Number Partition Heuristics

Rodney Lafuente Mercado

April 2020

1 Number Partition

The number partition problem can be described as finding the subset of a list of numbers whose sum is closest to half the sum of all numbers. If $A = [a_1, a_2, \dots, a_n]$ is the list of numbers then the optimal sum is $\lfloor \frac{sum(A)}{2} \rfloor$.

This problem can be solved dynamically in the following way.

- Let M be a table such that $M(i, s)$ is a subset S of $\{a_1, a_2, \dots .a_i\}$ and $sum(S) = s$. If no such subset exists, $M(i, s) = \emptyset$.
- Recursively define $M(i, s)$ as equal to $M(i-1, s)$ if $M(i-1, s) \neq \emptyset$ or as $M(i-1, s-a_i) \cup a_i$ if $M(i-1, s-a_i) \neq \emptyset$. If $M(i-1, s) = M(i-1, s-a_i) = \emptyset$ then $M(i, s) = \emptyset$.
- Build M starting at $(i = 0, s = 0)$ such that in the end M has dimension $n \times \lfloor \frac{sum(A)}{2} \rfloor$.
- Return the non-empty set with the largest sum in the last row of M.

The explicit solution can be found by constructing a set sol of length n such that $sol[i] = -1$ if $A[i]$ is in the resulting set and $sol[1]$ otherwise.

For every possible i and $s < \lfloor \frac{sum(A)}{2} \rfloor$, the algorithm checks if there is a subset of the first i numbers in A whose elements sum to s. In the end the subset of all numbers with the largest sum (i.e. the sum closest to $\lfloor \frac{sum(A)}{2} \rfloor$) is returned. This partition is the optimal one as it represents the subset of A with sum closest to half the entire sum, and thus the optimal residue.

Each table update runs in constant time so the total runtime, including the final search for the set with the largest sum, is $O(n \cdot \lfloor \frac{b}{2} \rfloor + n) = O(n \cdot b)$ where b is the sum of all elements in A.

2 Karmarkar-Karp

The Karmarkar-Karp algorithm can be implemented in $O(n \log n)$ runtime if one uses a binary heap. Instead of repeatedly choosing the two largest numbers the numbers can be put in a heap that prioritizes larger numbers. The two largest numbers can then be repeatedly extracted and their difference inserted until only one element is left in the heap. This last number is the residue.

Building a binary heap from the list takes $O(n)$ time. The operation for extracting the largest number from the heap runs in constant time and the operation for inserting a number into the heap runs in $O(\log n)$ time. The total runtime is $O(n \log n)$ since we are inserting numbers into a heap roughly $n/2$ times.

3 Data

The following are the residues from running 100 trials with 25,000 repetitions for each of the randomized algorithms.

ALG	Mean	Median	Min	Max	Std. Dev
KK	325900.7	104636.0	1304	2913758	561731.3
Standard Representation					
RAND	262800670.6	198162967.0	1291156	1155720559	243199501.4
HILL	366760740.9	220357590.5	1147667	2788988992	455859965.5
SIMUL	383969460.6	209085607.0	3776016	2349206310	431895491.9
Prepartitioned Representation					
RAND	151.1	113.5	5	618	137.6
HILL	606.5	318.5	4	4230	796.0
SIMUL	1478.0	919.5	8	8746	1580.1

The following are the times taken, in seconds, by each algorithm.

ALG	Avg. Per Trial
KK	0.00079
Standard Representation	
RAND	1.65
HILL	0.40
SIMUL	0.42
Prepartitioned Representation	
RAND	34.87
HILL	32.75
SIMUL	33.10

It is clear that the prepartitioned algorithms performed drastically better than their standard represented counterparts (by a factor of 10^6 in most cases). This performance is also reflected in their significantly longer running times.

On average, the Repeated Random algorithm performed better than the other two and the Hill Climbing algorithm was the worst. Perhaps with more than 25,000 iterations the

Simulated Annealing algorithm would have performed better then the other two. These comparisons are the same among the medians for each algorithm, however, the medians are much lower than the average for each algorithm, meaning they were significantly not affected by outliars.

4 Using Karmarkar-Karp

Using a solution given by the Karmarkar-Karp algorithm would give a lower bound for all of our approximation algorithms. Specifically, for Repeated Random, it would give a starting point that would be much less likely to be improved by random solutions. For Hill Climbing it would give it a starting point more likely to be close to a more optimal local (or global) minimum in the solution space. These two effects would simultaneously take place in the Simulated Annealing algorithm, as it both hill climbing and, in a way, repeated randomness (moving to worse neighbors) take place.