation, each operator has an associated probability ρ_o of being selected on a given iteration, and each selection of an operator is made independently. This differs from some implementations, in which a mutation can only be performed after a crossover. We also attempt to insert

both solutions that result from a crossover into the population, rather than picking one arbitrarily. This complicates the relationship between iterations of the algorithm and the number of solutions evaluated, since two evaluations are performed during an iteration in which the selected operator o is a crossover, but it assures that all good solutions produced by a crossover have the opportunity to be added to the population.

Following Davis [1], we use a linear ranking system for parent selection. Solutions are ranked by cost, then chosen with a probability inversely related to their rank. Even if the best solution is much better than the second- and third-best solutions, it will only be given the same relative preference that the second-best solution enjoys over the third-best. By selecting solutions according to their rank, rather than directly by their cost, we hope to avoid uniquely qualified solutions from being chosen too often at the start of a run, and to avoid all solutions being weighted evenly when the population has become uniform towards the end of a run. The selectivity of this parent selection process is controlled in our algorithm by the parameter *rank-factor*, which is zero when all solutions are equally likely to be chosen and one when the best solution is twice as likely to be chosen as the median solution. For a solution $s \in P$, where the index of s in a sorted list of all solutions in P is i, $|P| = p$, and *rank-factor* $= r$, we take the probability ρ_s of choosing s as:

$$\rho_s = \frac{-2r}{p(p-1)} i + \frac{r+1}{p}$$

To implement this distribution, we need a function to map a random number in $[0, 1]$ to the index of the proper solution. By integrating the previous equation, we obtain an intermediate formula, the inverse of the one we seek, which assigns a point n in the interval $[0, 1]$ to each i, $0 \leq i < p$:

$$n = \frac{i^2 r}{p(1-p)} + \frac{i(r+1)}{p}$$

Solving for i, we are left with a formula which maps a randomly chosen number $0 \leq n < 1$ onto the index of the proper solution in the sorted list:

$$i = \frac{p - 1 - r + pr - \sqrt{p-1}\sqrt{p - 1 - 2r + 2rp - r^2 + pr^2 - 4prn}}{2r}$$

Of course, i must then be truncated to an integer.

To summarize, the parameters of our genetic algorithm include: the size of the population P, the set of operators O, the probabilities ρ_o of selecting

each operator $o \in O$, and the selectivity of parent selection, *rank-factor*. The results reported here used a population size of 1000 and a *rank-factor* of 0.8; empirical testing has shown that these settings achieve a good balance of population diversity and exploitation of good solutions.

3.4.2 Schema Processing

The notion of the neighborhood space induced around a particular solution by the action of a particular operator becomes more complicated when one allows crossovers between two solutions; it is not enough to say that $\binom{n}{2} = n(n-1)/2$ different crossovers can occur. Analysis of a genetic algorithm is usually expressed in its ability to combine pieces of encoding, or 'schema,' from two good solutions to form a new, better solution (see [4] for more detail). Schema processing plays a key role in operator design; if the operator does not combine useful pieces of solutions, but instead adjoins portions of the encoding that will not contribute toward lowering its cost, the genetic algorithm will not be able to construct good solutions.

This has been acknowledged as a fundamental weakness of the genetic algorithm. While it successfully avoids the local optima which entrap local optimization, the algorithm cannot construct solutions that require combining schema that do not perform well separately. Techniques for solving such 'deceptive' problems are an active area of research.

3.5 Parallel Local Optimization

In order to assess the effectiveness of the crossover operations so fundamental to the operation of a genetic algorithm, we also ran the genetic algorithm without any crossover operators at all. This is very different from local optimization, however, because of the genetic algorithm's process of selecting promising solutions from a population structure. In this way, a castrated genetic algorithm, or parallel local optimizer, can remember multiple promising solutions and balance exploitation of the best solutions currently known with exploratory operations around other promising solutions.

3.6 Mixed Algorithms

Although the main focus of our work has been to assess the effect of representation on particular algorithms per se, we also experimented with combining two or more of our algorithms in order to find the best possible solutions. We have experimented with using local optimization as a post-processing step

on results obtained by the random generate-and-test method and the genetic algorithm. We have also used the solution constructed by the Karmarkar-Karp heuristic as a starting point for other algorithms.

Because there is often a many-to-one correspondence between encoded solutions and partitionings of an instance, results from these mixed algorithms can depend on the particular solution in the second algorithm's representation that is chosen to represent the partitioning found by the first algorithm. We have implemented the most straightforward transformations; we have not guaranteed that the transformed partition will be encoded in the most advantageous way for the new algorithm.

4 Implementation

These algorithms were each implemented in ANSI C in a parameterized manner that separated the details of any particular solution representation from the essential workings of the algorithms. They were run on a DECstation 5000/33 (MIPS CPU) under ULTRIX 4.2a (GNU C compiler) and on a Sun 4m 670 (four Sparc2 CPUs) under SunOS 4.1.3 (Sun ANSI C compiler). All reported times are scaled to indicate elapsed user-level CPU time on the DECstation.

Unless otherwise indicated, all results are the geometric mean of 100 runs of each algorithm. Where $\mathrm{cost}(s_i)$ refers to score of the final solution of run i in a batch of m runs, the geometric mean was computed as:

$$10^{\bar{l}} \text{ where } \bar{l} = (1/m) \sum_{i=1}^{m} \log_{10} \mathrm{cost}(s_i)$$

Since results tend asymptotically toward zero, this value more closely reflects their distribution than an ordinary arithmetic mean would.

We used test cases with 100, 200, and 500 elements chosen uniformly from [0,1). Results are for the 100-element problem unless noted otherwise. In each instance, numbers were specified to at least five more decimal places than were necessary to represent the expected difference of an optimal solution to a problem of that size (see table 2). Since solutions are evaluated by summing each partition and subtracting the two sums, it is important that these arithmetic operations be performed with full accuracy. Any round-off error in the addition will change the computed difference between partitions, possibly even creating or denying a perfect partitioning. To avoid this

n	digits	exp'd cost(s_{opt})	exp'd cost(s_{kk})	cost(s_{kk})	cost(s_{best})
100	36	7.9e-30	1.0e-4	1.1e-8	1.3e-14
200	65	8.8e-60	5.1e-6	1.3e-10	2.2e-16
500	156	6.8e-150	5.2e-8	6.5e-12	5.1e-19
100†	36	7.9e-30	1.0e-4	1.5e-7	1.1e-12
200†	36	0	5.1e-6	1.7e-9	1.1e-14
500†	36	0	5.2e-8	1.5e-12	3.8e-17

Table 2: Attributes of the test instances: length of instance, number of digits specified, expected optimum, expected Karmarkar-Karp solution (α assumed to be one), actual Karmarkar-Karp solution, best known solution (found using the algorithms of section 7.1.3). Instances marked '†' correspond to test cases from Johnson et al.

problem, all algorithms were implemented using arbitrary precision integer arithmetic (the GNU MP library).

Although we attempted to benchmark our algorithms using the test cases of Johnson et al. [6], our results are incomparable due to the lack of arithmetic error in our computations.

5 Direct Representation

The first representation to consider is the most straightforward one: the direct encoding of a partitioning. A solution consists of a list of partition labels, with the label at position i specifying the partition into which the ith instance number is to be placed. Since we consider partitioning a given instance into two partitions, this direct representation ranges over $(0,1)^n$ yielding 2^n possible solutions. This is twice the number of possible partitionings because we allow the same partitioning to be labelled in two ways. Enforcing a regular labelling would not restrict the possible search space, is expensive to calculate, and has unpleasant effects with mutation operators (see Jones and Beltramo [7] for an example and empirical results).

5.1 Operators

We have defined several operators for direct partition encodings. We started by implementing the "SW_k" operators from Johnson et al.:

one-move Creates a new solution by moving one random instance number to the other partition. The n neighbors s' of a solution s are all solutions such that if s defines a partitioning into disjoint sets A_1 and A_2 and s' defines sets B_1 and B_2, then A_1 and B_1 differ by one element, i.e., $|A_1 - B_1| + |B_1 - A_1| = 1$.

The average move cost ($|\text{cost}(s) - \text{cost}(s')|$) is twice the average element value, namely $2(0.5) = 1$, and the smallest possible move cost would be the value of the least element, or $1/n$ in the average case.

two-move Moves one element to a new partition and a second distinct element to a random partition (possibly the one it is already in). Using our previous terminology, $|A_1 - B_1| + |B_1 - A_1| \leq 2$. Thus the neighborhood of a given solution is of size $n + n(n-1)/2 = (n^2 + n)/2$, much larger than for the one-move operator. The smallest possible move would be the difference between the two smallest elements, or $(1/n) - 1/(n-1) = 1/(n^2 - n) \approx 1/n^2$ on average.

We also considered additional operators, including two crossovers:

two-always-move Always moves two elements to new partitions. Induces a neighborhood of size n^2, with a minimum move cost of $1/n^2$.

one-swap Swaps the partition labels of two elements, i.e, $|A_1 - B_1| = |B_1 - A_1| \leq 1$. Note that the two elements are not required to be in different partitions. Neighborhood size n^2 and minimum move cost $1/n^2$.

one-always-swap Swaps two elements between distinct partitions, such that $|A_1 - B_1| = |B_1 - A_1| = 1$. Neighborhood size n^2 and minimum move cost $1/n^2$.

two-point crossover Given two solutions s_1 and s_2, swaps the partitioning specified for all instance numbers c_1 through c_2, where $1 \leq c_1, c_2 \leq n$ are chosen at random. The operator creates two new solutions s'_1 and s'_2 such that $s'_i = s_i$ except at positions between the two 'crossover points' c_1 and c_2. At those positions, the partitioning specified by s'_1 is that of s_2, and similarly for s'_2 (see table 3 for an example).

s_1	=	0	1	0	1	1	0
s_2	=	1	0	1	1	0	1
s'_1	=	0	0	1	1	1	0
s'_2	=	1	1	0	1	0	1

Table 3: An example of two-point crossover between partitionings. If '|' represents each crossover point, a two-point crossover between solutions s_1 and s_2 will produce solutions s'_1 and s'_2.

s_1	=	0	1	0	1	1	0
s_2	=	1	0	1	1	0	1
r	=	0	1	1	0	1	0
s'_1	=	0	0	1	1	0	0
s'_2	=	1	1	0	1	1	1

Table 4: An example of uniform crossover between partitionings. Using the random numbers r_i, where a value of one specifies a swap, a uniform crossover between solutions s_1 and s_2 will produce solutions s'_1 and s'_2.

Under two-point crossover, short building blocks of partition specifications that are clumped together in the encoding are favored over longer schema that are less likely to be completely reproduced. This is desirable when the encoding has an implicit structure in which related parts of a solution are close to one another. Because of this, we would expect an instance with unordered instance numbers not to benefit from two-point crossover, while an instance in which numbers were indexed in sorted order should do better.

uniform crossover Instead of using crossover points, uniform crossover makes the decision to swap independently for each instance number. In effect, for each instance number, the partition specified by s'_1 is chosen randomly between the partitions specified by s_1 and s_2 (see table 4 for an example).

The intent is to allow the reproduction of schema when the relationships between positions in the representation are not known [14]. In such a case, there is no advantage in encouraging the propagation of adjacent values in good solutions, and this practice can even be counter-productive. Instead of exchanging partitioning information about a group of similarly indexed instance numbers, uniform crossover exchanges information about randomly chosen instance numbers.

5.2 Results

As Johnson et al. [6] and Jones and Beltramo [7] have already shown, using stochastic algorithms to manipulate partitions directly is not very effective (see figure 2). All algorithms, even the genetic algorithm, which is not based upon local optimization, converge rapidly to poor solutions, with an average cost of about 1e-4. Random generate-and-test does worst, but the best algorithm, parallel local optimization, does not do significantly better (about 20% of the parallel local optimization solutions are worse than the median solution of random generate-and-test).

A comparison of the one-argument mutation operators using local optimization shows that, while the one-move operator does significantly worse than the others (1.1e-2), the others differ only in the rate at which they converge to the same local minimum (1.8e-4) (see figure 3). This is because of the significantly higher minimum move cost for the one-move operator. For any given solution, this operator can at best improve it by $1/n$, or 0.01 for the instance in figure 3. As indicated by the figure, no improvement can be found once a solution has a cost of 0.01. All other mutation operators have the same minimum move cost (about $1/n^2$), and converge to $1/100^2 =$ 1e-4. They differ only in the size of the neighborhood they induce around a particular solution and how likely the operator is to find the small improvement needed. As expected, the two-move operator, which has a smaller neighborhood space than the one-swap and one-always-swap operators, takes much longer to converge. The one-swap and one-always-swap operators, which have the same neighborhood size and differ only slightly in how likely they are to find better solutions, converge at roughly the same rate.

The poor performance of the genetic algorithm is due to the crossover operators (see figure 4). Those runs of the genetic algorithm that are most successful are those that use the least crossover, and no crossover seems best of all (only one-swap mutation). As expected, when crossover is used, the uniform operator seems to work better than the two-point variant.

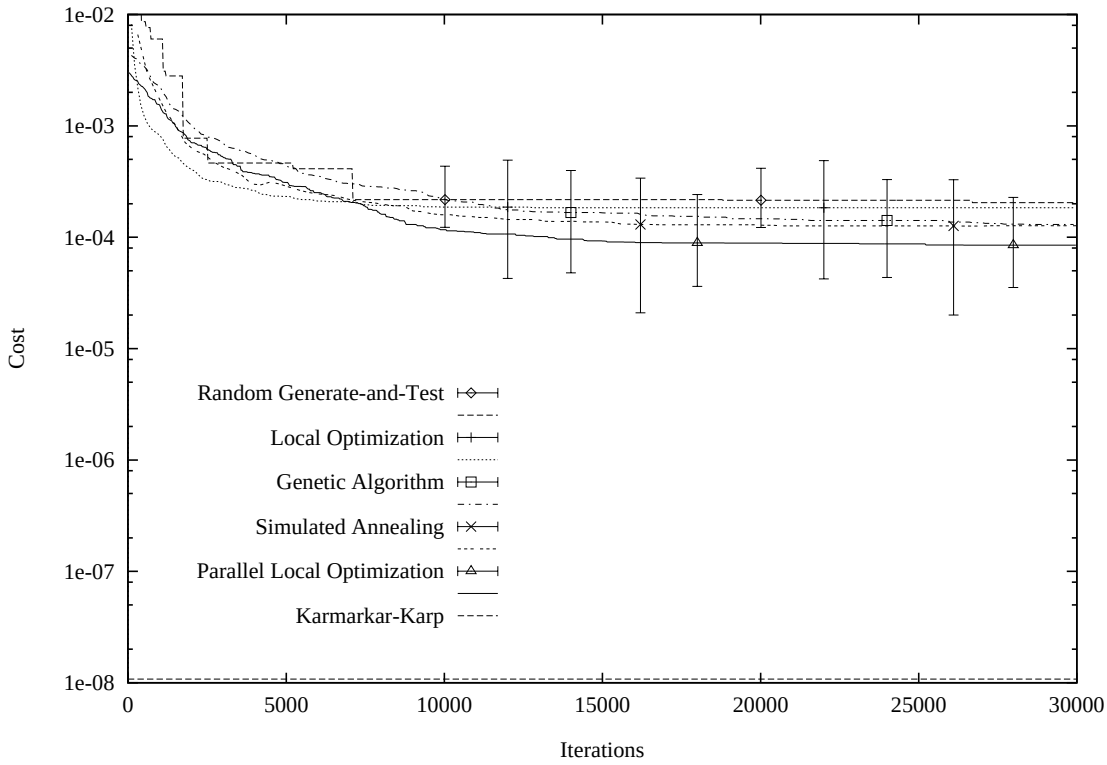

Figure 2: Solution cost over time for all algorithms manipulating partitions directly. All results from stochastic algorithms are the geometric mean of 100 runs. Error bars represent standard deviation. Each curve is identified by a unique point marker (such as '◇') and dash pattern. Note the horizontal line at 1.08e-8 representing the solution found by the Karmarkar-Karp algorithm.

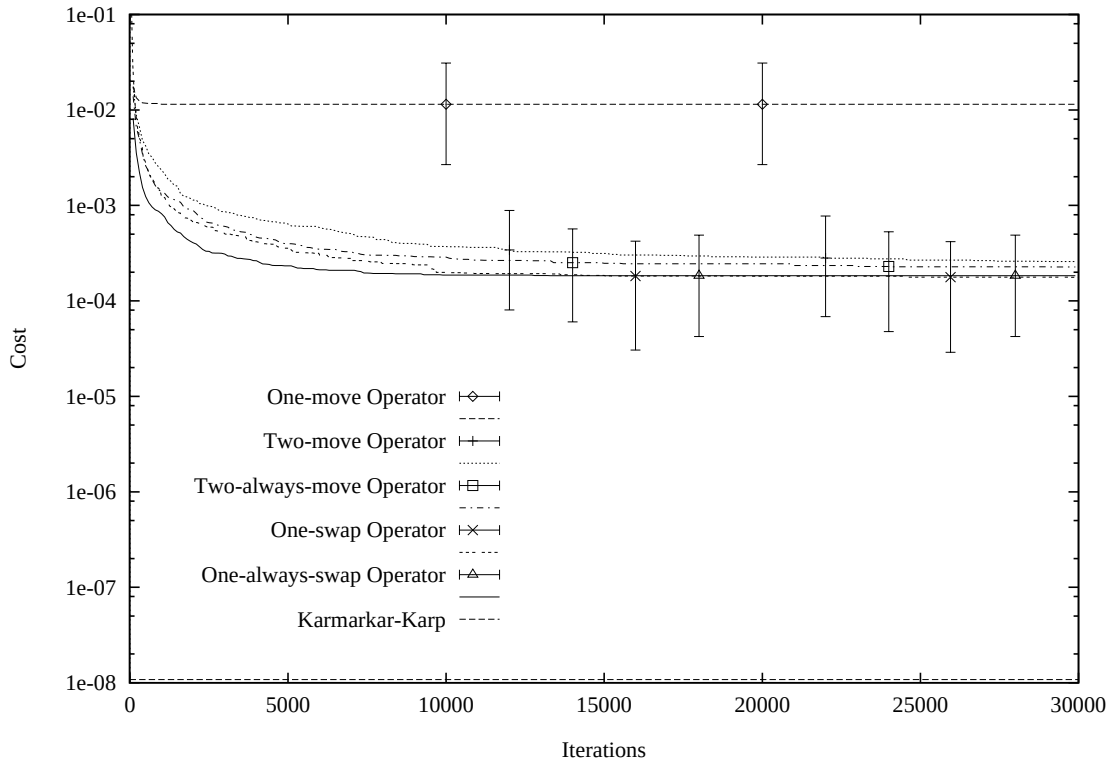

Figure 3: Solution cost over time for the local optimization algorithm using the direct representation of partitions and many different operators. The one-move operator does significantly worse than the rest.

