Approximations for the number partition problem

Lev Kruglyak[1]

levkruglyak@college.harvard.edu[1]

1 Introduction

The number partition problem (NumberPartition) is a classic example of an NP-complete problem; given some sequence of n numbers $A = (a_1, a_2, \ldots, a_n)$, we would like to find some sequence of "signs", $S = (s_1, s_2, \ldots, s_n)$ with each $s_i = \pm 1$ such that the following quantity is minimized:

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

This is called a *residue* of A. Notice that this corresponds to partitioning A into two disjoint subsets and trying to minimize the difference in the total sums of both sets, justifying the name "number partition". First we'll show that even though NumberPartition isn't solvable in polynomial time, it is solvable in pseudo-polynomial time.

1.1 Dynamic programming solution

To find a dynamic programming solution to this problem, we will first need some sort of recursion. Let S be a set of (positive) numbers, and for any positive k and Σ, let $SS(S, k, \Sigma)$ be the function which returns true if there is a subset of S of size k which sums to Σ. Then clearly,

$$NP(S) = \max\{0 \le m \le \lceil \Sigma/2 \rceil \mid \exists k, SS(S, k, m)\}$$

where Σ is the total sum of S and $NP(S)$ is the minimal residue of S. SS then has a very simple recursion,

$$SS(S, k, \Sigma) = SS(S, k-1, \Sigma) \vee SS(S, n-1, \Sigma - S_{n-1})$$

with base cases $SS(S, 0, \Sigma) =$ `false` and $SS(S, n, 0) =$ `true`. We can efficiently calculate the values of this recurrence using dynamic programming. Let $D_{i,j}$ be an array of size $(n+1) \times (\lceil \Sigma/2 \rceil + 1)$. We could like $D_{i,j} = SS(S, i, j)$. Based on the recursion, our updating scheme is:

$$D_{i,j} = \begin{cases} D_{i-1,j} & S_{i-1} > j \\ D_{i-1,j} \vee D_{i-1,j-S_{i-1}} & \text{otherwise} \end{cases}$$

Updating this array in the order of calculating j rows first, then incrementing i and calculating the next j-th row, we are able to calculate the entire array in $O(n\Sigma)$. All this time, we can keep track of the maximal j for which $SS(S, i, j)$ is true, and this j is the solution to the number partition problem.

1.2 The Karmarkar-Karp algorithm

A good deterministic heuristic for NumberPartition is the Karmarkar-Karp algorithm. This algorithm uses *differencing*, the idea being to take two elements a_i, a_j from the set A and replace the larger one with $|a_i - a_j|$, and replace the smaller one by zero. Finally, when only a single nonzero element is left, we have the desired residue. (If we want to actually calculate the solution, we can keep track of which elements we chose and which one was larger.)

To efficiently implement this algorithm, we can represent A as a binary max-heap or priority queue. At each stage, we pop off the maximal two elements and insert their difference back into the heap. Popping the heap is constant time complexity and inserting is $O(\log n)$, so since we run n iterations, we get an overall time complexity of $O(n \log n)$, assuming arithmetic operations are constant time. This turns out to be one of the most efficient algorithms, but it doesn't always yield the optimal solution.

2 Randomized heuristic algorithms

To get some improvement on the Karmarkar-Karp algorithm, we can try some randomized algorithms. These involve starting at some random solution and repeatedly altering it based on some heuristic to get as close to the optimal solution as possible. We do this for a fixed number of iterations which determines both the accuracy and the time complexity.

2.1 Repeated random

In the most basic algorithm, we simply keep generating random solutions to the problem and return the one which one has the smallest residue. In pseudocode,

Algorithm 1: Repeated-Random

```
S ← a random solution
for i = 1 ... N do
    S' ← another random solution
    if residue (S') < residue (S) then
        S ← S'
    end
end
return residue (S)
```

2.2 Hill climbing

In the hill climbing approach, we start with a random solution, and make random moves to attempt to bring it closed to an optimal solution. This is a bit faster that the repeated random approach because we don't have to regenerate the entire solution, only modify it slightly. Here, a random move means In pseudocode,

Algorithm 2: Hill-Climbing

```
S ← a random solution
for i = 1 ... N do
    S' ← random_move(S)
    if residue (S') < residue (S) then
        S ← S'
    end
end
return residue (S)
```

A problem with this approach is that we might start at a solution which is very far from optimal, so it would be unlikely that we could every reach it. To address this we can present a variant of this algorithm known as *simulated annealing*.

2.3 Simulated annealing

In this approach, we allow the algorithm to make a potentially bad random move with a certain probability, and decrease this probability as time goes on. In pseudocode, this algorithm looks like,

Algorithm 3: Hill-Climbing

```
S ← a random solution
S'' ← S
for i = 1 ... N do
    S' ← random_move(S)
    if residue (S') < residue (S) then
        S ← S'
    else
        S ← S' with probability
          exp(−(residue(S')−residue(S))/T(i))
    end
    if residue (S) < residue (S'') then
        S'' ← S
    end
end
return residue (S)
```

Here $T(i)$ is called the *cooling schedule*, and the assignment suggested that we set it to $T(i) = 10^{10}(0.8)^{\lfloor i/300 \rfloor}$. We found that this worked for our purposes, so we didn't choose to modify this function.

2.4 Prepartitioning

When running these random algorithms on the naive way to represent solutions, i.e. as sequences of ± 1, we don't actually get any better results than Karmarkar-Karp, in fact we get significantly worse results even with a large number of iterations. This isn't a fault of the algorithm, but rather with the way we're representing the state space of solutions.

A better way to represent the state space of solutions is a scheme called prepartitioning. This represents the solution as a sequence $P = (p_1, \ldots, p_n)$ of numbers between 1 and n with the idea being that if $p_i = p_j$, then i and j must have the same sign in the solution. We can transform such a solution into a regular solution by first constructing the set S', defined as

$$S'_j = \sum_{j=p_i} S_i.$$

We then run Karmarkar-Karp on this set to get the residue, and using this we can reconstruct the sign representation of the solution.

3 Experimental results

Finally, let's perform some experiments to determine which of these algorithms is works the best. We generated 50 instances of the problem, with sequences of 100 integers chosen uniformly from the interval $[1, 10^{12}]$. For the random heuristic algorithms, we used $N = 25,000$ iterations. The averages of our results are shown in the following table.

Table 1. NumberPartition algorithms

Algorithm	Residue	w/ partitioning
Karmarkar-Karp	215314	
Repeated random	476367170	172
Hill climbing	659368703	1202
Simulated annealing	537855299	215

As we can see, Karmarkar-Karp gives a decent solution to the problem, with the main benefit being that it is blazingly fast compared to the others. (Not asymptotically of course, but for these relatively small scales) The random algorithms perform awfully without the benefits of the prepartitioning scheme, but when we add in prepartitioning we get the best results yet. Among these, it seems that the repeated random and simulated annealing algorithms perform the best, with very close averages.

3.1 Improving on these methods

Based on our experiments, it seems that the random algorithms works best when the numbers in the set are uniformly distributed. This suggests that we could first run Karmarkar-Karp for a fraction of the iterations, and then switch to a random algorithm once we've smoothed out the data sufficiently.