YiHua Leng, Daniel Schwartz
CS 124
04-26-21

Programming Assignment 3: Number Partition Problem

Introduction

Our goal was to implement the Karmarkar-Karp algorithm and multiple heuristics on random inputs to solve the Number Partition Problem. These heuristics include the repeated random heuristic, hill climbing, and simulated annealing, all of which we use two representations: by a sequence and pre-partitioning. We used C++ to implement the code and Python for the analysis. We found that the pre-partirion representation out-performs the sequence representation, but at the expense of time. We also found that simulated annealing is generally the best heuristic to work with, with some exceptions.

Proving Number Partition Solvable in Pseudo-Polynomial Time

A pseudo-polynomial time algorithm for number partitioning is possible using dynamic programming. The idea behind this algorithm is that if we know the number partition of the set that contains $|A| - 1$elements, then we can use that solution to figure out the set of all $|A|$ elements. We start with a set of numbers *A* that sum up to some number *b*, and we make our dynamic programming table such that there are $|\lfloor b/2 \rfloor|$rows and $|A|$columns. Let our dynamic programming table be defined by $D(n, s)$where *n* is the set and *s* is as sum. We can populate our table with 1s and 0s, where a field has a 1 if the set of numbers labeling that column don't have a subset sum of that row's value, and there is a 0 if that set of numbers can sum to the row's value. This may seem counterintuitive since our 1 is "false" and our 0 is "true", but it provides a more intuitive process (for me) down the line. An example is below for the set $A = \{1, 2, 3, 4\}$:

	{}	{1}	{1,2}	{1,2,3}	{1,2,3,4}
0	0	0	0	0	0
1	1	0	0	0	0
2	1	1	0	0	0
3	1	1	0	0	0
4	1	1	1	0	0

5	1	1	1	0	0

Now that we have this populated table, we can calculate the residue. We do this by checking if $D(n, s) = 1$ or 0. If it is a 0, then we break and add a residue of 1 to the running residue if *b* is odd and 0 if *b* is even. But if $D(n, s) = 1$then we check if $D(n - 1, s) = 1$or 0 and we check if $D(n - 1, s - a_n) = 1$or 0. We continue this until $D(n, s - 1) = 0$ or $D(n - 1, s - a_n) = 0$. For our recurrence, we know that the 0th row will always be populated by a 0 since all sets, including the empty set, can be partitioned such that one set sums to 0. So our recurrence relation is:

$$D(n, 0) = 0$$

$$D(n, s) = D(n - 1, s) \,||\, D(n - 1, s - a_n)$$

In the worst case, we will go through every square in the matrix, giving us O(n*b*) time which is pseudo-polynomial.

Implementation

Karmarkar-Karp: The first thing we did was write a program to solve the Karmarkar-Karp (KK) algorithm because we would need it for other functions like calculating the pre-partition residue function. We decided to use a max heap for our implementation because we knew that we would have to always have the two max's of the set of numbers available, and a heap is ideal for this because you can pop off the max in O(logn) time rather than O(n) if we just kept the numbers in a standard array. We could use C++'s standard priority queue class for this, but we decided to create our own data structure for the sake of learning. We included the functions pop, push, get_max, heapify, and print_heap. Once we had our heap data structure, implementing KK was fairly straightforward. Given an array of numbers, we would put them into heap. Then we would save the max as *max* and pop it off, save the new max as *second_max* and pop it off, then we would push the difference value back into the heap. There was no need to save the zeros—even though they replaced the minimum of the two numbers with a 0 in the example—because we can just break the program when $size(heap) = 1$rather than stopping the program when the second number is a 0. We can recur on this new heap and continue while $size(heap) > 1$, and then return the number that's left.

For the other algorithms, we first need to define a few functions. Of course two of these functions are calculating the residual from the standard sequence representation as well as the pre-partition representation. These functions are necessary because we need to use both of

these representations for each algorithm. For the standard sequence representation, we take an array *nums* of numbers and an array *s* of a random sequence of -1s and 1s.

> Side note: To create this *n* length array of -1s and 1s, we have a function called *random_sequence* that uses rand() in C++ to generate a 1 or -1 with a probability of 0.5.

