CS 124 Programming Assignment 3

Qiang Fei, Jordan Turley

April 29, 2020

1 Introduction

In this programming assignment, we analyzed several heuristics for the number partition problem. The problem is defined as follows: given a sequence $A = (a_1, a_2, \ldots, a_n)$ of non-negative integers, we aim to find 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. We will look at the Karmarkar-Karp differencing algorithm, which is deterministic, as well as random algorithms: repeated random, hill climbing, and simulated annealing.

2 Dynamic Programming Solution (Task 1)

The number partition problem can also be viewed in the following way: dividing n numbers to two sets so that the sums of two sets are closest to each other. In other words, this problem is solved if we can find a subset of these n numbers so that the sum of this subset is as close to $\lfloor \frac{b}{2} \rfloor$ as possible, where b is the sum of all n numbers. Thus, if we define $T(m, j)$ as an indicator whether the first j numbers contain some subset of numbers that adds up to m, then if $T(\lfloor \frac{b}{2} \rfloor, n) = 1$, we know that there exists a subset of n numbers which adds up to half of the sum of all numbers, which is the best result we can achieve for partitioning the numbers, yielding a residual of 0 if b is an even number or 1 if b is an odd number.

Generally, we want to find $T(k, n)$ where k is the closest number to $\lfloor \frac{b}{2} \rfloor$. (Without loss of generosity we seek for some k such that $k \leq \lfloor \frac{b}{2} \rfloor$.) In this case, the optimal solution for the number partition problem yields a residual of $(b - k) - k = b - 2k$. From this $T(k, n)$, we can easily trace back how two subsets of numbers can be constructed.

We use dynamic programming to find $T(k, n)$ by constructing a table of size $\lfloor \frac{b}{2} \rfloor * n$ to record values for $T(m, j)$ at each entry. We initialize $T(m, 0) = 0$ for all $m \leq \lfloor \frac{b}{2} \rfloor$, and update the values by columns using the following recursive formula:

$$T(m, j) = \begin{cases} 1, & \text{if } T(m, j-1) = 1 \text{ or } T(m - a_j, j-1) = 1. \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

(a_j is the jth number in the sequence)

Basically, $T(m, j)$ is set to 1 only if the $j - 1$ terms before it yields some subset that sums up to m or m minus the jth number.

In order to be able to construct the subset of numbers for partition, we need to record whether the jth term is added to get m for each $T(m, j - 1) = 1$. Then after we have constructed the whole table and find $\max_{k \leq \lfloor \frac{b}{2} \rfloor} T(k, n) = 1$, we can see whether nth term is included in the subset that sums up to k. We then go

to $T(k-m, n-1)$ if nth term is included and $T(k, n-1)$ otherwise to check if $(n-1)$th term is included. By repeating this step successively we eventually get to $T(\_, 1)$. At that point, we know the indices of all numbers included in the subset that sums to k. The remaining numbers constitute the other subset that sums to $b-k$.

In this way, as the size of the table is $\lfloor \frac{b}{2} \rfloor * n$ and each operation for updating one value in this table takes constant operation, the overall time efficiency to construct the table is $O(\lfloor \frac{b}{2} \rfloor * n) = O(nb)$. The time for constructing the subset of numbers is n times constant operations, which is $O(n)$. The overall time efficiency is then $O(nb)$. The space efficiency is also $O(nb)$ as we need to construct two tables of size $\lfloor \frac{b}{2} \rfloor * n$.

3 Karmarkar-Karp and Random Algorithms

The Karmarkar-Karp algorithms works by repeatedly taking the difference between the two largest elements in the array and replacing then with $|a_i - a_j|$ and zero, until we are left with only one non-zero element which is the residue.

(Task 2) We can implement Karmarkar-Karp in $O(n \log n)$ time by using a max-heap. With a max-heap, we can add numbers to the heap in $O(\log n)$ time and finding the max element in a list of numbers is $O(\log n)$. We must add and remove numbers $O(n)$ times, so the resulting algorithm is $O(n \log n)$. This allows us to quickly perform Karmarkar-Karp, rather than using a naive implementation like scanning over the array every time to find the two largest elements.

The three random algorithms we are looking at are repeated random, hill climbing, and simulated annealing. Repeated random repeatedly finds random solutions and keeps the best. Hill climbing begins with a random solution and moves to a random neighbor if it is better. Finally, simulated annealing starts with a random solution and improves through moves to neighbors, but sometimes moves to the neighbor even if it isn't better with a certain probability to explore the solution space.

For each of the random algorithms, we can run these using the typical solution, i.e. a sequence of -1, and 1, or we can use a method of prepartitioning, where we force certain numbers to be in the same set. When using this representation, we must convert the set back to the typical representation, then use Karmarkar-Karp to find the residue. We will see later that this can result in much more accurate results, but at the cost of processing time. This makes it much more crucial that we have a $O(n \log n)$ algorithm for Karmarkar-Karp instead of a $O(n^2)$ algorithm.

