Coding Assignment 3

CS 124, Spring 2021

Due Wednesday, April 28, 2021

Thomas Jaklitsch, no collaborators
Number of late days on previous problem sets: 12
Number of late days including this problem set: 12

1 DP solution to Number Partition

Let b be the sum of all the entries in the array we are given. Let us define $D(i, j) = 1$ if there exists a subset of the first j elements that sum to i. Otherwise $D(i, j) = 0$. We will use these to determine the correct partition of the set. For the base cases let us define $D(0, j) = 1$, since the empty set has a sum of 0. Next let us define $D(i, 0) = 0$ unless $i = 0$.
Now for the recursion let us define $D(i, j) = 1$ if $D(i, j-1) = 1$ or if $D(i - a_j, j - 1) = 1$ where a_j is the jth entry of the array. In the first case, if the first $j - 1$ elements had a subset that sums to i then so do the first j. In the second case if the first $j - 1$ elements have a subset that sums to $i - a_j$, then adding the next element will have a subset that sums to i. Now we can build up from $D(0, j)$ and $D(i, 0)$ to populate the matrix to $D(\lceil b/2 \rceil, n)$. Also, each time we calculate a 1, we will record where each of these j values come from, so that we will be able to return a partition that will minimize the difference between the sums of the two sets.
For the run time, each stage of the recursion takes a constant amount of time and there is $O(bn)$ many entries of the matrix to populate. Therefore, the running time is $O(bn)$.

2 Karmarker-Karp in $O(n \log(n))$

We can show that Karmarker-Karp can run in $O(n \log n)$ as follows. If we are given an unsorted array we can build a binary heap in $O(n)$ time. Once we have built the binary heap, at each step, we want to find and remove the maximum element twice. Each of these steps take $O(n \log n)$. We are assuming the taking the difference of the numbers is $O(1)$, so we take the difference and then insert the element which is $O(\log n)$. This gives us $O(3 \log n) = O(\log n)$ for each step. We must continue this algorithm until we have a single element remaining, so this will be $O(n)$ steps. Therefore, we have a total of $O(n \log n)$ for this stage of the algorithm. Finally, we must two-color the graph as described in the assignment, which is $O(n)$. Therefore, for the entire algorithm, we have $2O(n) + O(n \log n) = O(n \log n)$.

3 100 Random Instances of the Problem

I wrote my code in C/C++ for this part of the assignment. I generated 100 random arrays and then ran each of the algorithms on these 100 random arrays. I ran the random algorithms 100,000

times each for each given array. I then calculated the maximum residue, minimum residue and average for each of the algorithms as well as tracking the time it took for each algorithm to run all 100,000 iterations for each of the 100 random problems. The Karmarker-Karp algorithm and the algorithms using the standard representation are given in the first table. The algorithms that use prepartitioning are given in the second table.

algorithm	Max	Min	Avg	Time
KK	5722977	5119	282154.00	0.00425
RR	208032313	706407	62184854.66	22.35604
HC	344388650	1297328	81850281.16	2.482304
SA	374682248	19092	74010258.50	6.726550

algorithm	Max	Min	Avg	Time
RR	465	0	229.74	375.917711
HC	462	0	229.50	337.217849
SA	462	0	229.46	342.423840

The first interesting thing about these different algorithms is the large gap between the standard representation algorithms, the Karmarker-Karp algorithm, and the prepartitioning representation algorithms in terms of both performance and time. The standard representation algorithms seem to be roughly two orders of magnitude behind Karmarker-Karp in terms of both their maxes and average value. In terms of minimums, however, The simulated annealing algorithm had a much better minimum than the other two random algorithms. I suspect that this is because it can move to solutions that might be worse, so it does not always get caught in locally optimal regions. Therefore, in the best case it can find a good solution.
The prepartitioning algorithms on the other hand performed extremely well even compared to the Karmarker-Karp algorithm, and they were also very close to one another in terms of performance. I was suprised to see that in the prepartitioning case, hill climbing was not again the worst performer as it was in the standard representation case.
I used 100,000 iterations to improve performance on the different algorithms and get a better idea of how they operate given many trials. This came at the cost of time, however. The prepartitioning algorithms are especially expensive on time with all of them taking over five minutes for all trials. The randomized algorithms were much faster than them. This seems to be one of the great advantages of the Karmarker-Karp algorithm, it can produce good results compared to standard representation random algorithms and it does not have to do many trials so its total run time will be orders of magnitude faster.

4 Karmarker-Karp as Starting Point

We could use the Karmarker-Karp algorithm as a starting point for the randomized algorithm by using the sequence it returns as the starting point instead of initially generating a random sequence. For the repeated random algorithm, this has a very high chance of simply causing the algorithm to just return the Karmarker-Karp solution. This is because new random solutions are not dependent upon previous ones and based off the experimental results, a random solution is very unlikely to beat Karmarker-Karp. The hill climbing algorithm and simulated annealing will only be able to improve from the Karmarker-Karp solution. This would seem to only improve the algorithms based off of the large gap between Karmarker-Karp and the randomized algorithms above. There is a

chance, however, that they could get stuck in a local optimal situation. The hill climbing algorithm may benefit the most from this since the Karmarker-Karp solution is already good and the hill climbing does not move to worse positions unlike simulated annealing.