And with this *s* and *nums*, we can calculate the sequence residue by looping through *nums* and *s*, both of which are the same length, and multiplying their corresponding index values, adding them to a cumulative variable *res*. We take the absolute value of *res* and that is our residue for a random sequence.

Now for the pre-partition residue function, we take *nums* and *s* again, except *s* is now calculated as a random partition rather than a random list of 1s and -1s.

> Side note: To create this *n* length random partition, we have a function called *random_partitions* that uses rand() to generate n numbers between 1 and *n.*

We can then make our *parti_residue* function that builds an array *p* where the indexes of *p* that have the same value are added together in *A* to make A'. We then run KK on these numbers *A'* and return the result.

With these functions *seq_residue* and *parti_residue*, we can now build our algorithms. For each of these algorithms, we take an array *nums,* an array *start,* a value *n,* and a boolean value *is_seq*. Start is the initial random array (sequence or pre-partition).

Repeated Random: For both the random sequence and partition, differing by changing the *is_seq* value, we start with a random solution *S.* Then we iterate from 1 to 25,000—as described in the problem—and generate a new solution *S'.* If the residue(S') < residue(S') then S = S'.

Hill Climbing: The only difference here is that instead of generating a new random solution each iteration, we now set S' to a random neighbor of S. We need a function to generate that random neighbor, and this neighbor will be different depending if we are using random sequence or pre-partitioning.

> For the random sequence neighbor, we pick two *different* values in S. We randomly flip the first element and flip the second with probability ½. For the pre-partitioned neighbor, we randomly switch one element in S.

Now we do the same thing that we did in Repeated Random, except we aren't comparing S to a randomly generated S', we are comparing S to a random neighbor. The concept is similar to gradient descent in Deep Learning, in that we continuously optimize in an attempt to find the minimum loss and the closest thing to the true residue.

Simulated Annealing: Simulated Annealing is similar to Hill Climbing, but we add one more possible move as we navigate our state space. In the case that the residue of the random neighbor of S is not less than the residue of S, we don't immediately go to the next iteration. We go to a move that is *not* better with some probability of

$currentResidue - neighborResidue\ /coolingSchedule$, where our cooling schedule is $T(iter) = 10^{10}(0.8)^{iter/300}$.

KK Runtime

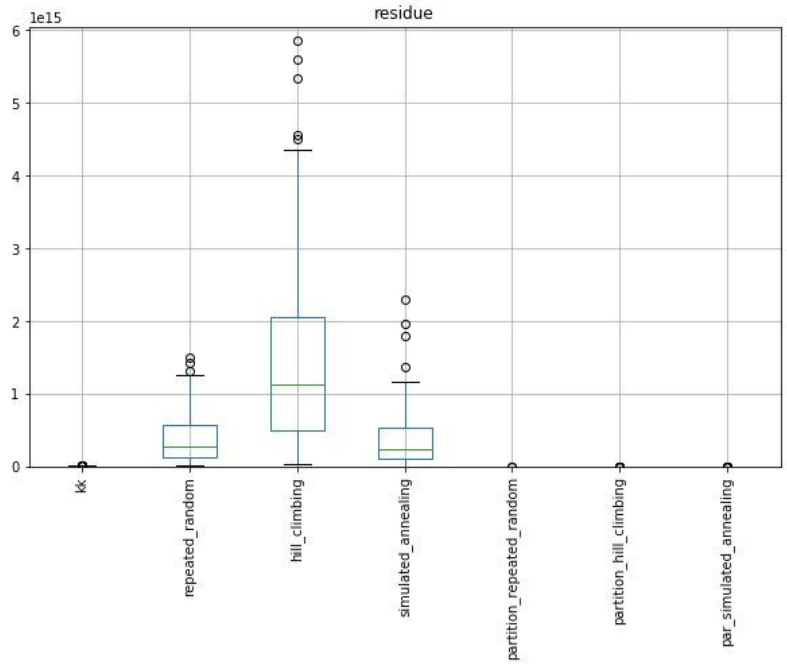
The KK algorithm can be implemented in $O(nlogn)$ steps using a max heap, as we used. Building the heap from an array of numbers is $O(nlogn)$, and popping off each max is $O(logn)$, which we do $n - 1$times. So we get $O(nlogn + (n-1)logn) = O(nlogn)$.

