Analysis of Heuristics for the Number Partition Problem

Eliel Dushime

April 2022

1 Abstract

Our goal is to implement a number of heuristics for solving an NP-complete problem, the Number Partition problem. We make use of two different representations (standard representation and a prepartitioning representation). Our aim will entail comparing results from a variety of algorithms (the Karmarkar-Karp algorithm and a variety of heuristic algorithms applied to random input sets: Repeated Random, Hill Climbing, and Simulate Annealing heuristics). The entire code is found in partition.py, with inputfile as a text file that contains a list of 100 (unsorted) integers, one per line. The output is the residue of this list of 100 integers. The submitted code is executed with the run command:

./partition flag algorithm inputfile

where the flag argument is used for debugging purposes (uses 0 by default), and the algorithm argument designates an integer code word for each specific heuristics algorithm.

2 Introduction

The number partition problem can be stated as follows: given an input $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers, produce a sequence $S = (s_1, s_2, \ldots, s_n)$ of signs $s_i \in \{-1, +1\}$ such that the residue

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

is minimized. Analogously, we are trying to split the integers in A into two sets A_1 and A_2 such that the sums of A_1 and A_2 are as similar as possible, with the residue being the absolute value of the difference between the sets. Even though this problem is NP-complete, there does exist a pseudo-polynomial time dynamic programming algorithm for it.

3 DP solution to the Number Partition problem

Given a sequences of n numbers $A = a_1, a_2, \ldots a_n$, the Number Partition problem is to decide whether A can be divided into two subsets A_1 and A_2 of equal sums.

We solve the problem dynamically as follows:

Pseudocode:

1. Calculate the Sum of numbers in given sequence of numbers. If Sum is odd then return 0.

2. We create a 2D-array D of size $Sum/2 + 1$ by $n + 1$, where sum is the sum of all the numbers in the sequence. The row of D represents a sum while its column represents the size of an array. The entries in the 2D-array will be bits (0 or 1). The value of $D[i][j]$ will be 1 if there exists a subset of size less than or equal to j and sum equal to i.

3. Populate the 2D-array in a bottom-up fashion as follows:

- Fill in the base cases: $D[0][i] = 1$ and $D[i][0] = 0$ if $i < Sum$
- Recursively build the rest of the array from $i = j = 1$ with $D[i][j] = 1$ if $D[i][j-1] = 1$ or $D\Big[i - A[j-1]\Big][j-1] = 1$ (for $i \geq A[j-1]$)

4. We return $D[Sum/2][n]$

This algorithm takes $n * b$ with b the Sum

4 Karmarkar-Karp implementation

One deterministic heuristic for the Number Partition problem is the Karmarkar-Karp algorithm, or the KK algorithm. This approach uses differencing. The differencing idea is to take two elements from A, call them a_i and a_j , and replace the larger by $|a_i - a_j|$ while replacing the smaller by 0. The intuition is that if we decide to put a_i and a_j in different sets, then it is as though we have one element of size $|a_i - a_j|$ around. An algorithm based on differencing repeatedly takes two elements from A and performs a differencing until there is only one element left; this element equals an attainable residue. (A sequence of signs si that yields this residue can be determined from the differencing operations performed in linear time by two-coloring the graph (A, E) that arises, where E is the set of pairs (a_i, a_j) that are used in the differencing steps). For the Karmarkar-Karp algorithm suggests repeatedly taking the largest two elements remaining in A at each step and differencing them.
To implement the Karmarkar algorithm, we will need a way to extract the two largest numbers from the input sequence at each step of the algorithm. A max heap therefore is a good strategy. Applying max heapify to get the two largest numbers from the heap will take $log(n)$. Since there will be at most n steps to implement differencing, we get overall run time of $O(n \log(n))$.

Karmarkar-Karp algorithm as a starting point for the randomized algorithms
This Karmarkar-Karp algorithm gives a good approximation for the actual residue. One may use it as a starting point to optimize the other algorithms. We would constrain the starting random solution to have a residue less than that given by the Karmarkar-Karp algorithm.This allow better searches of the state space and avoid bad neighbors.

Representations of Solutions
The standard representation of a solution is simply as a sequence S of $+1$ and -1 values. A random solution can be obtained by generating a random sequence of n such values. Thinking of all possible solutions as a state space, a natural way to define neighbors of a solution S is as the set of all solutions that differ from S in either one or two places. This has a natural interpretation if we think of the $+1$ and -1 values as determining two subsets A_1 and A_2 of A. Moving from S to a neighbor is accomplished either by moving one or two elements from $A1$ to $A2$, or moving one or two elements from A_2 to A_1, or swapping a pair of elements where one is in A_1 and one is in A_2.

A random move on this state space can be defined as follows. Choose two random indices i and j from $[1, n]$ with $i = j$. Set s_i to $-s_i$ and with probability $1/2$, set s_j to $-s_j$.

