Analysis of Heuristics for the Number Partition Problem
George Zhang
CS124 Programming Assignment 3 Writeup
https://github.com/gzhang01/cs124prog/tree/master/prog3

1 Abstract

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.

Dynamic Programming Solution
To start, instead of considering the optimization problem, let us consider the yes/no problem as follows: given an input A, is there a way to partition A into A_1 and A_2 such that the sums of A_1 and A_2 are equal? We let b be the sum of the sum of the elements in A, and so our question becomes whether we can produce a set A_1 such that the sum of the elements in A_1 is equal to $\lfloor b/2 \rfloor$. Note that if b is odd, the sum of A_2 will be $\lceil b/2 \rceil$, but we shall for our purposes say that this off by one answer still produces a "yes" result to our question. We shall see that with the filled array from this question, we can produce the answer to the optimization problem.

To obtain the answer to this question, we shall let $D(i, k)$ represent whether there exists a subset of $(a_1, \ldots a_i)$ whose elements sum to k. The answer we want, then, is the value $D(n, \lfloor b/2 \rfloor)$. We notice that all sets have a subset whose elements sum to zero: namely the empty subset. We also notice that in order for a set $(a_1, \ldots, a_i)$ to contain a subset whose elements sum to k, either the set $(a_1, \ldots, a_{i-1})$ contains a subset whose elements sum to k, or that set contains a subset whose elements sum to $k - a_i$. Thus mathematically, we have:

$$D(i, 0) = \text{True}$$

$$D(i, k) = D(i-1, k) \text{ or } D(i-1, k-a_i)$$

Using this recurrence, we fill an array of size $n \times \lfloor b/2 \rfloor$. Our answer is the value $D(n, \lfloor b/2 \rfloor)$. We notice that each square in the array takes constant time to fill, since we are accessing two values in the array, and we iterate over the entire array. Thus the run time of this algorithm will be $O(nb)$, and correctness follows from the logic of the recurrence.

However, this does not yet give us an answer to the optimization problem we want to solve. We notice that if $D(n, \lfloor b/2 \rfloor)$ is True, then we know the optimal residue: 0 if b is even and 1 if b is odd. If $D(n, \lfloor b/2 \rfloor)$ is False, then we can look at $D(n, \lfloor b/2 \rfloor - 1)$. If this is True, then we know that the optimal residue must be 2 if b is even and 3 if b is odd, since if one set sums to $\lfloor b/2 \rfloor - 1$, the other set must sum to $\lceil b/2 \rceil + 1$, and the difference between the sum of sets is either 2 or 3 depending on the parity of b. This pattern continues, and thus we have an algorithm for finding the solution to the optimization problem based on our existing array: find the smallest $p \geq 0$ such that $D(n, \lfloor b/2 \rfloor - p)$ is True. The smallest residue is the $2p$ if b is even and $2p + 1$ if b is odd. The final step at worst travels up one dimension of the array, and so it has runtime $O(b)$, and thus the solution to the optimization problem is still $O(nb)$, or pseudo-polynomial time.

Karmarkar-Karp Algorithm
The Karmarkar-Karp algorithm is a deterministic heuristic for the Number Partition problem. It repeatedly takes the two largest numbers in A and replaces them with their difference, with the intuition being that placing the two numbers in different sets is analogous to placing their difference in some set.

If we assume that arithmetic operations take constant time, we can implement the KK algorithm in $O(n \log n)$ time using a max heap. Initialization of the heap will take about $O(n \log n)$ time, since insertion is $O(\log n)$ and there are n elements to insert (of course, this is a loose bound, since insertion will really only take log of the number of elements already in the heap). Each step in KK will require popping two elements off the heap and inserting back on. Each of these actions require $O(\log n)$ time, so the total for a single step is still $O(\log n)$, since we are doing a constant number of these actions. Finally, we run the algorithm for $n - 1$ steps to remove all but one element, and so the entire algorithm runs in $O(n \log n)$ time with a max heap.

Representations of Solutions and Other Heuristics
There are two representations of solutions we will consider. One is simply a set S of $+1$ and -1 values. We create a random solution by choosing n values from $\{-1, +1\}$, and we define a move from one solution to another as changing a random element's set s_i to $-s_i$, and with probability $1/2$ changing another random element's set from s_j to $-s_j$. We can also represent our solution via prepartitioning, Here a solution is of the form $P = \{p_1, \ldots, p_n\}$ where $p_i \in \{1, \cdots, n\}$ and if $p_i = p_j$, then a_i, a_j are in the same set. A random solution in this representation will assign values from 1 to n for all the p_i and then run KK on the prepartition to produce the two subsets. We define a move on this state space by selecting a p_i and assigning it a different value on $[1, n]$.

There are also three other heuristics to consider. The first is repeated random, where we repeated generate random solutions to the problem and keep the best one (where "best" is defined as the one with the lowest residue). The second is hill climbing, where we start with a random solution and then try to improve it through moves to better neighbors. Finally, we will consider simulated annealing, where we'll generate a random solution and then try to improve it by moving to neighbors which may not always be better – that is we will always move to a neighbor that is better, but we will move to a neighbor that is not better with some probability P that may be a function of how long our algorithm has run.

Our goal will be to implement the KK algorithm, as well as our three other heuristics each using both representations. We will then compare the success of these heuristics as well as their running times.