Outputs/Analysis

kk ALG:	
count	1.00E+02
mean	2.10E+12
std	2.93E+12
min	1.29E+10
25%	4.68E+11
50%	1.14E+12
75%	2.46E+12
max	1.80E+13
Avg Time:	0.02591

Random	Seq		Hill Climbing	Seq		Sim Anneal	Seq
count	1.00E+02		count	1.00E+02		count	1.00E+02
mean	3.84E+14		mean	1.68E+15		mean	3.67E+14
std	3.59E+14		std	1.69E+15		std	4.04E+14
min	3.78E+12		min	3.37E+13		min	2.23E+12
25%	1.16E+14		25%	4.95E+14		25%	9.80E+13

50%	2.72E+14		50%	1.12E+15		50%	2.36E+14
75%	5.74E+14		75%	2.05E+15		75%	5.29E+14
max	1.49E+15		max	8.57E+15		max	2.29E+15
Avg Time:	48.0782		Avg Time:	18.4967		Avg Time:	21.6229
Random	Partition		Hill Climbing	Partition		Sim Anneal	Partition
count	1.00E+02		count	1.00E+02		count	1.00E+02
mean	1.72E+09		mean	6.29E+09		mean	2.00E+09
std	1.59E+09		std	7.24E+09		std	2.05E+09
min	0.00E+00		min	0.00E+00		min	0.00E+00
25%	1.68E+04		25%	2.15E+09		25%	3.36E+04
50%	2.15E+09		50%	4.29E+09		50%	2.15E+09
75%	2.15E+09		75%	8.59E+09		75%	2.15E+09
max	8.59E+09		max	5.15E+10		max	8.59E+09
Avg Time:	1005.41		Avg Time:	915.115		Avg Time:	535.038

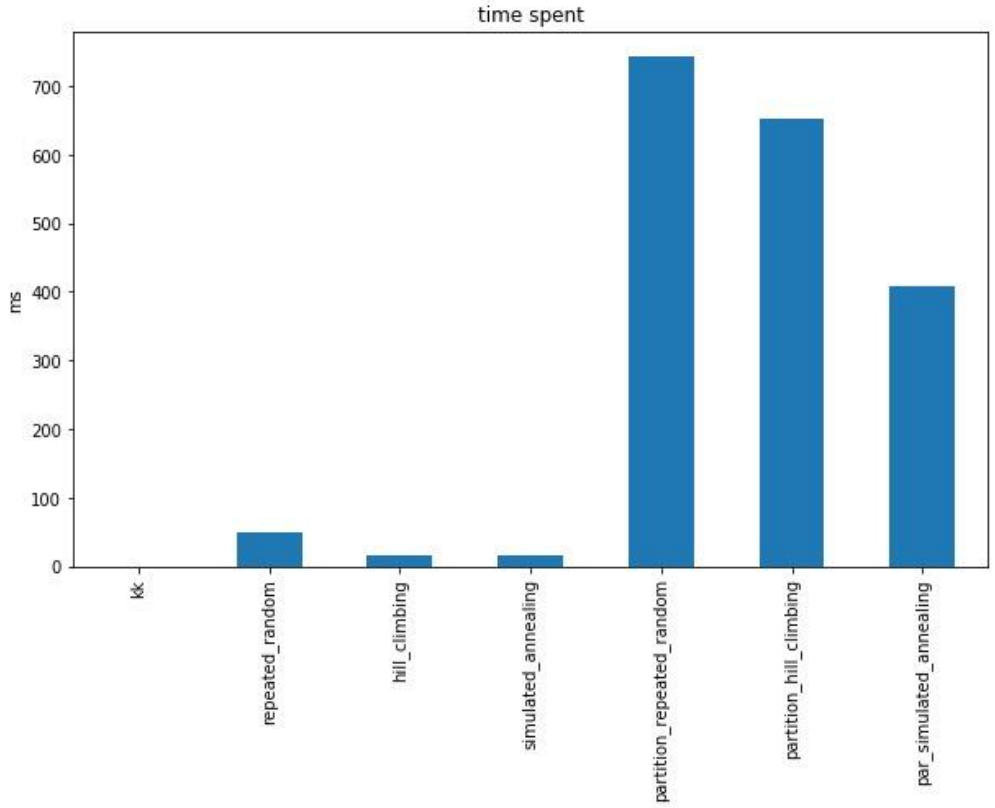


The pre-partitioned representation is better than the random sequence representation across the board. Within the sequence representation, we see that the mean of the random annealing is the lowest, then the hill climbing, then the random. But for pre-partition, random is the lowest, second is annealing, then hill climbing. First we can tackle the reason why pre-partition is so much better than random sequence, and then we can go into the performance of algorithms within these representations.