An alternative way to represent a solution called prepartitioning is as follows. We represent a solution by a sequence $P = p_1, p_2, ..., p_n$ where $p_i \in 1, ..., n$. The sequence P represents a prepartitioning of the elements of A, in the following way: if $p_i = p_j$, then we enforce the restriction that a_i and a_j have the same sign. Equivalently, if $p_i = p_j$, then a_i and a_j both lie in the same subset, either A_1 or A_2.
Additionally, we compare the performance of the Karmarkar-Karp algorithm to three different heuristics that make use of randomness:

Repeated Random: The Repeated Random heuristic is a simple approach that involves generating random solutions repeatedly and selecting the one with the best residue. The residue represents the absolute difference between the sums of the two partitions in the Number Partition Problem. By generating multiple random solutions and selecting the one with the lowest residue, this heuristic aims to find an optimal or near-optimal solution through random exploration. However, it does not utilize any optimization techniques to improve the solution iteratively.

Hill Climbing: Hill Climbing is a local search algorithm that starts with an initial solution and continuously moves to a neighboring solution that improves the objective function value (in this case, the residue). It explores the solution space by iteratively considering the neighbors of the current solution and moving to the one with the lowest residue. This process continues until a local optimum is reached, where no neighboring solution yields a better residue. Hill Climbing is relatively simple to implement and can quickly converge to a local optimum. However, it may get stuck at suboptimal solutions and fail to explore the entire solution space.

Simulated Annealing: Simulated Annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It aims to overcome the limitations of hill climbing by allowing occasional uphill moves to escape local optima. Simulated Annealing starts with an initial solution and iteratively explores neighboring solutions. Unlike hill climbing, Simulated Annealing

accepts worse solutions with a certain probability based on a cooling schedule. The probability of accepting a worse solution decreases as the algorithm progresses, mimicking the cooling process. This allows the algorithm to escape local optima and search for globally better solutions. As the algorithm proceeds, the probability of accepting worse solutions decreases, focusing the search on better solutions. Simulated Annealing can effectively explore the solution space and has a higher chance of finding better solutions compared to Hill Climbing. However, it requires careful tuning of the cooling schedule to balance exploration and exploitation.

These heuristics provide different trade-offs between solution quality and computational efficiency. Repeated Random is a simple and fast approach but may not find optimal solutions consistently. Hill Climbing improves on Repeated Random by performing local exploration but can get trapped at local optima. Simulated Annealing offers a more robust approach by allowing uphill moves to escape local optima, but it requires careful parameter tuning. Selecting the appropriate heuristic depends on the specific requirements of the problem and the trade-offs between solution quality and computational resources available.

5 Experiments

You run experiments on sets of 100 integers, with each integer being a random number chosen uniformly from the range [1, 1012]. We write a routine that takes three arguments: a flag, an algorithm code (see Table 1), and an input file. We'll run typical commands to compile and execute the code, as in Python:

./partition flag algorithm inputfile

The flag is meant to provide flexibility with testing, debugging, or extensions. The algorithm argument is one of the values specified in the Table as "algorithm code". We assume the inputfile is a list of 100 (unsorted) integers, one per line. The desired output is the residue obtained by running the specified algorithm with these 100 numbers as input.

Heuristics			
Algorithm code	Algorithm	Mean Residues	Mean Runtimes
0	Karmarkar-Karp	314347	
1	Repeated Random	276465755	31
2	Hill Climbing	383478472	14
3	Simulated Annealing	214563773	17
11	Prepartitioned Repeated Random	311	1457
12	Prepartitioned Hill Climbing	769	1334
13	Prepartitioned Simulated Annealing	267	1315

6 Results and Discussion

To evaluate the performance of different heuristics for solving the Number Partition problem, we conducted a comprehensive analysis involving 50 test trials. The trials were carried out on ran-

domly generated numbers ranging from 1 to 10^{12}. Our objective was to compare the effectiveness of various heuristics, including the KK algorithm, repeated random, hill climbing, and simulated annealing.

For each trial, we calculated the residue using the KK algorithm as a benchmark. We employed both sequence-based and partition-based solutions and collected data for a thorough analysis. Tha table from the last section showcases the mean residues obtained from the different heuristics. We found that the partition-based solutions consistently outperformed the sequence-based solutions. The initial residues obtained from randomly chosen partition solutions were notably superior to those from sequence solutions. This outcome can be attributed to the flexibility inherent in the prepartition approach, where numbers are grouped together before applying the KK algorithm. Consequently, prepartition-based solutions yielded final residues that were approximately a million times better than the sequence-based residues.

It is crucial to consider the computational complexity associated with prepartitioning. As the prepartition approach necessitates running the KK algorithm to find the residue, it incurs higher runtimes compared to the sequence-based approach. The result table showcases the average runtimes of the heuristics, revealing that prepartitioning increased the runtime by approximately 100 times compared to the sequence-based methods. These numbers are based on our own testing and provide a realistic understanding of the computational costs involved.

For both the partition-based and sequence-based solutions, simulated annealing consistently performed the best, followed by repeated random and then hill climbing. In summary, our results demonstrate that prepartition-based solutions consistently outperformed sequence-based solutions due to their inherent flexibility. However, it is important to consider the increased computational costs associated with prepartitioning. Among the heuristics, simulated annealing generally yielded the best results for both solution representations. Starting with the KK solution instead of a random solution could potentially benefit the sequence-based heuristics by providing a lower starting point, although the impact may be less pronounced for prepartition-based solutions. Further investigation and experimentation are necessary to validate and refine these findings.