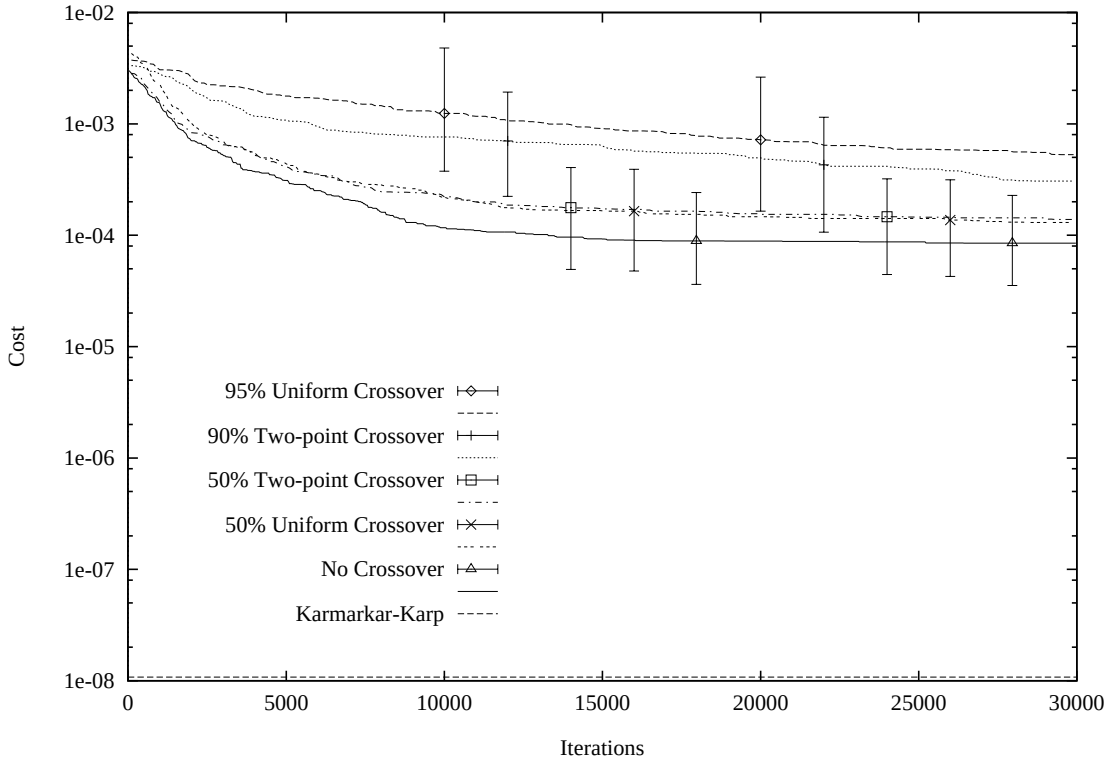

Figure 4: Performance of the genetic algorithm with different crossover operators on an arbitrarily ordered 100-element instance. The lowest curve represents the genetic algorithm without any crossover at all.

Surprisingly, an attempt to increase the linkage in the encoding by indexing the instance numbers in sorted order does not seem to help two-point crossover approach the effectiveness of uniform crossover (figure 5). This may indicate that the schema reproduced by uniform crossover do not involve instance numbers of similar sizes, but instead specify scattered groups composed of numbers of many sizes.

Given these expectedly poor results, we concur with Johnson et al. and Jones and Beltramo that the enormous size of the representation space and the great variety of possible solutions ($n/2$ to $\sqrt{n}/2^n$ for the average instance) yields a terrain that is just too large and mountainous for a search-based algorithm.

6 Permuted Lists with Decoders

Following Jones and Beltramo [7], we have defined several representations based around a permutation of (1,2,..., n). Such a list is manipulated by operators without reference to its meaning, but then serves as instructions to a decoding algorithm for constructing a partitioning, usually specifying the order in which to consider the instance numbers. Each decoding transformation we define below is many-to-one, and maps the $n!$ possible permutations to 2^{n-1} or fewer partitionings.

6.1 Operators

Operators for permuted lists are an active area of research in the genetic algorithm community; we have merely implemented the most traditional:

one swap This operator returns the permuted list obtained by swapping the positions of two distinct elements of the given list. This induces a neighborhood of size $n(n-1)$ around any permuted list of length n. Since this neighborhood is defined in the space of permutations, neighboring solutions may correspond to radically different partitionings, depending on the decoder algorithm. Because of this, it is difficult to generalize about neighborhood structure and move costs under this operator.

pmx This operator, known as 'partially matched crossover,' is an extension of two-point crossover to permuted lists. As before, the s'_i are initialized from the s_i. But instead of merely swapping values between

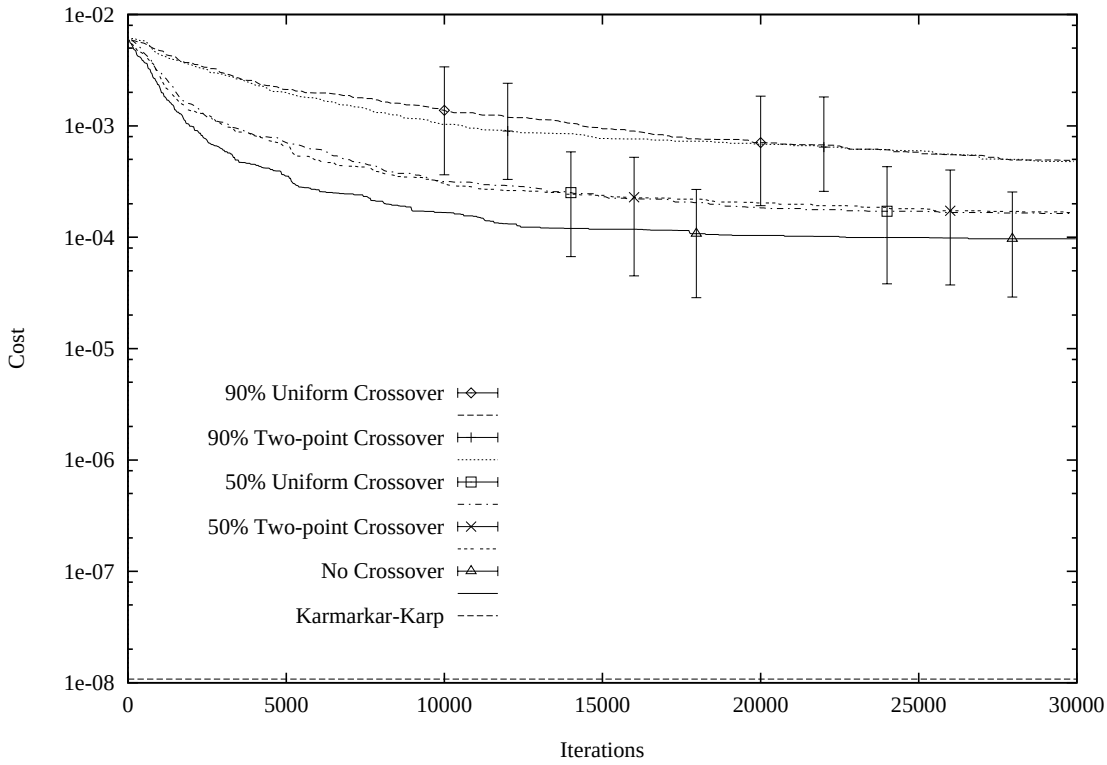

Figure 5: The genetic algorithm using the direct representation of partitions and two-point and crossover on a sorted problem instance. Two-point crossover still does worse than uniform crossover, and both tend to hinder the progress of the algorithm. The lowest curve represents parallel local optimization.

s_1	=	9	8	4	5	6	7	1	3	2	10
s_2	=	8	7	1	2	3	10	9	5	4	6
s'_1	=	9	8	4	2	3	10	1	6	5	7
s'_2	=	8	10	1	5	6	7	9	2	4	3

Table 5: An example of pmx between permuted lists. If '|' represents each crossover point, pmx between solutions s_1 and s_2 will produce solutions s'_1 and s'_2. (This example comes from p. 171 of Goldberg [3]).

solutions, which would yield encodings that were not permuted lists, for all positions $c_1 \leq i < c_2$ between crossover points c_1 and c_2, the value a in position i of s_1 is transferred to the new solution s'_2 by swapping a from its current position j in s'_2 with whatever value b is at position i in s'_2. Solution s'_2 will then have a at the proper position i and b in position j, where a originated (see table 5 for an example). Since the transfer of information is accomplished using this swapping process and not via direct copying, the resulting solutions are said to be 'partially determined' by the originals.

uniform crossover Again, we extend the operation for the canonical representation to preserve permuted lists. Starting with $s'_i = s_i$, we choose, at every place in the permuted list, whether or not to force the instance number specified by $s'_{\{1,2\}}$ at that place to match the number specified in $s_{\{2,1\}}$. If we chose to exchange the information, it is performed just as with pmx, by rearranging one solution based on information from the other solution (see table 6 for an example). As in the canonical case, we hope that this will be effective for an unstructured encoding and eliminate any bias toward short schema.

6.2 The Splitting Decoder

We have defined several decoding transformations from permuted lists to partitionings. The splitting decoder follows the most straightforward approach. Given a permuted list of length n, the first partition A_1 of the decoded solution consists of the instance numbers indexed by the first $n/2$ elements of the permuted list. The second partition A_2 is formed from the remaining elements. For example, if the permuted list were $(1, 3, 2, 4)$, then

s_1	=	9	8	4	5	6	7	1	3	2	10
s_2	=	8	7	1	2	3	10	9	5	4	6
r	=	0	1	1	0	1	0	0	1	0	1
s'_1	=	9	7	1	10	3	8	4	5	2	6
s'_2	=	7	8	4	2	6	5	9	3	1	10

Table 6: An example of uniform crossover between permuted lists. When one of the random numbers r_i hold the value one, information is exchanged at position i in the permuted list. A uniform crossover between solutions s_1 and s_2 will yield solutions s'_1 and s'_2.

the first and third instance numbers would be put in A_1, and the second and fourth in A_2.

Given the correct permutation, one can create any partitioning such that $|A_1| = |A_2|$, or $\frac{1}{2}\binom{n}{n/2} = n!/2(n/2)!^2$ different partitionings. Unfortunately, one has no guarantee that the optimal solution will be 'balanced' in this way. If it is not, an algorithm using the splitting decoder will be unable to find the global optimum. But given randomly selected instance numbers such as ours, a random balanced partitioning is more likely to be close to the optimum than a completely random partitioning. For an average instance, the worst possible solution constructible by this decoder has a cost of $n/4$, as opposed to a cost of $n/2$ for the worst possible partitioning. The solution space is smaller as well; the ratio of $\binom{n}{n/2}$ to 2^{n-1} is $2(n!)/2^n(n/2)!^2$, which tends to zero as n increases (for example, $\frac{1}{2}\binom{100}{50} = 5.05\text{e}28 < 6.34\text{e}29 = 2^{99}$).

6.2.1 Results

The results using permuted lists and the splitting decoder are slightly better than those for the direct partition representation, but still poor (see figure 6). Methods based on local optimization converge to solutions similar to those found using partitions directly (1.7e-4). The restricted range of the decoder is evident, however, in the surprisingly good performance of the random generate-and-test algorithm. It starts with the expected poor results, but continues to find better solutions as it runs, surpassing all but the genetic algorithm after 23,000 iterations. This may be due to the richer nature of the search space.

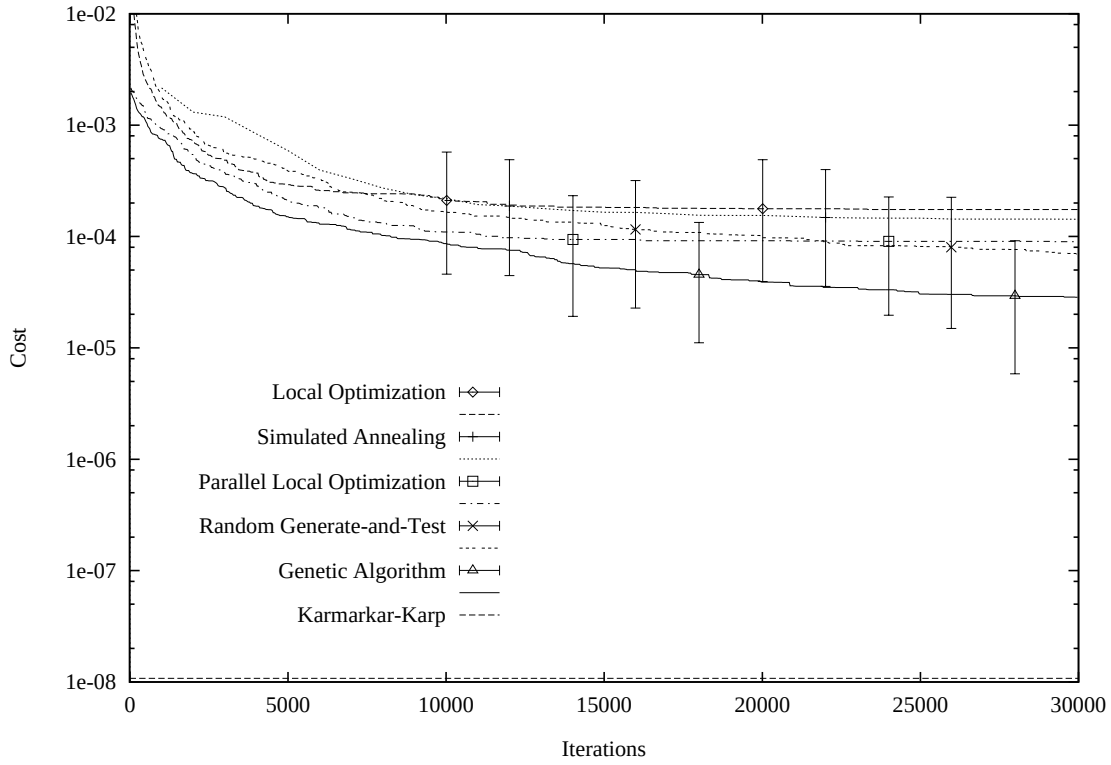

Figure 6: Performance of all algorithms using the permuted list representation with the splitting decoder. Note the line representing random generate-and-test, which has the highest value at first, but then surpasses all but the genetic algorithm.

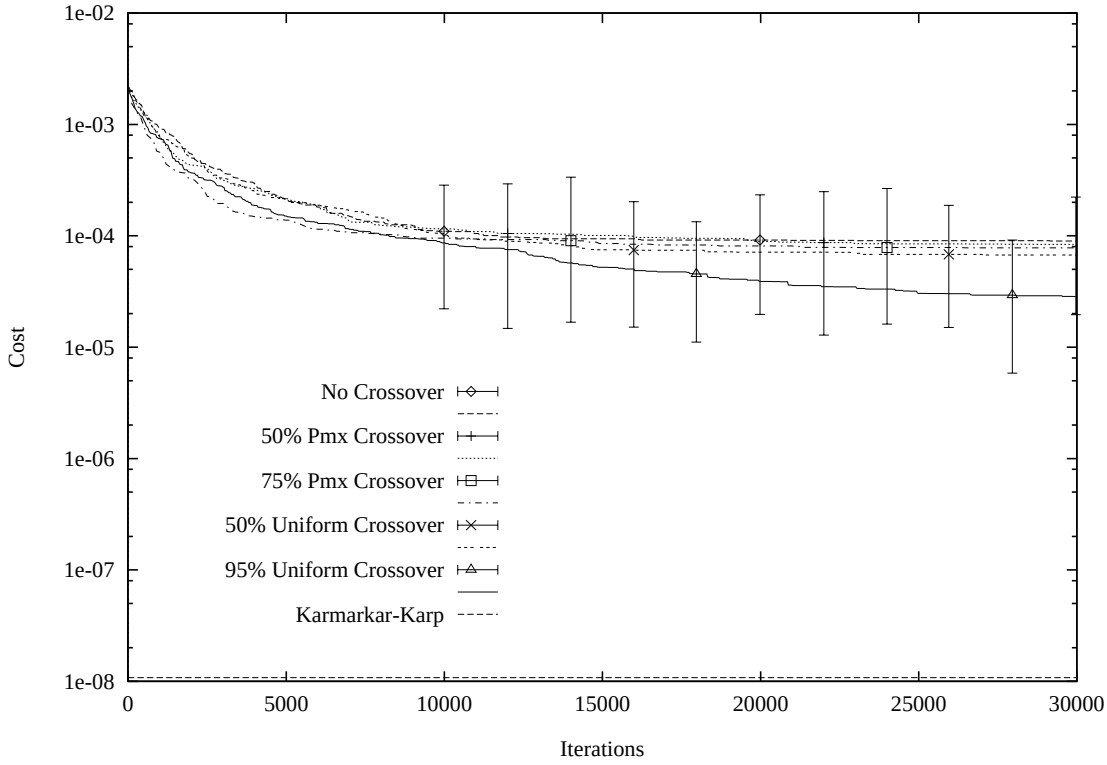

Figure 7: The effect of crossover operator and crossover frequency on the genetic algorithm using the permuted list representation with the splitting decoder. Uniform crossover does much better than pmx.

When using the genetic algorithm, the uniform crossover operator performed quite well (figure 7). Two-point crossover seemed ineffective, and performed only as well as parallel local optimization. This is probably due to the nature of the decoder; information regarding both partitions is more valuable, and is more likely to be transferred by uniform crossover. Two-point crossover, on the other hand, is more likely to specify the indices that should appear at one isolated part of the permuted list, thereby specifying only part of one partition while ignoring the other.

Using permuted lists with the splitting decoder is certainly more effective than manipulating partitions directly. But there's more than one way to decode a permuted list!

6.3 The Alternating Decoder

The alternating decoder forms a partitioning by putting the instance numbers corresponding to every alternate index in the permuted list in the same partition. For example, given the permuted list $(1, 3, 2, 4)$, the first and second instance numbers would be put in the same partition. This interleaving of information about the two partitions should allow a two-point crossover to work more effectively.

Like the splitting decoder, the alternating decoder can only create balanced partitions, thus restricting the space of possible solutions and lowering the value of the worst possible solution.

6.3.1 Results

Since only the ordering of information has changed, one would expect results similar to those of the splitting decoder, with the possible exception of the genetic algorithm. As figure 8 shows, this representation held no surprises. As with the splitting decoder, methods based on local optimization fared poorly, while the random generate-and-test algorithm continued to easily find better solutions throughout the run.

Using a genetic algorithm, one can see the impact of the rearrangement of information in the encoding (figure 9). Now that information about both partitions can be represented in any short schema, two-point crossover will be able to propagate useful information. Indeed, the genetic algorithm finds better solutions during the first 30,000 iterations when using pmx crossover than it does with uniform crossover, although it converges sooner. As with the splitting decoder, selecting the crossover operator only half of the time produces negligible results when using a large population of 1,000 solutions, and the more frequently it is used, the better the results.

As expected, algorithms using permuted lists with the alternating decoder find the same quality of solutions as they do using the splitting decoder, although the improved performance of the pmx crossover allows the genetic algorithm to find good solutions faster.