The pre-partition representation having better results makes sense because we are not only grouping values together to provide some initial separation, but we are also running the KK algorithm to further optimize the residue. For the sequence representation, we are simply assigning random groups without running KK. The discrepancy is clear in the box plot above. The sequence representation medians are clearly above the pre-partition representations, and there are even a number of outliers in the sequence representation that are *significantly* higher than pre-partition.

Our results were much more surprising within each representation. We expected the simulated annealing algorithms to produce the best result because it took into account the fact that we may just be stuck in a local minimum. This was true for the sequence representation. In the box plot, we can see the sequence annealing and sequence repeated random are very similar. Although annealing has more high outliers, which is likely a result of unlucky iterations where we ventured down the non-optimal path and weren't actually escaping a local minimum. Surprisingly, it turned out that simulated annealing was worse than the random sequence for pre-partitioning. It's possible that we would have gotten a different result if we used a different probability for going in a worse-off direction. Maybe if we looked more closely at our state space and noticed it to have many somewhat deep local minima, we could raise the probability of trying a worse off direction.

We also measure the running time for each algorithm.



As expected, sequence ran faster than pre-partition. This makes sense because in the pre-partition residue function, we use the KK algorithm on top of generating *S*, whereas in the sequence residue function, we just have to generate *S* and then multiply it by *A* and add up the numbers. And since we are calling part_residue 25,000 times, that is 25,000 multiplied by our average KK runtime of 0.02591 which is ≈ 650, close to our average runtimes for our pre-partition algorithms.

The difference in runtimes within representations makes sense as well. The random algorithm is always the slowest because each iteration it must generate an entirely new random sequence. Whereas hill climbing and annealing only need to find a neighbor of the current sequence, which consists of only changing one or two elements. What's less trivial is the runtimes between simulated annealing and hill climbing. It is expected that annealing would always be slower than hill climbing because we are doing less operations in hill climbing. This is true for sequence but not random partition. The reason it is not always true—as we see in random partition—is because of our implementation of the algorithm. We calculate the residue of *S* and *S'* and save them for the next iteration. So if $residue(S') < residue(S)$, then we don't have to re-calculate the residue of *S* in the next iteration. Therefore if we reach a good *S* residue quickly, then it will save us calls to the residue function in future iterations. And it appears that we do reach this optimal residue quicker in the annealing algorithm than the hill climbing algorithm because we are attempting to free ourselves from the dreadful local optima. It

is inconclusive why this same effect didn't happen with the sequence representation. It is possible that it did happen to some degree but the effect was less drastic because calling the residue function for the sequence representation matters less in terms of time.

Discussion

We will discuss the effect of using the KK algorithm as a starting point for randomized algorithms. From the data above, we can see that the pure KK algorithm results are better than all sequence representation results. So there isn't really any reason to run the sequence representation if the solutions produced by each iteration are worse than those produced by the KK algorithm. We might as well run the KK algorithm first, and use that value as a threshold that the sequence representation algorithms must cross before we use their result. And it isn't likely that they will cross this threshold often. In the residue box plot, the first quartile of all sequence algorithms doesn't even reach the KK value. We can use this technique for the pre-partition representation as well, and it would hopefully speed up many of the initial iterations since we start off with a fairly optimal KK value. It would then get further optimized as the pre-partiion representation helps our results.

The programming assignment was much needed after learning about so many heuristics that make sense but are hard to wrap your mind around. After implementing the algorithms, I can clearly see this whole other arena of algorithms that rely on stricter parameters but produce better results. The implementation was especially fascinating because I can see practical uses of heuristics like these. For example, if we need an algorithm for number partitioning—say, we want to evenly separate two groups of people so that their ages are similar—and we don't care much about time, then we may start with a pre-partitioned representation. But if we *do* care about time and just need a somewhat optimal solution, then a sequence representation may be the better way to go.