4 Implementation

(Task 3) We implemented our algorithm using C++. Below are the commands to compile and run the program:

```
$ g++ partition.cpp -o partition
```

To run the program with input from a text file, containing exactly 100 integers each on a separate lines, and with a given algorithm using the codes given in the programming assignment, run the program as follows:

```
$ ./partition 0 [algorithm code] [input file]
```

For example:

```
$ ./partition 0 12 input.txt
```

To run the simulation for task 4, run the program as follows:

```
$ ./partition 1 0 0
```

5 Algorithm Simulation Results (Task 4)

To test and simulate our algorithms, we generated 100 random sequences of numbers, each consisting of 100 numbers from $[1, 10^{12}]$. We ran the base Karmarkar-Karp algorithm, as well as the three random algorithms for each of the two representations for a maximum of 25000 iterations, so seven different algorithms in all. Below are the results averaged over the 100 trials.

Algorithm	**Average Residue**	**Average Runtime (seconds)**
Karmarkar-Karp	236420	0.00008
Repeated Random	294848000	0.02
Hill Climbing	330351000	0.0203
Simulated Annealing	317500000	0.027
Prepartitioned Repeated Random	175.98	3.417
Prepartitioned Hill Climbing	876.58	3.221
Prepartitioned Simulated Annealing	342.32	4.872

Somewhat surprisingly, the deterministic Karmarkar-Karp results in a residue that is far better than any of the base random algorithms, and runs in a fraction of the time. Karmarkar-Karp runs almost instantly on an array of only 100 numbers. The random algorithms did surprisingly bad, considering the other residues we get. However, they still run very quickly, taking only a fraction of a second to run for 25000 iterations. It is possible that with many more iterations, we could achieve a much better solution.

We expected the prepartitioned results to be better than the regular results, since this is a more sophisticated method of representing the solution. It requires more time and effort to code and to run, so we would hope it is better than the base algorithms; otherwise there would be no point. However, we were surprised that using this representation results in a residue that is orders of magnitude better than either the Karmarkar-Karp algorithm or the base random algorithms. We were surprised to see that prepartitioned repeated random was significantly better than either of the 'more-sophisticated' versions of the random algorithm. However, the downside of these algorithms is the runtime. Calculating the residue of the base random algorithms is simple and is a quick $O(n)$ operation, but in this case, we have to convert back to the standard representation and then run Karmarkar-Karp on this representation to find the residue.

A quick back-of-the-envelope calculation tells us that allowing the base random algorithms to run for the same amount of time as the prepartitioned algorithms would allow us to run them with a maximum number of iterations of 1.6 million, which would almost definitely result in a better residue. We did not analyze this, but it would be interesting to look at. We assume that the prepartitioned would still perform better, but this would give the randomized algorithms a lot more time to find better solutions.

(Task 5) Karmarkar-Karp gives us a solution itself, so this could be used as a starting point for the other algorithms. To do this, we would run Karmarkar-Karp to find the solution S, a sequence of -1 and 1. We can plug this directly into the first three random algorithms, or if we want to convert this to a prepartition, we can simply assign -1 to 1 and 1 to 2 in the prepartition sequence. This could result in even better results, as the result from Karmarkar-Karp can be computed almost instantly and will likely be better than a random solution that we would regularly start with.

6 Conclusion

This programming assignment embodies a lot of really useful techniques and skills that can be used to break down a problem that we may initially think is intractable. If we used the most naive algorithm for this problem, then we would try all 2^n possible solutions to find the best. For n up to about 32 isn't too much of a problem, but for $n = 100$ this is impossible. Even after reducing it to dynamic programming, it will still take significantly more time to compute this than to use one of our methods here.

Here, instead of finding the optimal solution, we find a solution that is near-optimal. We see that with prepartitioning we are able to do very well and get within about 200 of the optimal solution, which is impressive when we have 100 numbers drawn from $[1, 10^{12}]$. We also only ran the algorithm for 25000

iterations, or a few seconds. If we let this run for a few minutes or even a few hours, I am sure that we could do even better.

There should also be emphasis on how we represent the problem. The most natural way is as a sequence of -1 and 1, but we see that this representation does not give us very good results. Even the deterministic Karmarkar-Karp algorithm was significantly better than what we found using this representation. However, by changing to the prepartitioning representation, we get results that are orders of magnitude better than before. We do pay a cost in processing time, however. The earlier representation takes a fraction of a second to compute, but preprocessing took up to an average of five seconds to compute. Five seconds is nothing, but if we wanted to look at sets of size $n = 1000$ or $n = 10000$ or even greater, than this could become a problem. Also, if we account for this added time, we could run the original algorithms for many more iterations which would almost definitely result in a better solution. It would be interesting to compare results when we allow both to run for a certain amount of time rather than a certain number of iterations.

Finally, we see the power of a fast approximation compared to finding the optimal solution. In many scenarios, finding a good approximation of a solution is almost as valuable as finding the correct solution. Take for example the traveling salesperson problem and the US Postal Service. The US Postal Service would love to be able to find the optimal route to take to deliver packages so that they can maximize the number of packages delivered in a certain amount of time and minimize costs like fuel. However, a really good approximation of the solution is very useful too, and if it can be computed quickly, then this is nearly as valuable as the optimal solution itself.