6.4 The Number-Based Splitting Decoder

Although the splitting and alternating decoders improve upon the direct manipulation of partitions, they impose the arbitrary restriction that solutions must have the same number of numbers in each partition, ignoring the actual instance numbers that are being partitioned. The number-based splitting

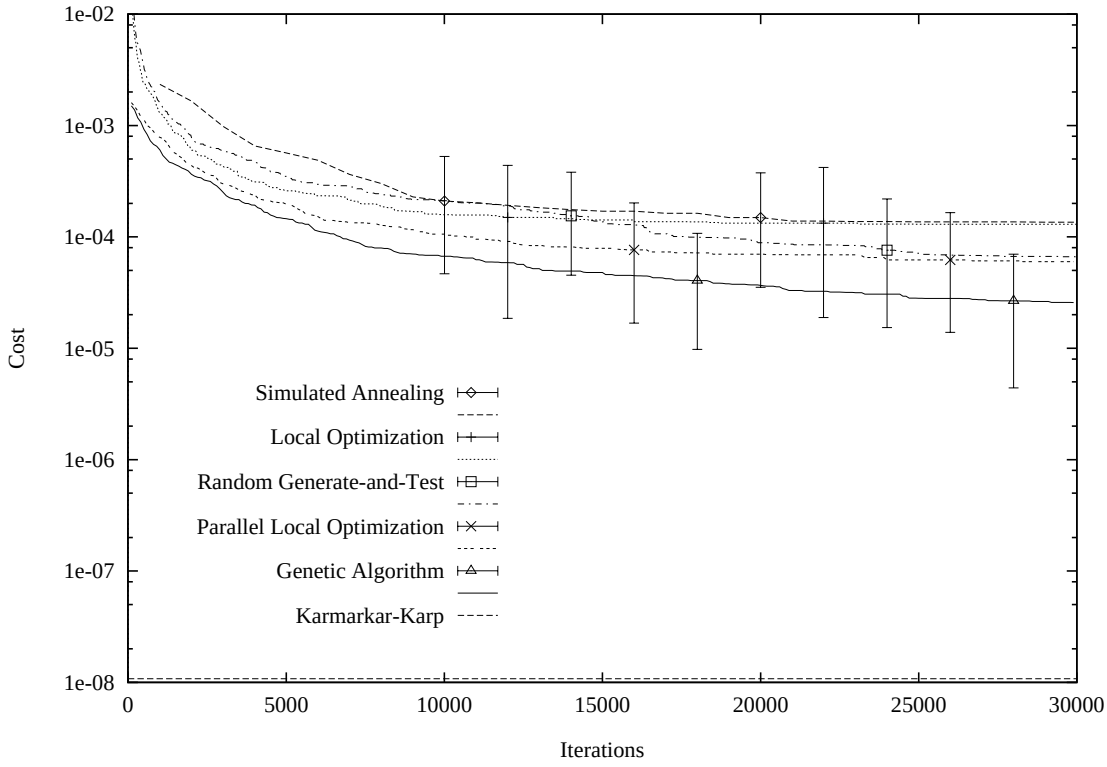

Figure 8: Performance of all algorithms using the permuted list representation with the alternating decoder. As with the split decoded, random generate-and-test continues to find good solutions, but the genetic algorithm is the leader.

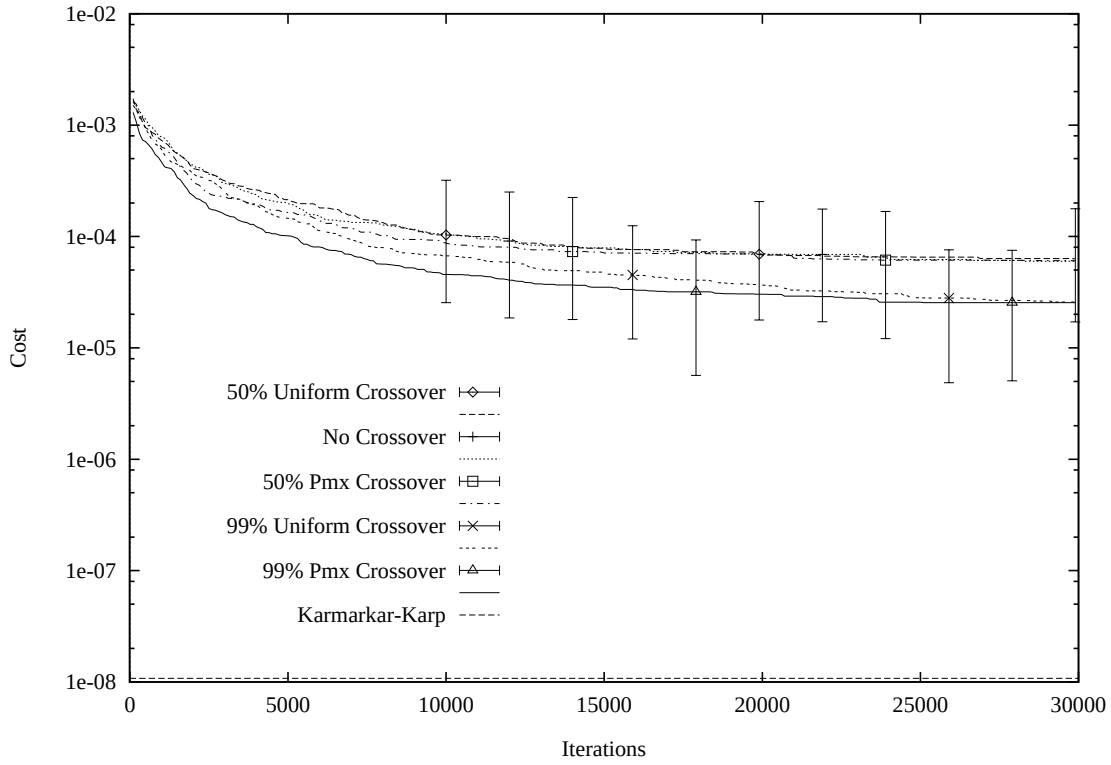

Figure 9: Performance of the genetic algorithm using the permuted list representation with the splitting decoder. Pmx crossover converges faster than uniform crossover, but uniform continues to improve and eventually finds slightly better solutions.

decoder is a variant of the splitting decoder that addresses this need. It creates a partitioning from a permuted list by adding instance numbers to the first partition, in the order specified by the permuted list, until their sum exceeds the ideal partition size. (Since the instance numbers are static, the ideal partition size can be precomputed as half their sum.) The remaining numbers are put into the second partition. The instance number that causes the first partition to overflow is added to whichever partition has a lower final sum.

The number of different solutions that can be constructed in this way depends heavily on the problem instance, and cannot be easily quantified. The cost of the worst possible solution is easily specified, and is equal to half the largest instance number, or 0.5 on average. For any reasonably sized problem instance, this is much smaller than $n/4$, the worst solution cost for the splitting and alternating decoders. In addition, the optimal solution can always be constructed by this decoder.

6.4.1 Results

Empirical results show that the solution space is indeed richer (figure 10). The random generate-and-test algorithm returns solutions with an average cost of 1.0e-5, as opposed to 6.8e-5 when using the splitting and alternating decoders. The genetic algorithm finds solutions in the vicinity of 5.2e-6, as opposed to 2.6e-5 with the previous decoders. As with the split decoder, uniform crossover is much more effective than pmx.

Simulated annealing also behaves qualitatively differently from before, when it converged in the same manner as local optimization (figure 11). Under the number-based splitting decoder, simulated annealing is able to find solutions that are much better than those found by plain local optimization. This may be because of fewer and less entrenched local optima, since the worst possible solution is much better under this decoder.

Although we have improved the performance of our algorithms by constructing a representation space with fewer poor solutions, none have yielded solutions that are competitive with those constructed by the Karmarkar-Karp algorithm. By making the permuted list decoder even more intelligent, we might hope for better solutions, but at the risk of distancing the decoded solution even further from the encoded representation that the algorithms are manipulating.

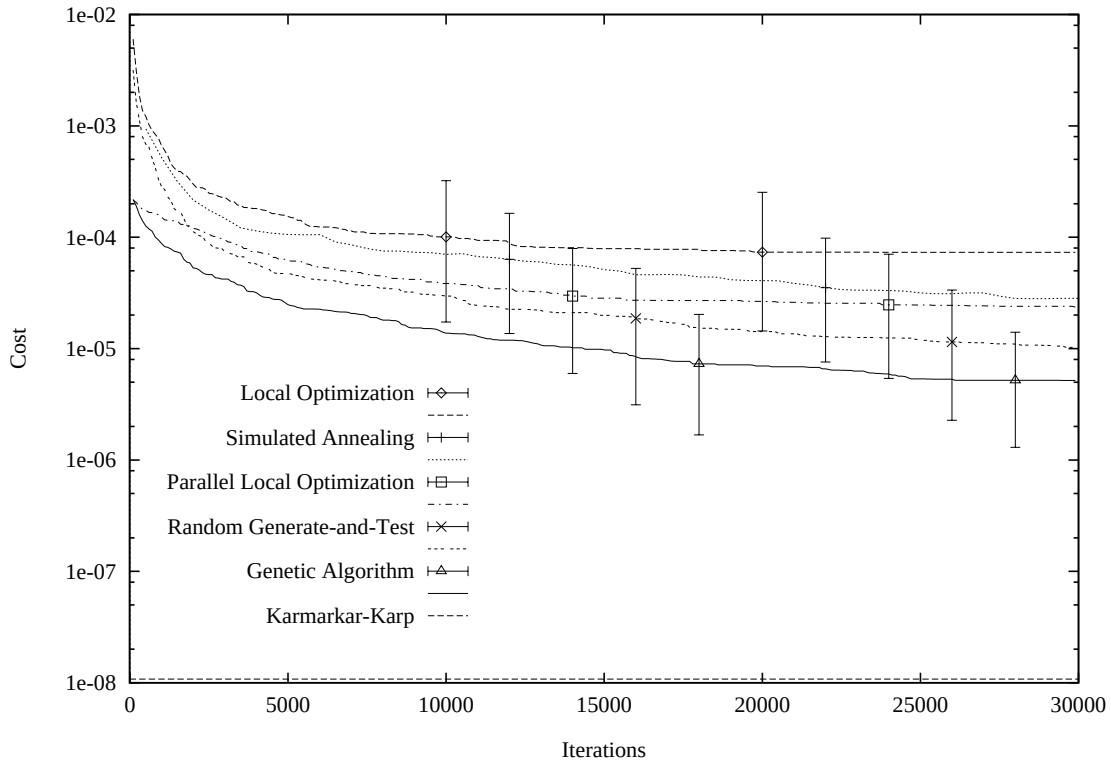

Figure 10: Performance of all algorithms using the permuted list representation with the number-based splitting decoder. Simulated annealing seems to be able to make progress here.

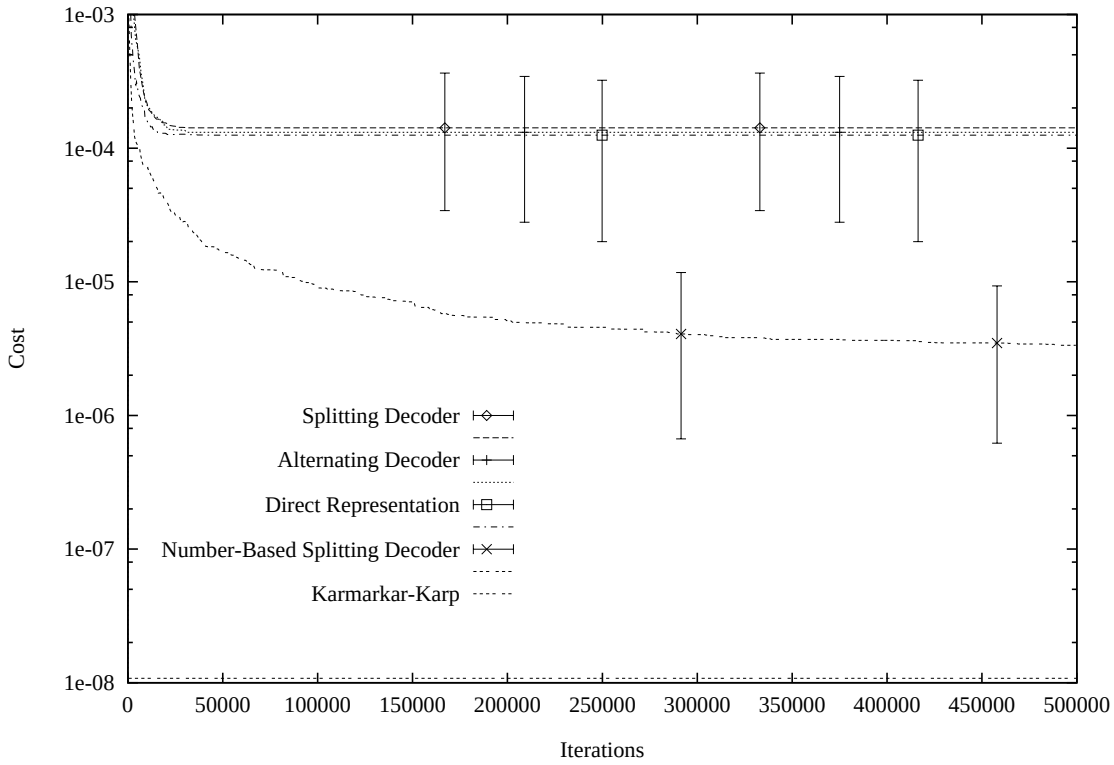

Figure 11: Performance of simulated annealing using the direct representation and the permuted list representation with the splitting, alternating, and number-based splitting decoders. The algorithm's behavior is qualitatively different with the number-based decoder. This graph represents more than sixteen times as many iterations as those in previous figures.

6.5 The Greedy Splitting Decoder

The greedy splitting decoder is a greedy version of the number-based splitting decoder. The number-based decoder split the permuted list by putting instance numbers into the first partition until it overflowed, and then putting the remaining numbers in the second partition. This greedy variant puts numbers into the first partition until it overflows, as did the previous decoder, but it then checks to see if each additional number will fit into the remaining space in the first partition. Only after such numbers have been added to the first partition are the remaining ones relegated to the second partition.

6.5.1 Results

The results of algorithms using the greedy splitting decoder are about an order of magnitude better than those obtained using the number-based splitting decoder (compare figures 12 and 10). The random generate-and-test algorithm returns solutions with an average cost of 1.2e-6 (versus 1.0e-5), and local optimization gives results averaging 8.3e-6 (versus 7.3e-5). Qualitatively, however, the results are very similar. Single and parallel local optimization both converge within 20,000 iterations, revealing the presence of many local optima, while random generate-and-test and the genetic algorithm continue to improve, reflecting the richness of the representation space. Due to the lack of linkage in the encoding, uniform crossover was again much more successful than pmx.

The only qualitative difference is in the behavior of the simulated annealing algorithm, which does not improve as quickly during the first 5,000 iterations relative to local optimization as it did using the non-greedy decoder. This may be because a random permuted list is more likely to decode into an acceptable partitioning, thus distracting the algorithm from vigorously exploring one particularly good area of the representation space. In other words, if almost any solution is good, there is less incentive to discriminate when the *temperature* variable is high. Nevertheless, the algorithm does make steady progress, and eventually succeeds in finding particularly good solutions (solutions after 500,000 iterations have an average cost of 1.5e-7; see figure 13). The solution space is so rich, however, that random generate-and-test surpasses all other algorithms within 150,000 iterations. This shows that even the most trivial of search methods can outperform more sophisticated techniques when using the right representation.

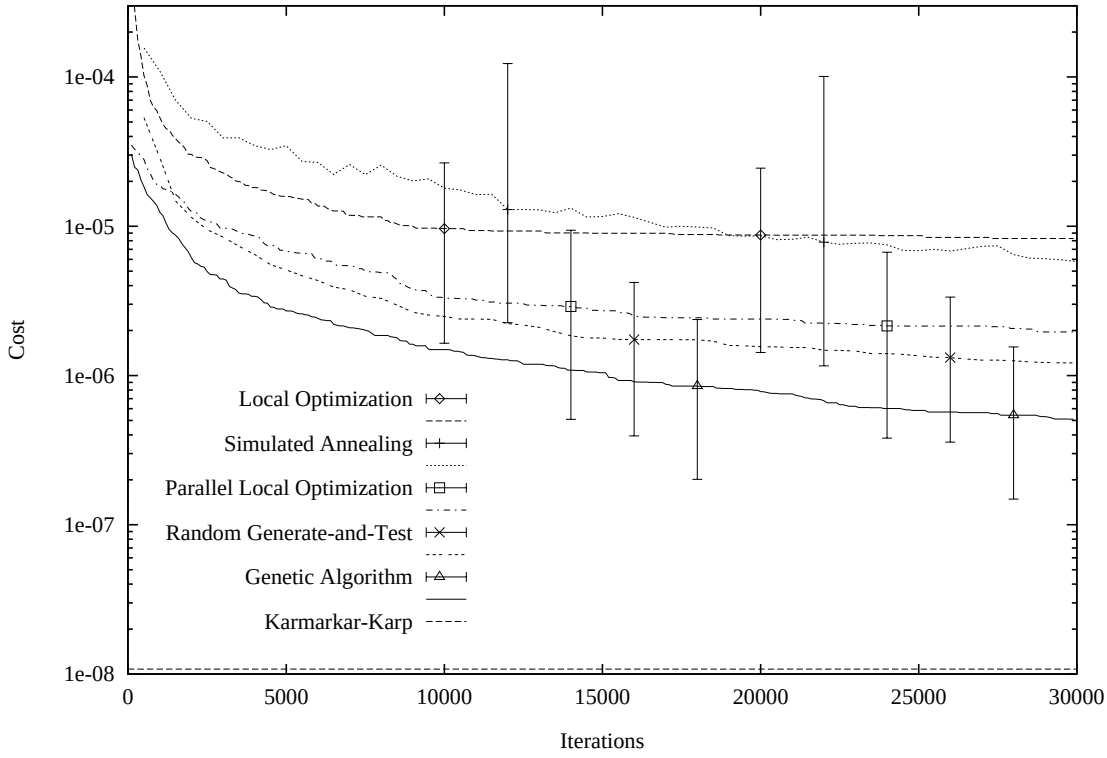

Figure 12: Performance of all algorithms using a permuted list representation with the greedy splitting decoder. Qualitatively similar to, although quantitatively better than, the number-based splitting decoder (compare with figure 10).

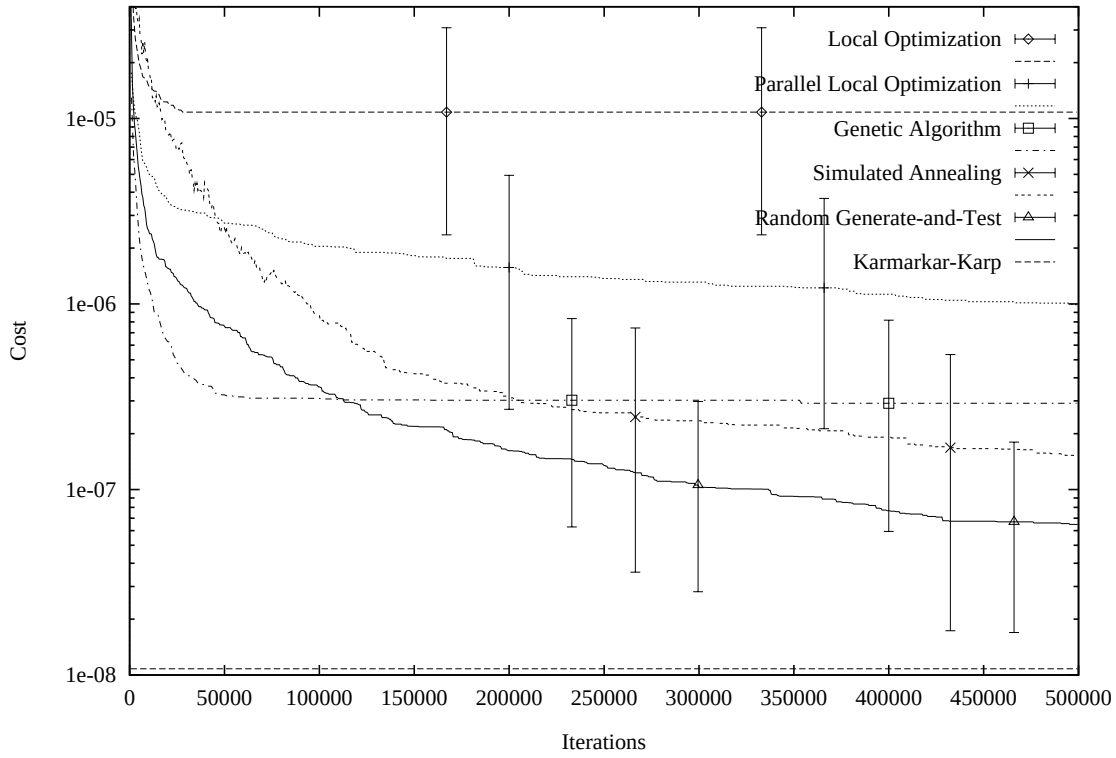

Figure 13: Long runs of algorithms using a permuted list representation with the greedy splitting decoder. The results from the random generate-and-test and simulated annealing algorithms approach the quality of the Karmarkar-Karp solution.

The additional heuristic of the greedy splitting decoder does not seem to significantly hinder any of the stochastic algorithms we have tried, and it consistently improves the quality of their solutions.

6.6 The Greedy Decoder

While the number-based splitting decoder seems successful, it may be more complicated than necessary. We have also implemented a plain greedy decoder, which just considers the instance numbers in the order in which they are specified by the permuted list, and adds a given number to the partition with the currently lowest sum. Just as the number-based splitting decoder is an enlightened variant of the ordinary splitting decoder, so the plain greedy algorithm could be considered a more flexible version of the alternating decoder.

The solution space of the greedy decoder is similar to that of the number-based splitting decoder: the worst solution that can be constructed has a cost of at most half the greatest instance number, 0.5 on average, and the optimum is clearly constructible.

6.6.1 Results

The results from algorithms using permuted lists and the greedy decoder are encouraging (see figure 14). Although random generate-and-test yields solutions that are comparable on average to the number-based splitting decoder (1.1e-5 compared to 1.0e-5 versus 1.2e-6 for the greedy splitting decoder), local optimization finds better solutions using this decoder than with any of the previous ones (3.3e-6 versus 8.3e-6 for greedy splitting). This may be because the neighborhood space of the greedy decoder is likely to be more continuous under a one-swap operator that it would be for a more complicated decoder, which induces more interdependencies in the encoding. The path of a local descent algorithm would thus be smoother and easier to traverse.

The genetic algorithm does quite well, although crossover does not seem particularly beneficial (figure 15). Uniform crossover clearly hinders the algorithm, but pmx seems be little different from a mutation. Parallel local optimization, which is a genetic algorithm without crossover, finds solutions which score, on average, the same as the genetic algorithm with frequent pmx crossover. Runs using only infrequent pmx crossover produce similar results on average, but are more consistent and exhibit less variety (i.e.,

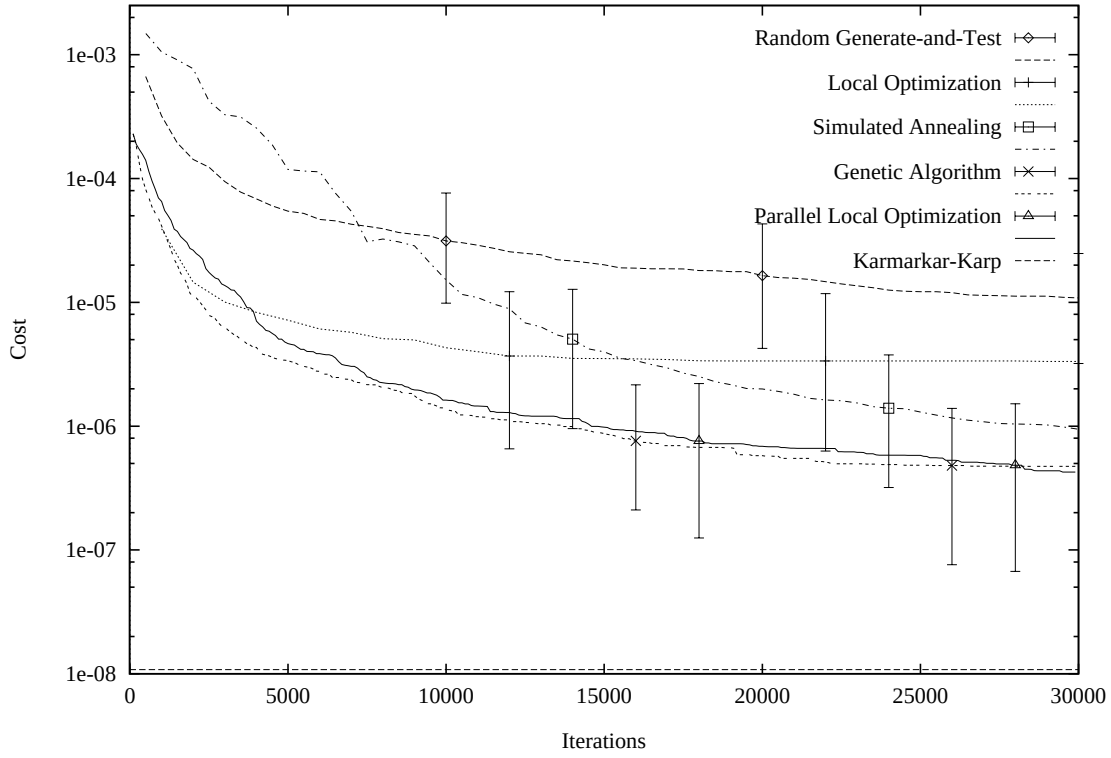

Figure 14: Performance of all algorithms using a permuted list representation with the greedy decoder. Parallel local optimization does just as well as the genetic algorithm.

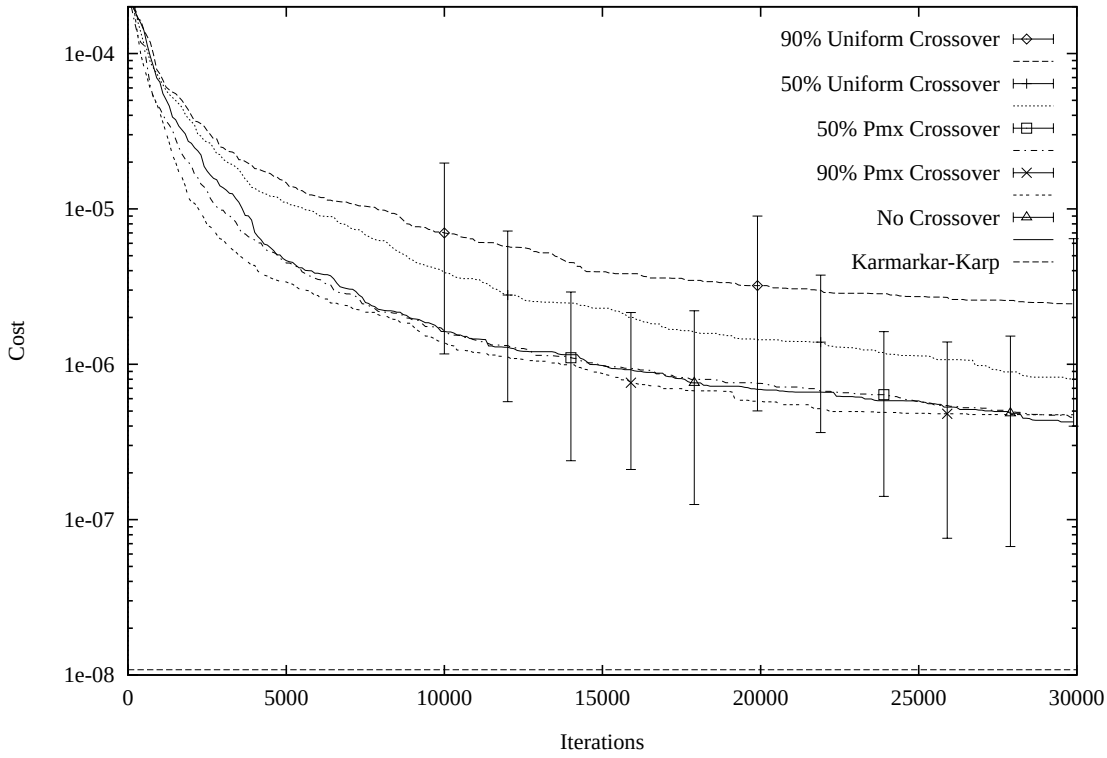

Figure 15: Performance of the genetic algorithm using a permuted list representation with the greedy decoder. No crossover at all seems to be best. Lots of pmx crossover is almost as good, while any amount of uniform crossover significantly hinders the algorithm.

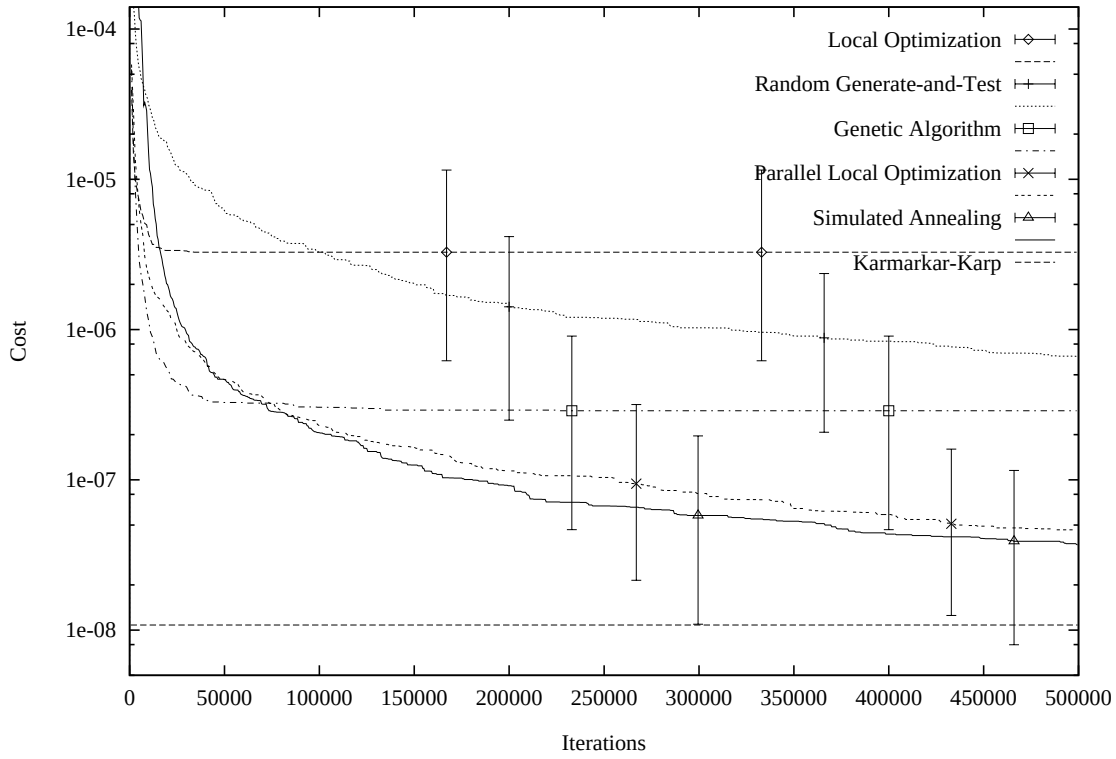

Figure 16: Long runs of algorithms using a permuted list representation with the greedy decoder. Parallel local optimization and simulated annealing are able to find solutions better than that constructed by the Karmarkar-Karp algorithm.

smaller standard deviation).

6.6.2 Long Runs

Although none of these algorithms can quickly and regularly find solutions that are better than the one constructed by the Karmarkar-Karp algorithm, both parallel local optimization and simulated annealing can find comparable solutions if given enough iterations (see figure 16). After 500,000 iterations, 13% of parallel local optimization runs (eight minutes per run) have found a solution better than the Karmarkar-Karp algorithm's, while a

full 20% of simulated annealing runs are superior (six minutes per run). If one needed a solution better than that of the Karmarkar-Karp algorithm, five runs of simulated annealing (about half an hour) would suffice. And since simulated annealing is a search method, one can always run it longer if one desires a better result.

6.6.3 Large Instances

While the permuted list representation with the greedy decoder allows parallel optimization and simulated annealing to compete with the Karmarkar-Karp algorithm on a 100-element problem instance, the search space induced by a 200-element problem instance remains beyond their grasp. Both local optimization and simulated annealing fall far short of the Karmarkar-Karp solution, showing only limited improvement over the quality of solutions found for the 100-element problem (figure 17). The slow improvement of simulated annealing after 125,000 iterations indicates that the algorithm is progressing on the 200 element problem much as it did in the 100 element case; implying that even doubling the number of iterations would not yield a solution close to that found by the Karmarkar-Karp algorithm.

While effective for finding solutions to 100 element instances that are competitive with those of the Karmarkar-Karp algorithm, it seems that the permuted list representation is not capable of performing well on larger problems.

6.7 Summary of Results

Although comparing the performance of various algorithms using a single representation has helped elucidate the structure of each representation space, it is also useful to compare the performance of the same algorithm using different representations. This makes the differences between the representations strikingly apparent.

6.7.1 Representation Spaces

Comparing the performance of simple algorithms in different representations can give a rough idea of the character of the search space defined by each. Table 7 compares the performance of the random generate-and-test algorithm with that of local optimization in each of the representations we have considered. The value of the worst representable solution in each representation gives a relative comparison of the density of good solutions in

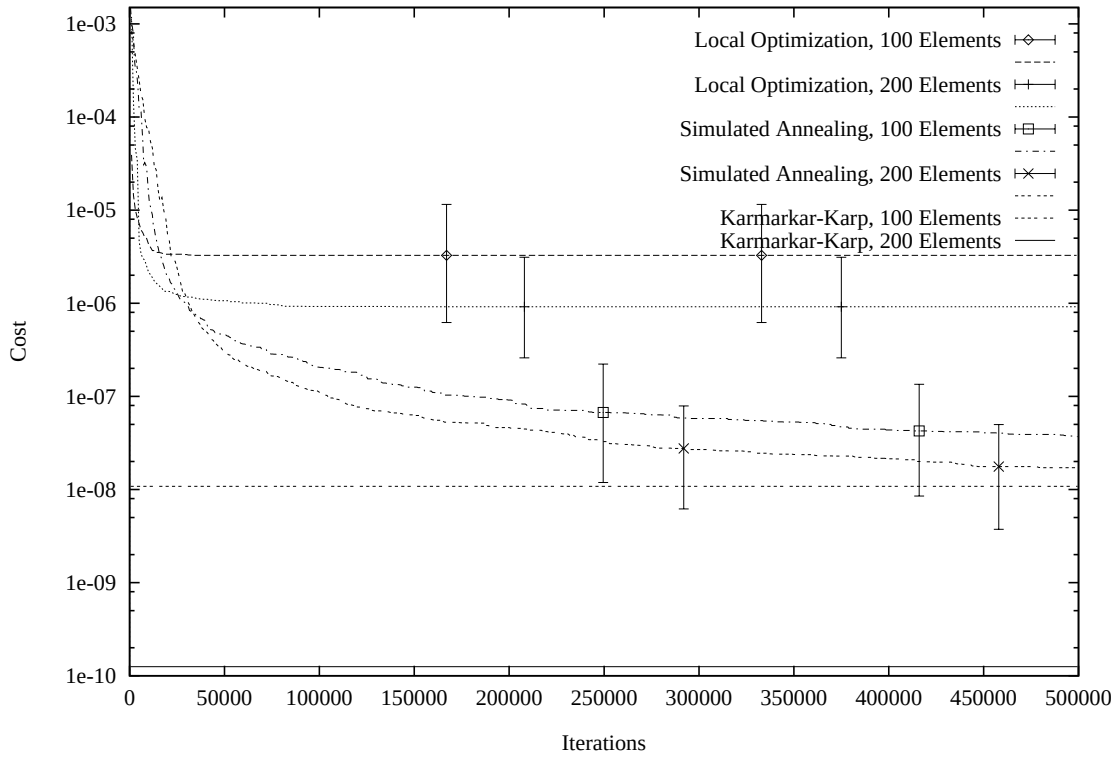

Figure 17: Local optimization and simulated annealing using a permuted list representation with the greedy decoder on problem instances of size 100 and 200. Neither algorithm is able to improve as much as the Karmarkar-Karp algorithm relative to its performance on the 100-element instance. Note the Karmarkar-Karp solution at 1.25e-10.

representation	random	worst sol.	local
direct	2.04e-4	n/2 = 50	1.76e-4
splitting	6.94e-5	n/4 = 25	1.75e-4
alternating	6.60e-5	n/4 = 25	1.30e-4
number splitting	1.02e-5	0.5	7.30e-5
greedy splitting	1.21e-6	< 0.5†	8.28e-6
plain greedy	1.09e-5	0.5	3.34e-6

Table 7: The results of the random generate-and-test and local optimization algorithms using each of the representations we have described (30,000 iterations), and the worst solution possible for an average instance using that representation. We have not derived the worst solution value for the greedy splitting decoder (marked with '†').

the representation space (the best possible solution value is that of the optimum except in rare cases; recall from section 6.2 that the splitting and alternating decoders will not be able to construct the optimum solution for some problem instances). The performance of the random algorithm should correspond roughly with the density of good solutions in the representation space, while the results of local optimization may give an indication of how mountainous the space appears when structured by the simple one-swap operator. Note the similarities between the representation spaces of the splitting and alternating decoders, reflected in the density of good solutions and the performance of the random generate-and-test algorithm. The number-based splitting and plain greedy decoders are more intelligent versions of these decoders, and also induce similar spaces. Note that the structure of similarly sized spaces under the action of the one-swap operator may be dramatically different; the number-based splitting decoder and the greedy algorithm perform identically under random sampling, but the interdependencies inherent in the splitting decoder mean that neighboring permuted lists may represent very different partitionings. This explains the difference in performance of local optimization using the two representations.

6.7.2 Random Generate-and-Test

The performance of the random generate-and-test algorithm increased dramatically when using a more restricted search space (figure 18). The algo-

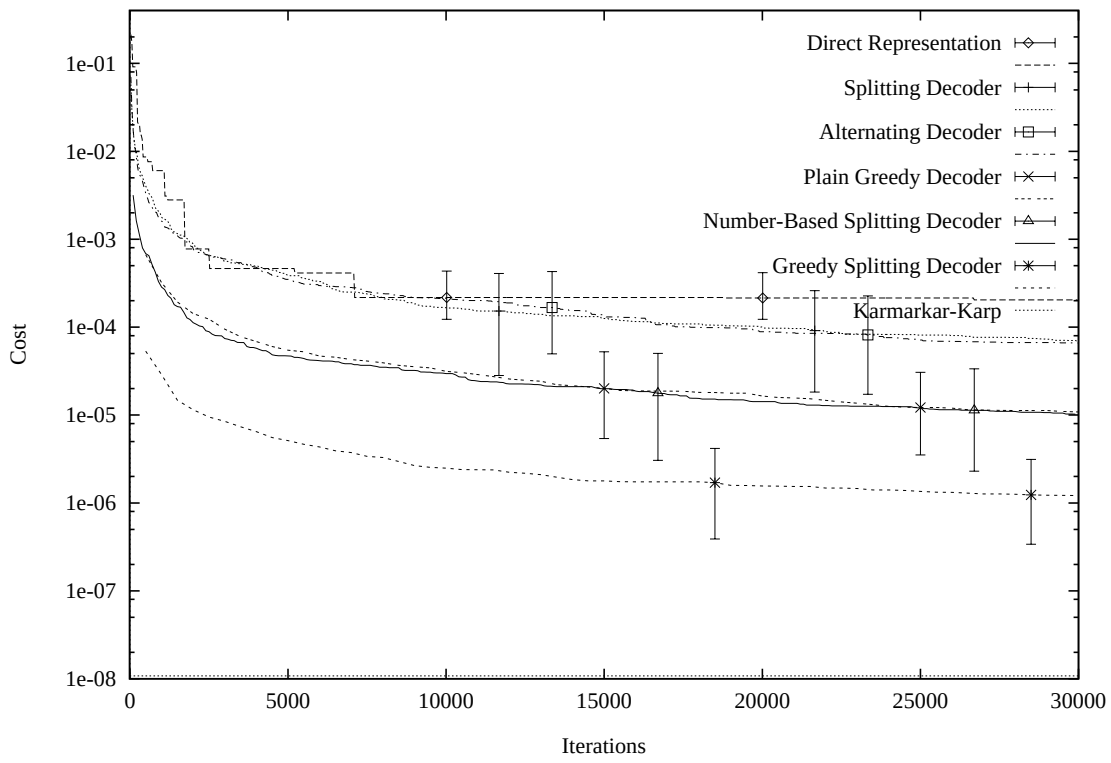

Figure 18: Performance of the random generate-and-test algorithm using many different representations. Note the algorithm's quick convergence when manipulating partitions directly.

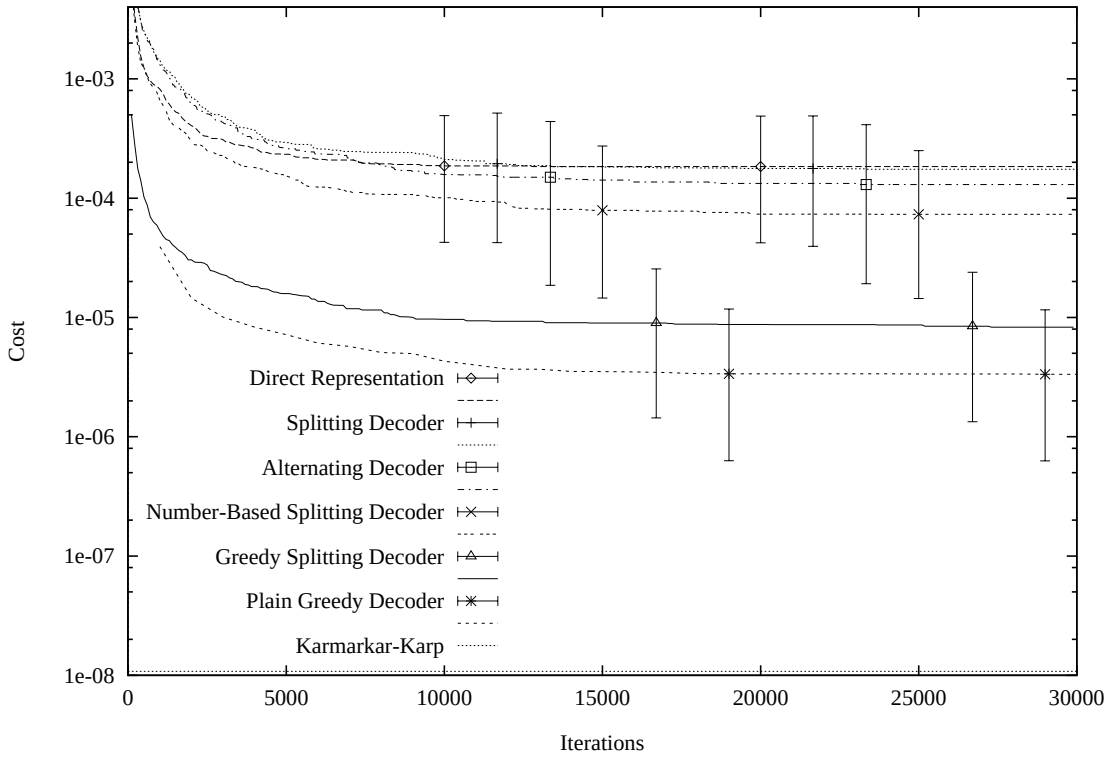

Figure 19: Performance of local optimization using many different representations.

rithm found remarkably similar solutions using the alternating and splitting decoders; this is probably due to the similar densities of their solution spaces. Similarly, results using the number-based splitting decoder and the plain greedy decoder were the same. Random generate-and-test was particularly effective using permuted lists and the greedy splitting decoder.

6.7.3 Local Optimization

As expected, local optimization algorithm was easily trapped in local optima in every representation (figure 19). In contrast to the behavior of random generate-and-test, local optimization performed better using the plain greedy decoder than the greedy splitting decoder, perhaps because of the smoother nature of the neighborhood space under the one-swap op-

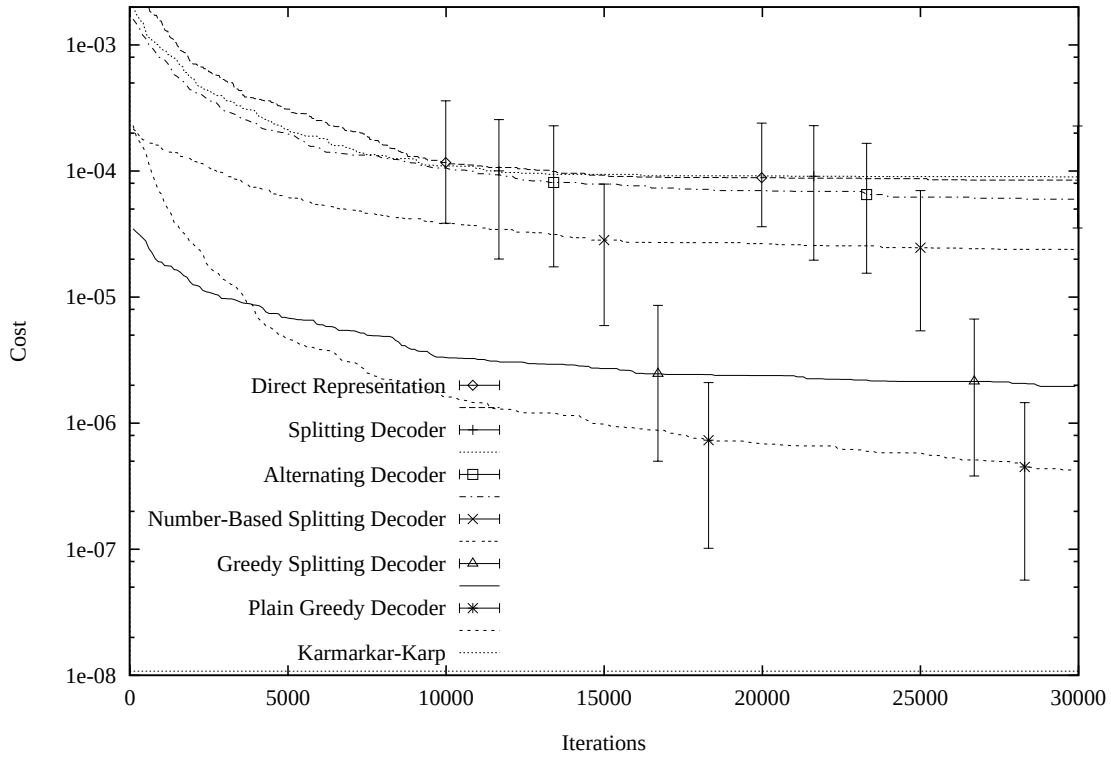

Figure 20: Performance of parallel local optimization using many different representations.

erator. When using the greedy splitter decoder, a swap operation in the permuted list may easily move many instance numbers between partitions.

6.7.4 Parallel Local Optimization

In general, a representation's performance under parallel local optimization was an exaggerated caricature of its behavior under ordinary local optimization, with all representations doing slightly better (figure 20). The permuted list representation with the plain greedy decoder and the index rules representation performed particularly well, while again, the greedy splitting decoder fared relatively poorly. While more robust than its ancestor, due to the simultaneous exploration of multiple solutions, the parallel

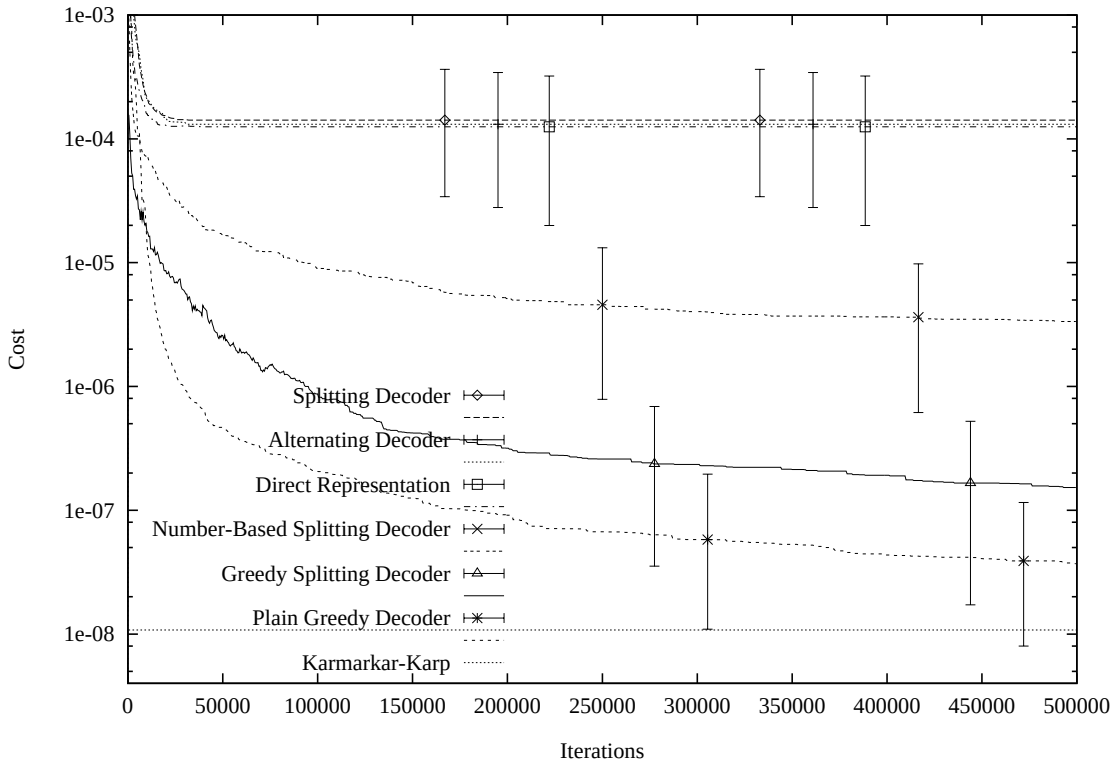

Figure 21: Performance of simulated annealing using many different representations.

variant of local optimization cannot avoid local optima, due to its lack of crossover. The overhead of population management in our implementation (approximately 15–20% of running time) means that parallel optimization is justified only for the plain greedy decoder; other representations may be better served by two runs of standard local optimization.

6.7.5 Simulated Annealing

The simulated annealing algorithm was able to overcome local optima and perform better than local optimization for all representations, but was still limited by the relative densities of the search spaces and, to a lesser extent, by the structure of the spaces (figure 21). For the direct representation and the splitting and alternating decoders, simulated annealing quickly con-

verged to solutions slightly better than those found by local optimization. Despite these gains over plain local optimization, however, annealing does not do as well with these representations as parallel local optimization (refer back to figure 20). Using the greedy variants of these decoders, the greedy splitting decoder and the plain greedy decoder, simulated annealing does qualitatively better than local optimization, but, with the plain greedy decoder, still only marginally better than parallel local optimization. These results show the importance of the structure of the representation space; in a smoother search space, local search from multiple points has a large chance of finding a good solution despite the remaining local minima. The roughness of the space induced by the greedy splitting decoder is even evident in the progress of the simulated annealing algorithm; it often gets stuck briefly in local optima.

6.7.6 The Genetic Algorithm

Contrary to what one might expect, the genetic algorithm performed the worst when using the representations whose encodings are interpreted most literally, and performed the best using the greedy decoders, for whom the relationship between encoding and solution is the loosest (figure 22). The disruptive process of crossover seems to work best when the representation is more robust, such as a permuted list with a greedy decoder. Such forgiving decoders may help smooth the search space, since they are able to construct good solutions from a large percentage of the possible encodings.

6.8 Seeded Algorithms

By changing the representation used by our search methods, we have been restricting their exploration to productive areas of the space of partitionings. An even easier way to focus a search algorithm on a promising area of the search space is to start it at a solution known to be good. We have tried seeding some of the more successful algorithms and representations with initial solutions found by other search algorithms or constructed by a simple sorting procedure.

6.8.1 Local Optimization

Using the permuted list representation with the plain greedy decoder, which performed well under local optimization, we experimented with using local optimization as a post-processor. Using permuted lists and the plain greedy

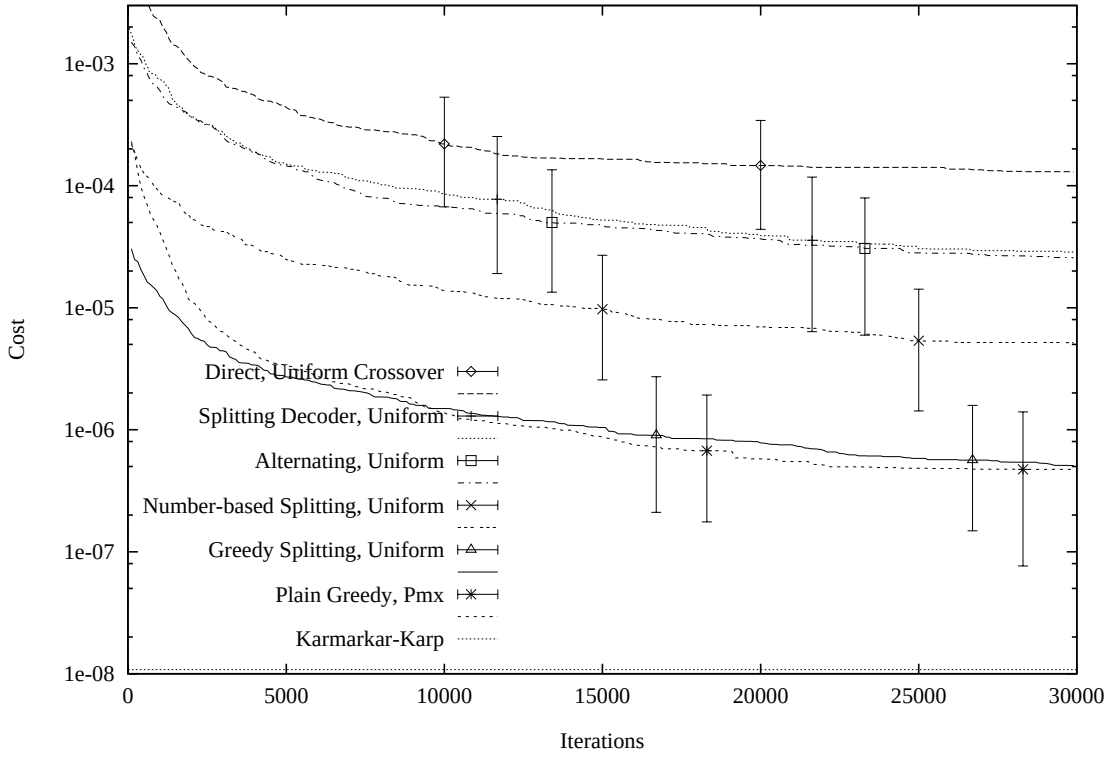

Figure 22: Performance of the genetic algorithm using many different representations. Only results using the most effective crossover operator for each representation are shown.

decoder and seeding with best of 1,000 random solutions, the algorithm did little better than when started from a randomly chosen solution. This shows that the benefits of parallel local optimization do not come merely from starting with access to better than average initial solutions, but from exploring multiple points simultaneously, and allotting exploration time for each solution according to its current performance.

When started with the output of 30,000 iterations of the genetic algorithm, local optimization made small improvements to the seed solutions, but no significant discoveries. The failure of local optimization as a post-processor for the genetic algorithm shows that our implementation is effective at local optimization, as well as global search.

We also tried seeding local optimization with a permuted list that presented the instance numbers to the decoder in sorted descending order (analogous to our experiments with the direct representation and sorted instance numbers). As expected, the number-based splitting decoder performed worse when given this starting solution, and the greedy-splitting decoder did somewhat better (see figure 23). The plain greedy decoder gave the same results with and without the sorted initial solution, as did its cousin, the alternating decoder. Overall, seeding local optimization with a sorted permuted list was not particularly effective.

And, as expected, dozens of runs of 30,000 iterations of local optimization using permuted lists and many different decoders failed to find any improvements to the Karmarkar-Karp solutions.

6.8.2 Simulated Annealing

Using the plain greedy decoder, we have experimented with seeding the simulated annealing algorithm with the solution found by the Karmarkar-Karp algorithm. We used a low starting temperature (*init-prob* = 0.2) so that only moves to other good solutions will be accepted, and lowered the temperature slowly (*temp-factor* = 0.95) so that the algorithm would still have time to explore. Although a graph of the algorithm's progress looks no different than before, it does search the area around the Karmarkar-Karp solution more than usual, as the final results were quite good (8.82e-9 vs. 3.96e-8). Only 10% more runs than usual found solutions better than the Karmarkar-Karp solution, though, so multiple runs are still necessary in order to guarantee a superior solution.

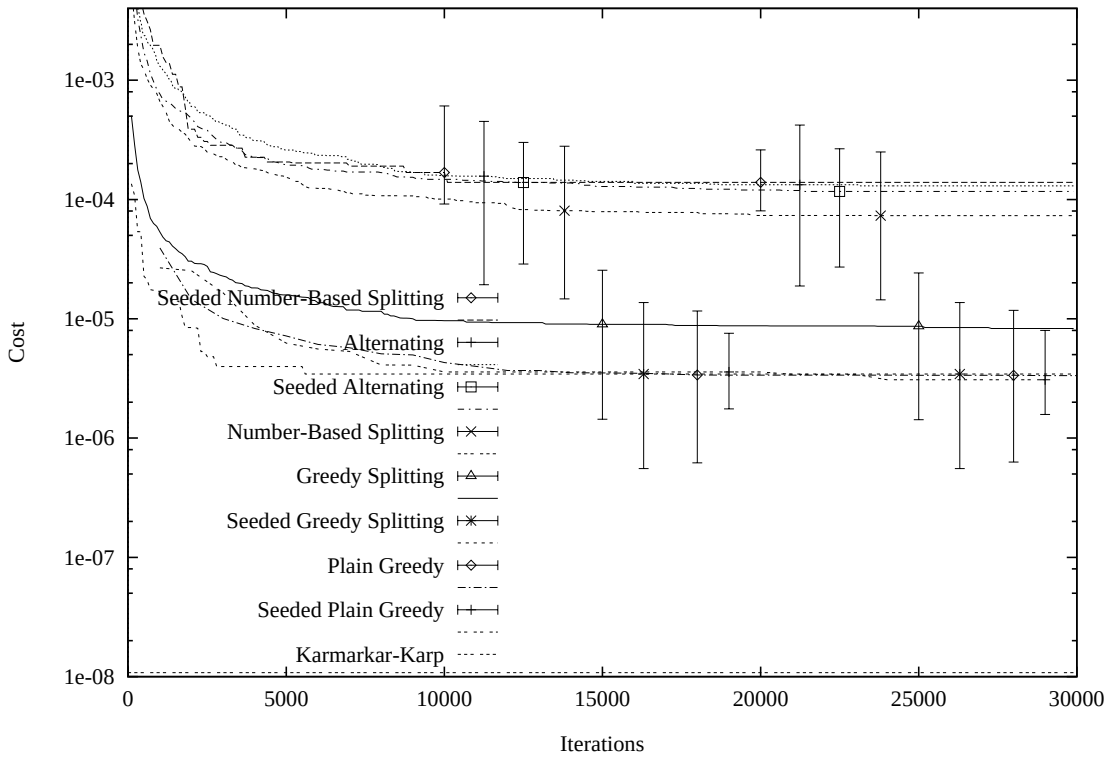

Figure 23: Performance of local optimization using permuted lists and many different decoders, each seeded with a permuted list representing the sorted instance numbers. Note the decreased performance of the number-based splitting decoder, and the increased performance of the greedy splitting decoder.

7 Difference Rules

Although the algorithms and representations of the previous section can find solutions that are superior to those produced by the Karmarkar-Karp algorithm, they take significantly longer. Although this is a less significant problem each year (machines become faster geometrically), it still makes these techniques impractical for some applications. The following representations take a different approach than those based on permuted lists. Instead of using a decoder based on a greedy algorithm creating partitions, the following representations are based directly on the Karmarkar-Karp algorithm.

Johnson et al. claim that the Karmarkar-Karp algorithm is "not based on local optimization or neighborhood structure at all," and cites this as an advantage of the algorithm over a simulated annealer working with representations of partitions ([6], p. 401). Instead, we have chosen to consider the Karmarkar-Karp algorithm as a greedy algorithm that operates on pairs of instance numbers to be placed in different partitions (see page 3 for a review of the algorithm). By thinking of sets of these pairs as instructions to the greedy algorithm, one can transform the deterministic Karmarkar-Karp algorithm into the decoder for a stochastic search procedure. We have considered three different ways of encoding these instructions, corresponding to different structurings and restrictions on the space of instructions.

7.1 Index-Based Difference Rules

This encoding represents each pair of numbers as a pair of indices into list of 'live' nodes sorted by node value. The ordinary behavior of the Karmarkar-Karp algorithm would then be represented as pairs of zeros. Since one node remains live at the end of the procedure and there is no choice of nodes for the last difference operation, there are $(n-2)$ pairs of indices, with the indices in the pair in position i assuming values in $[1,(n-i+1)]$ and $[1,(n-i)]$. The representation space is very large, at $O(n^3)$ possible encodings.

7.1.1 Operators

We have defined operators for this representation that are analogous to those we have used previously:

pair-mutate Given one solution, sets both indices in a randomly chosen rule to random legal values.

s_1	=	$(5,2)$	$(0,3)$	$(3,1)$	$(1,0)$	$(0,0)$	$(2,1)$
s_2	=	$(5,1)$	$(3,2)$	$(2,4)$	$(0,1)$	$(3,1)$	$(1,1)$
s'_1	=	$(5,2)$	$(3,2)$	$(2,4)$	$(0,1)$	$(0,0)$	$(2,1)$
s'_2	=	$(5,1)$	$(0,3)$	$(3,1)$	$(1,0)$	$(3,1)$	$(1,1)$

Table 8: An example of two-point crossover between difference rules. If '|' represents each crossover point, a two-point crossover between solutions s_1 and s_2 will produce solutions s'_1 and s'_2.

s_1	=	$(5,2)$	$(0,3)$	$(3,1)$	$(1,0)$	$(0,0)$	$(2,1)$
s_2	=	$(5,1)$	$(3,2)$	$(2,4)$	$(0,1)$	$(3,1)$	$(1,1)$
r	=	0	1	1	0	1	0
s'_1	=	$(5,2)$	$(3,2)$	$(2,4)$	$(1,0)$	$(3,1)$	$(2,1)$
s'_2	=	$(5,1)$	$(0,3)$	$(3,1)$	$(0,1)$	$(0,0)$	$(1,1)$

Table 9: An example of uniform crossover between difference rules. Using the random numbers r_i, where a value of one specifies a swap, a uniform crossover between solutions s_1 and s_2 will produce solutions s'_1 and s'_2.

two-point crossover Given two solutions s_1 and s_2, exchanges all rules between two randomly chosen but distinct indices. The resultant solutions s'_1 and s'_2 differ from the s_i only at locations i for $c_1 < i < c_2$, at which $s_{\{1,2\}} = s_{\{2,1\}}$. See table 8 for an example.

uniform crossover Given two solutions, swaps their corresponding rules at randomly chosen locations. Each location has a 50% chance of being chosen. See table 9 for an example.

7.1.2 Results

Due to the enormous number of solutions, only those methods based on local optimization are able to perform well using this representation (see figure 24). (Unfortunately, we were unable to try simulated annealing using the representations based on difference rules due to the extraordinary

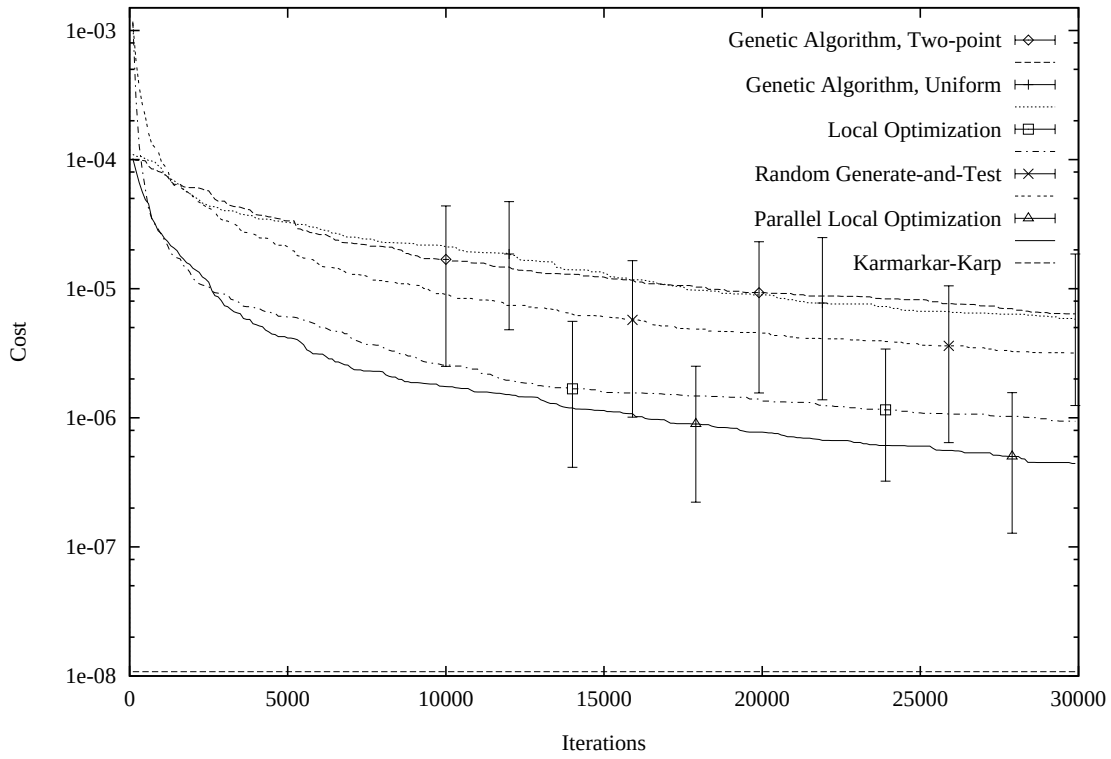

Figure 24: Performance of all algorithms using the index-based difference rules representation.

amount of computer time that would be required using our hardware and implementation.) The genetic algorithm fared equally poorly against parallel local optimization using each type of crossover. This may be because of interdependencies in the rules. For example, if a rule at the beginning of the encoding is changed, then the new value of the node remaining after the differencing step the rule specifies will probably be different. Because that node will be inserted in a different place in the ranked list of live nodes, other nodes will have different indices in that list than they would have had otherwise. This will change the meaning of all the rules in the encoding coming after the one which was mutated. While local optimization will perform only one mutation, a crossover will change many rules at once, causing even more of the previous rules to change their meaning. It may be this radical disruption that hinders the progress of the genetic algorithm.

In the case of local optimization, these interdependencies become an advantage. Combined with the large size of the representation space, they allow the algorithm to always find a way out of a potential local minimum. But for any other algorithm, the representation space of index-based difference rules seems too big and discontinuous to allow an effective search.

7.1.3 Seeding

As with permuted lists, we have conducted limited experiments with seeding the local optimization algorithm with the Karmarkar-Karp solution, this time represented using index-based difference rules. Since the decoder for this representation is a generalization of the Karmarkar-Karp algorithm, the Karmarkar-Karp solution is particularly easy to represent (all indices are zero, each differencing operation will involve the two greatest numbers). Since parallel local optimization performed better than the standard algorithm in our unseeded experiments, we have run both. Results (shown in figure 25) are excellent. When seeded with the Karmarkar-Karp solution for a 100-element problem, local optimization is able to improve on it by two orders of magnitude within 2,000 iterations (27 seconds), and almost three when given 30,000 iterations (seven and a half minutes). On a 500-element instance, the improvement is even more dramatic: four orders of magnitude within 3,000 iterations (nine and a half minutes). Parallel local optimization takes a few thousand iterations to saturate its population with mutations of the Karmarkar-Karp solution, but then makes quick progress, overtaking local optimization after 20,000 iterations. Similar results have also been obtained for additional problem instances. The behavior of these

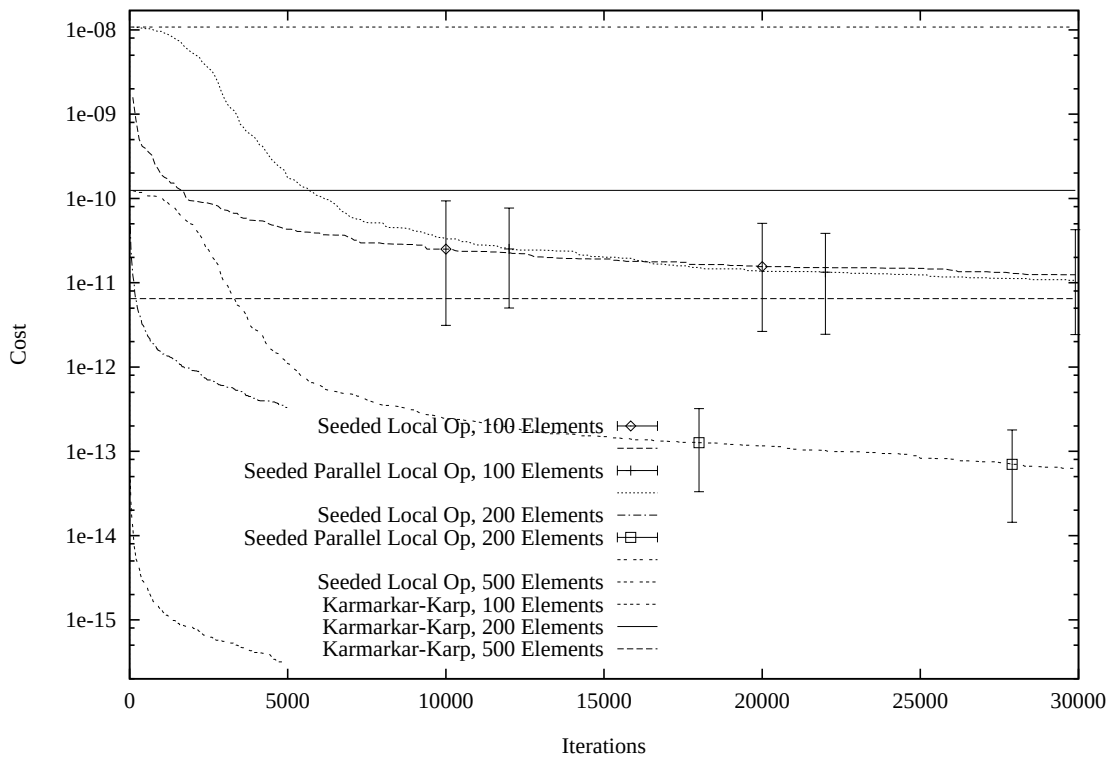

Figure 25: Performance of local optimization and parallel local optimization using the index-based difference rules representation seeded with the Karmarkar-Karp solution.

mixed algorithms is encouraging. Although they make great progress within the first 5,000 iterations, they often do not converge to a local minimum even after 30,000 iterations. They also seem to give results which decrease faster in relation to problem size than those of the Karmarkar-Karp algorithm. The space of difference rules may be unmanageably large for an unguided algorithm, but its structure seems to put excellent solutions within reach of local optimization when it is started from a good solution.

7.2 Weighted Index-based Difference Rules

Although productive, the search space of index-based difference rules is large and discontinuous. To help restructure the space and focus algorithms on the area around the Karmarkar-Karp solution, we have defined operators that use a skewed probability distribution for choosing indices. If r is a random number chosen from a uniform distribution on $[0, 1]$ and m is the maximum legal value for a particular index, then the corresponding index is defined by $r^4 m$. This distribution is used when creating random solutions. When mutating a given index, if its current value is non-zero, its next value is chosen by the same distribution, otherwise its next value is chosen as $1 + r^3(m-1)$, to assure a different new value. This weighting scheme makes low indices much more likely (32% zeros under $r^4 m$, 21% ones under $1 + r^3(m-1)$), and should help focus algorithms on what we would consider the more productive areas in the search space.

7.2.1 Results

Results using these weighted operators are, in general, a full two orders of magnitude better than those obtained with a uniform distribution of indices (see figure 26). The random generate-and-test algorithm does much better than before, indicating that the reshaping of the solution space is successful. As before parallel local optimization is the most effective algorithm, while the genetic algorithm fares poorly with both pmx and uniform crossover.

7.2.2 Seeding

While still very effective, seeding is not quite as successful with the weighted representation as it was previously (see figure 27). It seems that once an algorithm has found a good region of the search space, no pressure need be exerted to focus its attention there, and indeed, such restrictions can hinder its discovery of good solutions.

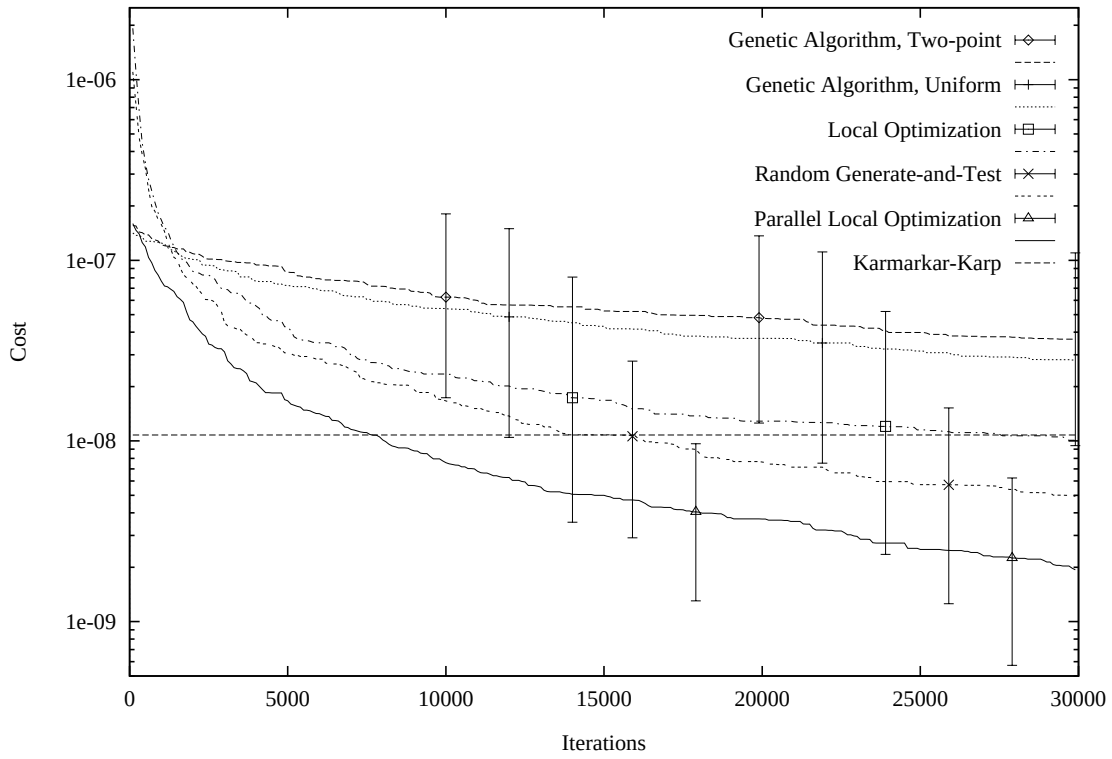

Figure 26: Performance of all algorithms using the weighted index-based difference rules representation.

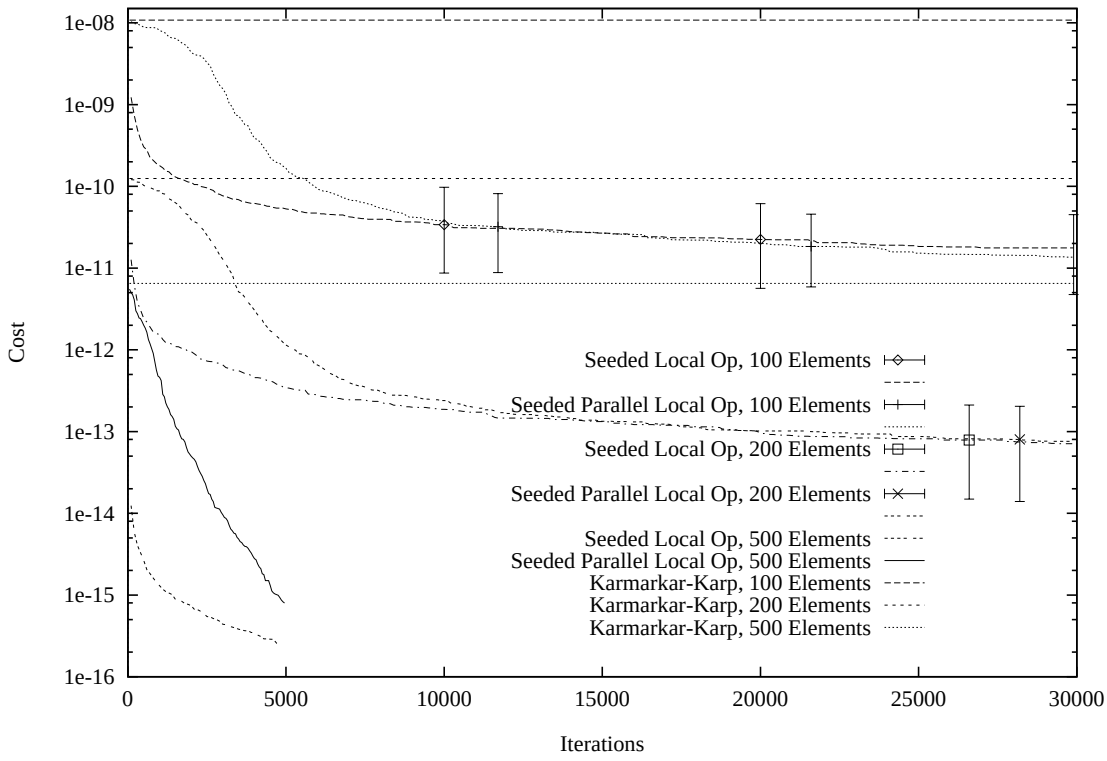

Figure 27: Performance of local optimization and parallel local optimization using the weighted index-based difference rules representation seeded with the Karmarkar-Karp solution.

7.3 Single-Index Difference Rules

A more direct way of focussing the search of our stochastic algorithms is to decrease the size of the representation space. (In contrast to our experiments with permuted lists, we are now using a fixed decoder and limiting the search space by restricting the representation and its operators.) We have implemented a single-index difference rules representation, which functions exactly as our previous weighted index-based difference rules encoding, except that only one number in each difference pair is specified by a particular solution; the other number of each pair is assumed to be always zero, referring to the currently greatest 'live' node.

7.3.1 Results

As with the weighting of operators, this further restriction of the search space improves the performance of the search algorithms (see figure 28). In general, results were about one order of magnitude better than the weighted index-rules representation (about three orders of magnitude better than the original unrestricted version). Even without seeding, all algorithms quickly find solutions better than that found by the Karmarkar-Karp algorithm. Local optimization performs worse than before, perhaps due to the smaller search space and a reduction in the effects of rule interdependence (recall section 7.1.2). The genetic algorithm again fares poorly, although it shows signs of continued gradual improvement when using uniform crossover.

7.3.2 Seeding

Again, seeding standard and parallel local optimization was very effective (see figure 29). Parallel local optimization does particularly well relative to the standard algorithm.

8 Prepartitioning

While difference rules representations modify the inner workings of the Karmarkar-Karp algorithm in order to produce a solution by specifying the numbers to difference, the prepartitioning representation works by changing the input to the algorithm. A solution consists in a list of n labels, each of which specifies a 'prepartition' into which the corresponding instance number is to be put. Up to n different prepartitions may be specified,

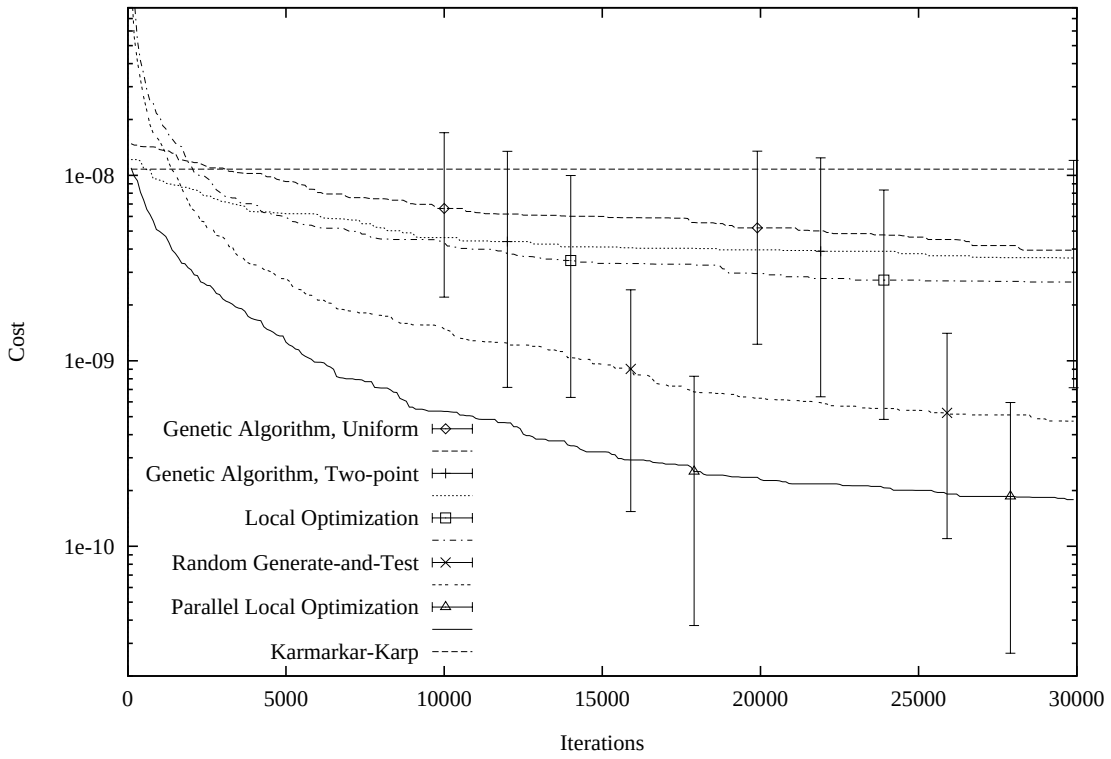

Figure 28: Performance of all algorithms using the single-index difference rules representation.

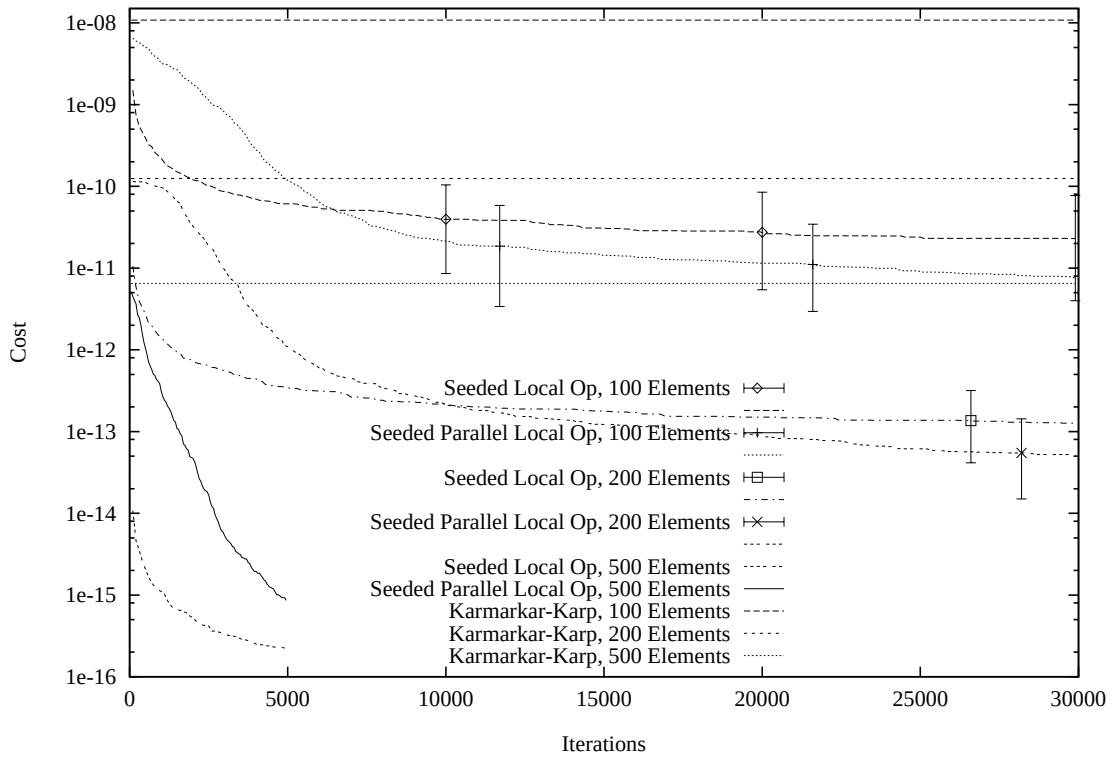

Figure 29: Performance of local optimization and parallel local optimization using the single-index difference rules representation seeded with the Karmarkar-Karp solution.

yielding a representation space of size n^n (much larger than the others we have considered). The sums of the prepartitions are then used as instance numbers for a new instance of number-partitioning, and given as input to the Karmarkar-Karp algorithm, which, in effect, treats each prepartition as a block of numbers to be kept in the same partition. Its solution to the problem of partitioning the prepartitionings can then be used to produce a partitioning for the original instance numbers. While the Karmarkar-Karp algorithm can only construct one solution to any given instance, prepartitioning attempts to transform a given instance of number-partitioning into another equivalent problem that can be solved better.

8.1 Operators

We have defined the same one-swap mutator, two-point crossover, and uniform crossover as with the direct representation of partitionings (recall section 5.1). The only difference is that instead of ranging over $(0,1)^n$, the encoding now has n possible target partitions and so ranges over $(1,\ldots,n)^n$.

8.2 Results

Although prepartitioning is perhaps the simplest technique we have considered, after the direct representation of partitionings, it performs the best (see figure 30). Note that, given a perfectly uniform distribution of random numbers, a random prepartitioning solution will put each number in its own prepartition, thereby constructing the Karmarkar-Karp solution. This explains the excellent performance of the random generate-and-test algorithm, which rapidly finds solutions of a quality similar to those of parallel local optimization. As with other representations based on the Karmarkar-Karp algorithm, the genetic algorithm makes little progress with either crossover.

8.3 Seeding

Results from seeding standard and parallel local optimization are poor compared to those of the difference rules representations (see figure 31). Local optimization seems prone to entrapment in local minima. And surprisingly, parallel local optimization does little better than the standard algorithm.

In general, the more restricted the search space, the better the performance of a single algorithm. When starting from the Karmarkar-Karp algorithm's solution, however, guidance toward that solution area is unnecessary and counter-productive.

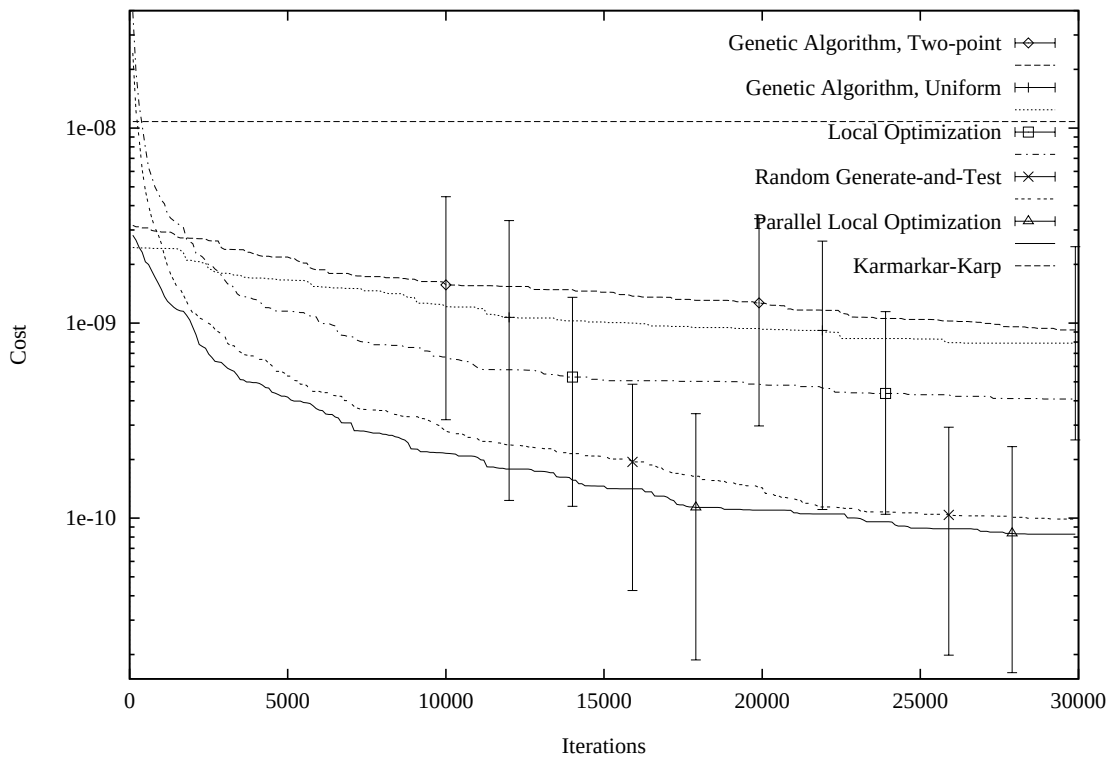

Figure 30: Performance of all algorithms using the prepartitioning representation.

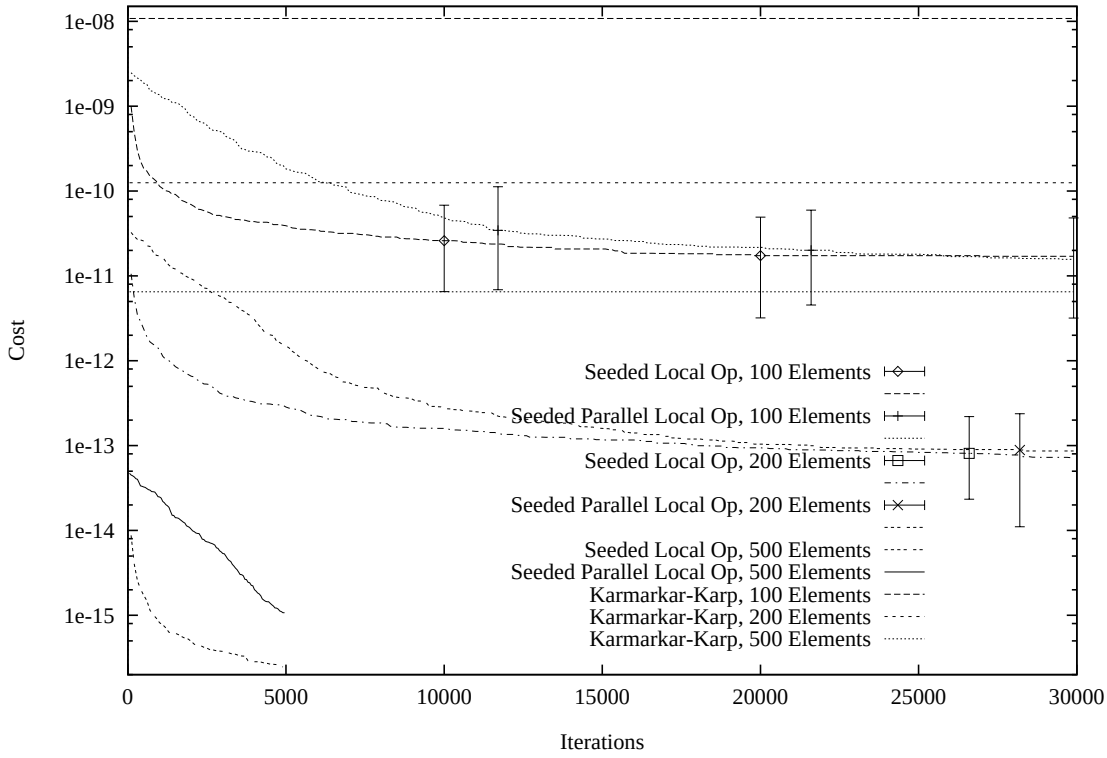

Figure 31: Performance of local optimization and parallel local optimization using the prepartitioning representation seeded with the Karmarkar-Karp solution.

representation	random	local
index rules	3.18e-6	9.40e-7
weighted index rules	4.92e-9	1.01e-8
single index rules	4.73e-10	2.66e-9
prepartitioning	9.90e-11	4.08e-10

Table 10: The results of the random generate-and-test and local optimization algorithms using each of the representations that require a Karmarkar-Karp-based decoder (30,000 iterations).

8.4 Summary of Results

As with the representations based on permuted lists, a comparison of techniques based on the Karmarkar-Karp algorithm may help highlight the qualities of each.

8.4.1 Representation Spaces

Table 10 is analogous to table 7, it compares the performance of the random generate-and-test algorithm with that of local optimization in each of the representations under consideration. As our previous results have indicated, as the solution space becomes more restricted, a random solution is more likely to be good, but local optimization will have a more difficult time improving it.

8.4.2 Single Algorithms

Figures 32–35 show the performance of each algorithm using the difference rules representations and prepartitioning. The performance of the random generate-and-test algorithm shows the relative restrictiveness of each representation space (figure 32). This basic hierarchy of performance holds for all the search algorithms we have considered. Parallel optimization was the most effective algorithm overall (figure 34), although random generate-and-test was quite competitive. The genetic algorithm makes progress when using the unrestricted index-based difference rules representation, although its performance is quite poor when compared to the same algorithm without crossover (i.e., parallel local optimization).

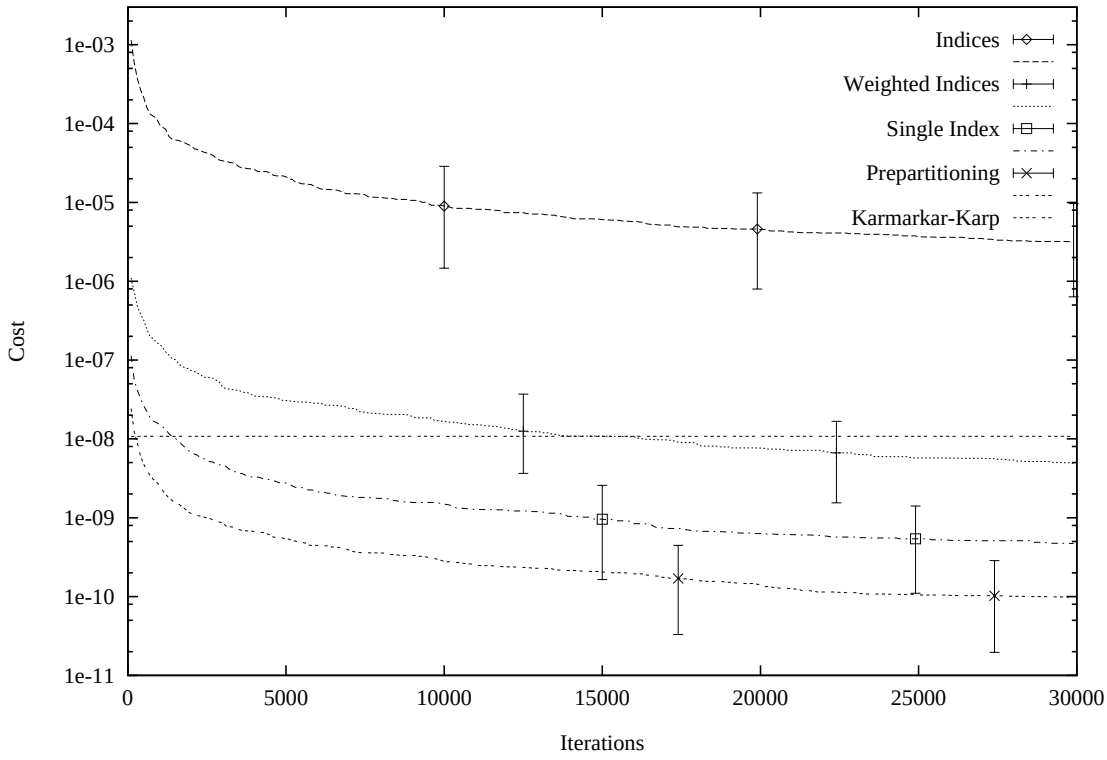

Figure 32: The random generate-and-test algorithm using many different representations. The density of good solutions in each representation increases as the representation space shrinks and the operators are more restricted.

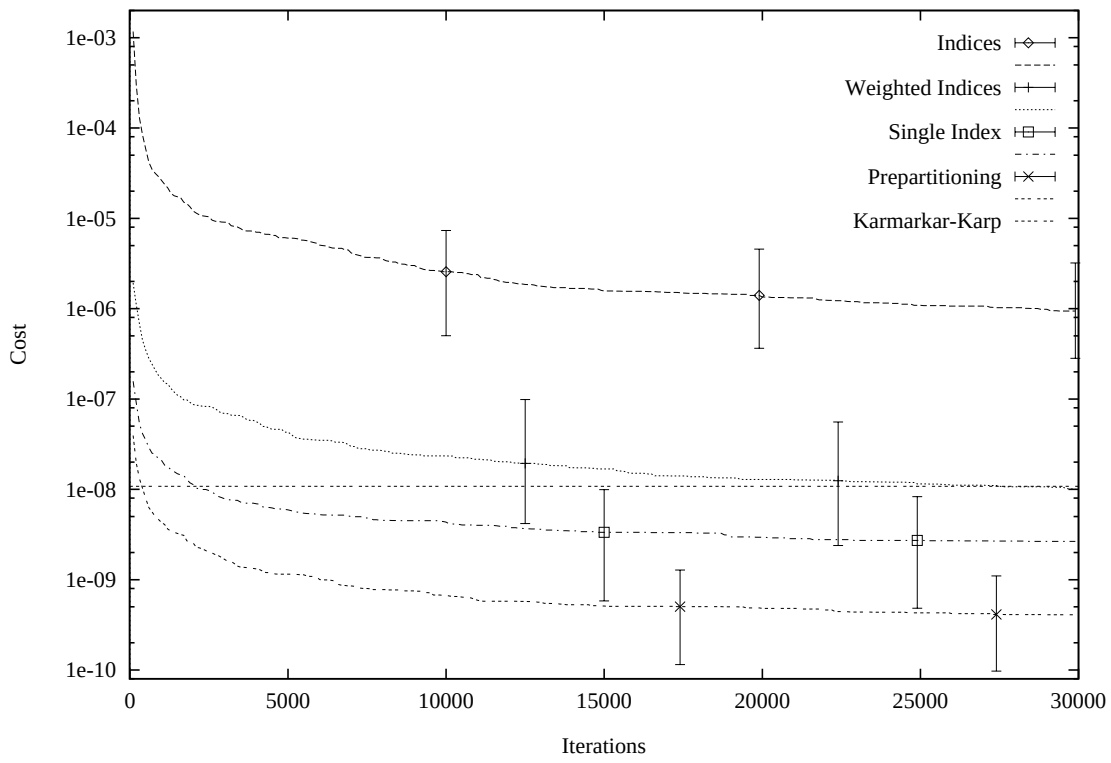

Figure 33: Local optimization using many different representations. The enormous size of the search space when using difference rules representations prevents the algorithm from becoming trapped in local minima.

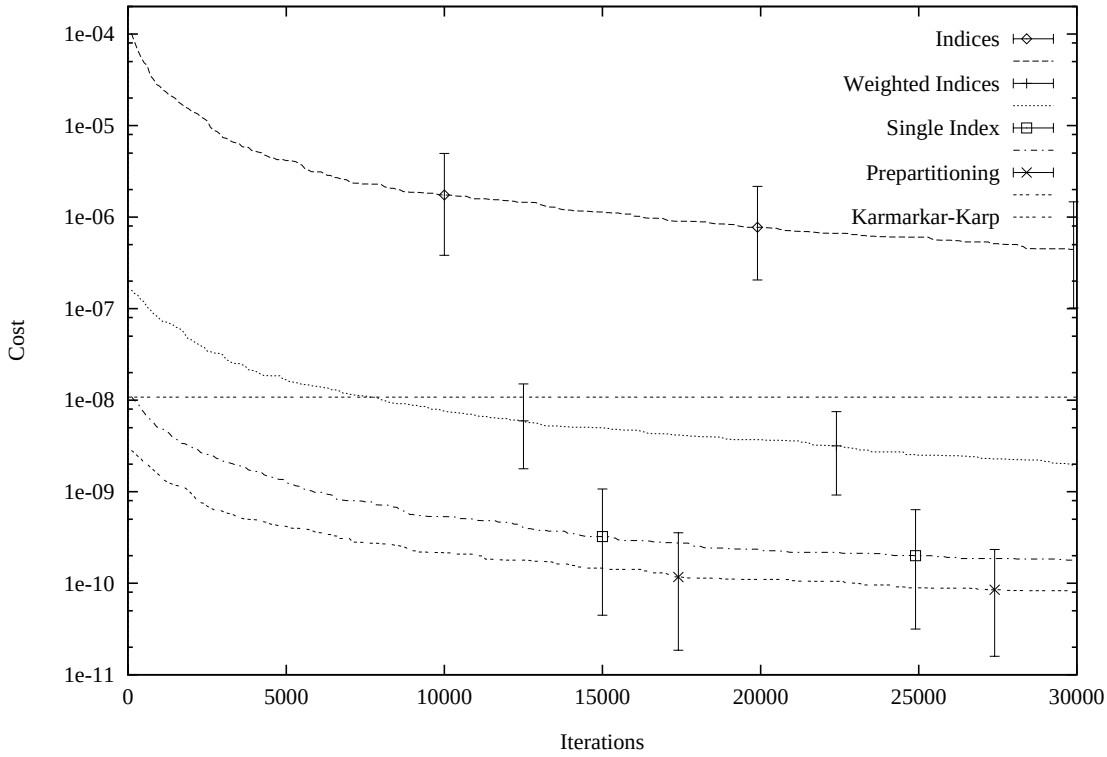

Figure 34: Parallel local optimization using many different representations.

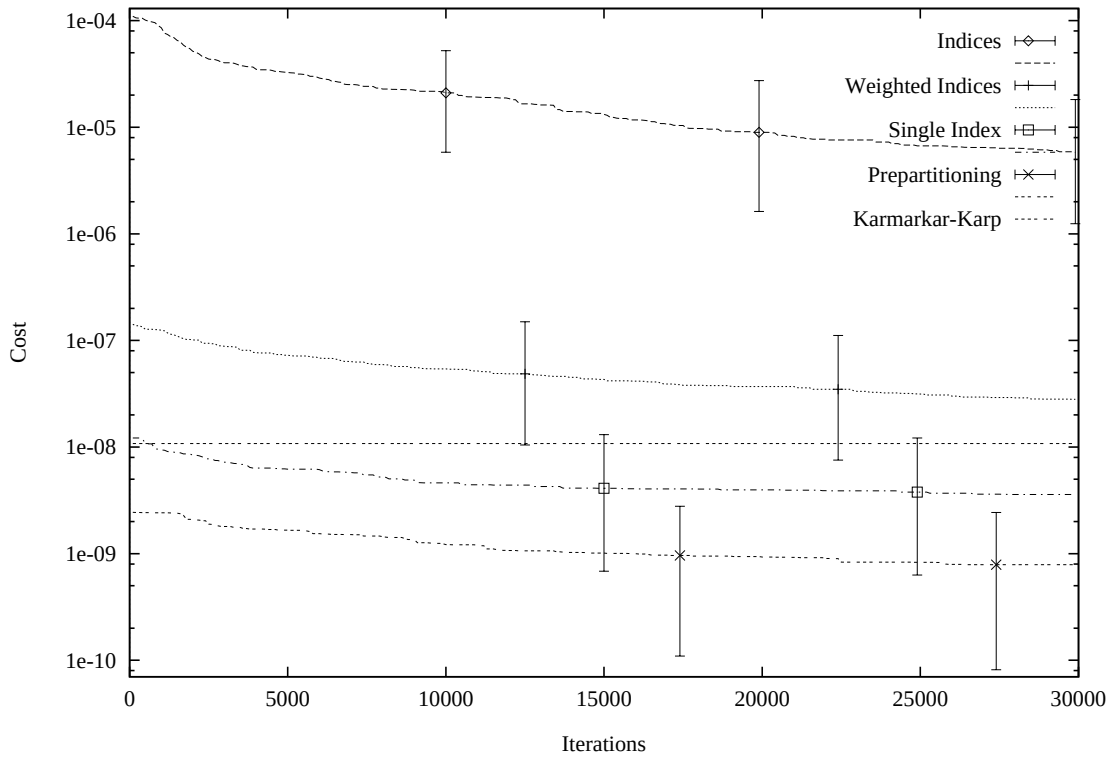

Figure 35: The genetic algorithm using many different representations.

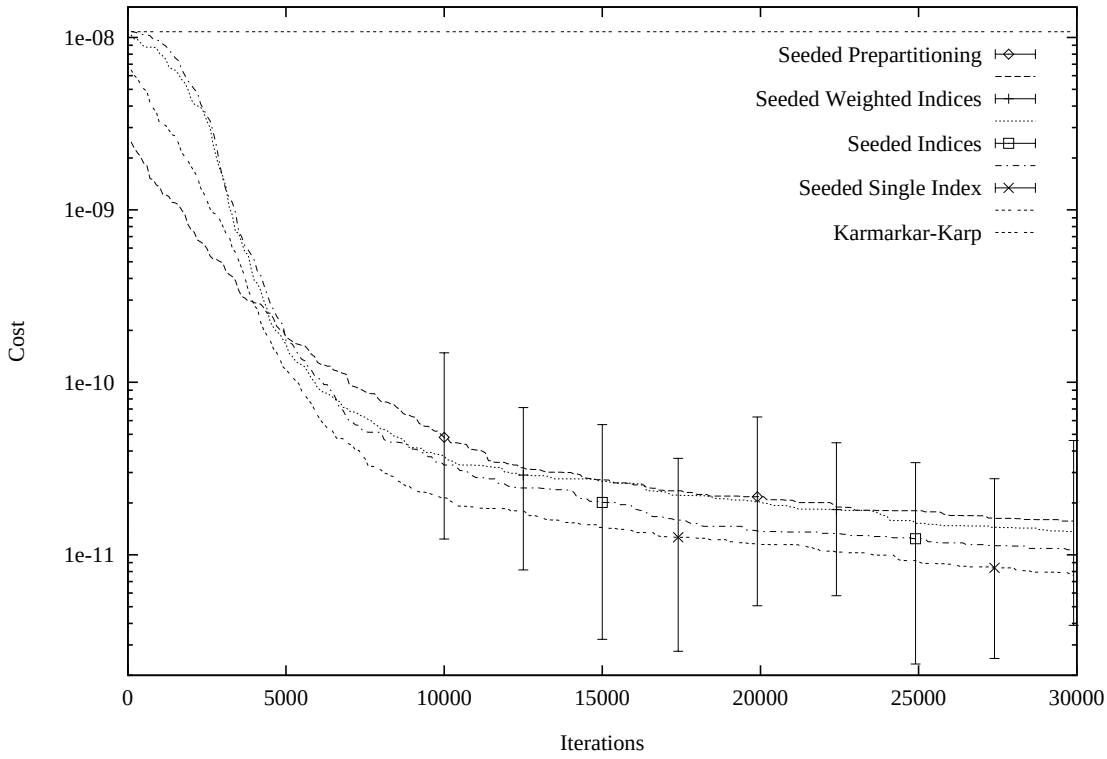

Figure 36: Performance of parallel local optimization using the prepartitioning, weighted difference rules, plain difference rules, and single-index difference rules representations, all seeded with the Karmarkar-Karp solution.

8.4.3 Seeded Algorithms

Seeding was quite effective in all difference rules representations, as well as prepartitioning (figure 36). As noted earlier, representations that were successful with unseeded algorithms because of the extra focus they provided on the area of the search space around the Karmarkar-Karp solution tended to be less successful when used with seeding. The restrictiveness of the representation kept the algorithms from trying a variety of solutions (note the larger standard deviation in results when using the unrestricted index-based difference rules).

8.5 Overall Summary

Figure 37 shows the performance of all algorithm and representation combinations. The dominant slope of the plot shows that representation is clearly the most important factor in determining performance, and that it plays an even larger role than the choice of search algorithm.

9 Conclusion

By applying the genetic algorithm community's notion of a solution encoding to other algorithms, a difficult problem previously thought ill-suited to stochastic search methods can be effectively solved. Using permuted lists and a greedy decoder, a simulated annealer can, in thirty minutes, find a solution superior to that constructed by the Karmarkar-Karp method. Using prepartitioning and the Karmarkar-Karp algorithm, one can surpass the Karmarkar-Karp solution in seconds, and improve on it by orders of magnitude within three minutes. And by seeding our algorithms, one can achieve excellent results immediately.

Our new approach to number-partitioning has shown that:

1. Stochastic search algorithms can find better solutions to instances of number-partitioning than the plain Karmarkar-Karp algorithm if given enough time (half an hour in the case of simulated annealing using permuted lists and the greedy decoder, two and a half minutes for random generate-and-test or local optimization using prepartitioning).

2. Given the right representation (such as index-based difference rules), stochastic search around the Karmarkar-Karp solution can be very effective.

3. The Karmarkar-Karp algorithm itself can be used as a starting point for constructing an effective stochastic search space (such as the difference rules and prepartitioning representations).

Our comparisons of representation spaces have also yielded some insights which may applicable to problems other than those related directly to number-partitioning:

4. When using methods based on local optimization, such as simulated annealing, it can be more important to have a continuously structured space than one in which all solutions are exceptionally good (e.g., permuted lists and the plain greedy decoder).

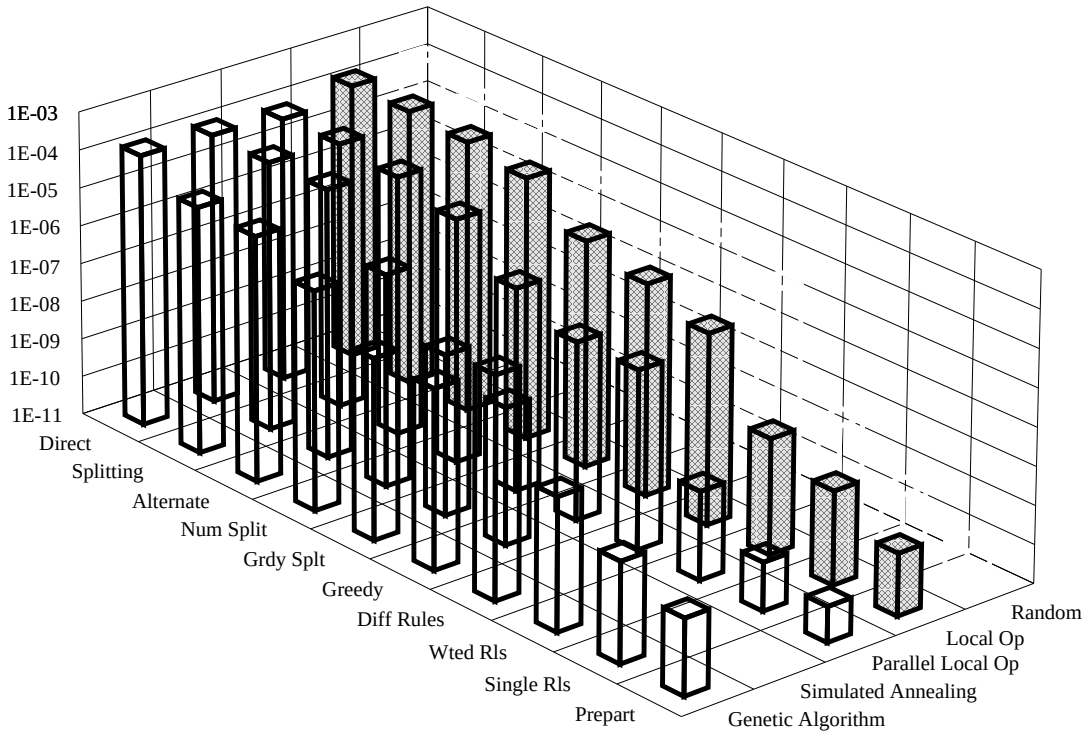

Figure 37: Final solution cost for each of the five algorithms using each of the ten representations (six in the case of simulated annealing). The representation plays an even more important role in determining performance than the particular stochastic algorithm.

5. With some representations, a genetic algorithm affords no advantage over parallel local optimization. Parallel local optimization is a good optimizer in its own right, often much better than standard local optimization.

And above all:

6. The choice of the representation space in which a stochastic search algorithm roams is more important than the choice of the algorithm itself.

Since number-partitioning has been acknowledged as a difficult problem, and one to which many others can be reduced, this work may have application to many other optimization problems. A key advantage of the methods we have investigated is that they take very little domain knowledge into consideration; they perform a relatively blind search. They are also easy to implement. Thus, these techniques may be easily applied to new problems.

Acknowledgements

This work was undertaken with generous advice, astute guidance, and many helpful suggestions from Stuart Shieber, Joe Marks, and Tom Ngo. Discussions with Jon Christensen and the Harvard Animation Group were also fruitful. Many thanks to David S. Johnson of AT&T Bell Labs for generously and speedily sharing his test instances. Apologies are due to Christopher Marks, who has grown up without his father, and to Stephen Frug, who can barely recognize his roommate. Finally, thanks to John A. Wheeler and Kate Sutherland for inspiration.

References

[1] Lawrence Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1991.

[3] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[4] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, second edition, 1992.

[5] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6):865–892, November-December 1989.

[6] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, May-June 1991.

[7] Donald R. Jones and Mark A. Beltramo. Solving partitioning problems with genetic algorithms. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 442–449, San Mateo, CA, July 1991. University of California, San Diego, Morgan Kaufmann.

[8] Scott Kirkpatrick Jr., C.D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

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

[10] Narenda Karmarkar, Richard M. Karp, George S. Lucker, and Andrew M. Odlyzko. Probabilistic analysis of optimum partitioning. *Journal of Applied Probability*, 23:626–645, 1986.

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

[12] G. H. Sasaki and B. Hajek. The time complexity of maximum matching by simulated annealing. *Journal of the Association for Computing Machinery*, 35:387–403, 1988.

[13] A. Shamir. On the cryptocomplexity of knapsack systems. In *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, pages 118–129, New York, 1979. Association for Computing Machinery.

[14] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989. Morgan Kaufmann.