Number Partition Heuristics

Katrina Brown[1] **and Swati Goel**[2]

katrinabrown@college.harvard.edu[1] sgoel@college.harvard.edu[2],

1 Introduction

Number partition is an NP-Complete problem. However, it has a pseudo-polynomial time solution using dynamic programming as well as several heuristic solutions, which we will explore in this paper. We detail the dynamic programming approach in the following section.

1.1 Number Partition DP

We define a dynamic programming solution via the following recurrence:

dp$[i][b_j]$ = min(dp$[i+1][b_j]$, dp$[i+1][b_j$+arr$[i]]$)

arr represents the given array of values to be partitioned. i represents the index of **arr** being considered. b_j represents the current sum of values in partition 1. We iterate through all i, b_j pairs to calculate partitions 1 and 2. We build **dp** in descending order of i. For a given i, we fill in ascending order of b_j. (b_j order is arbitrary. But it is key that all $i+1$ through $n-1$ have been calculated before examining i). **dp**$[i][b_j]$ represents the minimum residue possible from state (i, b_j). We initialize **dp**$[n][b_j]$ = $|n - 2b_j|$ for every value of b_j from 0 through b. We return **dp**[0][0].
To clarify the definition, before jumping into a proof of correctness. We are defining partitions 1 and 2 via inclusion or exclusion from partition 1. We define a state of the problem as (the next element to be assigned, the sum of elements in partition 1). The value of a state represents the minimum residue possible from that state.

A proof of correctness is as follows:
Any value referenced will already have been computed. **dp**$[i]$ depends solely on **dp**$[i+1]$. We compute **dp** such that every $i+1, b_j$ pair has been computed prior to examining **dp**$[i]$.
Any solution returned can be constructed with the given set of numbers. We may compute solutions which are not possible to construct (an inefficiency future algorithms may eliminate). This could occur, for example, when we compute **dp**$[0][b]$. However, we always return a solution which is possible to construct. Recall that we return minimum residue from starting state **dp**[0][0], which is definitely valid. If we start with a solution state that is valid, we can only move to other valid solution states. This is definitionally true from our recurrence relation, which moves to a next state by including or excluding the current element. Therefore we know the solution we return is possible to construct, even though some elements of the **dp** do not represent valid states.
At every decision point we examine both possible next states and take the best decision (ie minimize residue). Given that our returned solution is valid (which we've already shown), it must also be optimal. We show this formally via induction.
BASE CASE: $i = n, b_j$. The residue is simply $|n - 2b_j|$ (there are no more decisions to make).
IHOP: Assume the minimum possible residue has been computed correctly for all states $i+1$ through n.
ISTEP: We show that under the IHOP, the recurrence function to compute state i, b_j is correct. We can either include or exclude element i from partition 1. If we include element i, the problem becomes $i+1$, b_j+arr$[i]$. If we exclude i, the problem becomes $i+1, b_j$. The minimum of these two outcomes must therefore be the best outcome for i, b_j.
IPROVE:)
Runtime: There are nb states and each state is computed exactly once. There are a constant number of steps performed to compute a state. Thus, the asymptotic runtime is $O(nb)$.

2 Heuristic Approaches

2.1 Heuristic Algorithms

2.1.1 Karmarkar Karp

Implement Karmarkar-Karp in O(n log n) steps
We implement Karmarkar-Karp in O(n log n) steps using a max-heap.
Given an input list of integers arr, run build-max-heap on arr. Extract the two largest values from the array, then insert the difference of the two numbers, and the integer 0, back into the array. Repeat the previous step until the second largest value extracted is 0, at which point there is only one value–the residue–remaining in the array. Return that value.

Runtime: As proven in lecture, each insert() and extract-max() operation takes O(log n) time. Each time that we extract two values from the array, we insert two values into the array, at least one of which is zero. Thus the number of elements in the heap equal to zero increases by at least one for each iteration of the for-loop. It follows that the algorithm will terminate after $< n$ iterations of the loop, because there are always $\leq n$ values in the heap, and once $\geq n-1$ of those values are 0, the second extracted value will be zero, and the while loop/algorithm terminates. Each iteration of the while-loop contains 2 extract-max and 2 insert() operations that collectively take $O(logn)$ steps, and we have $O(n)$ such iterations, so our number of steps is $O(nlogn)$ (assuming that arithmetic operations only take 1 step each).

2.1.2 Repeated Random

As name suggests, repeatedly generate a random solution and compare to current best solution. Pseudocode as follows:

```
  Start with a random solution $S$
**for** iter = 1 to max_iter
  $S'$ = a random solution
  **if** residue($S'$) < residue($S$) **then** $S = S'$
return $S$
```

2.1.3 Hill Climbing

As name suggests, make a random move to an adjacent solution (but only if that solution lowers residue). Hill descending, as it were. Pseudocode as follows:

```
  Start with a random solution $S$
**for** iter = 1 to max_iter
  $S'$ = a random neighbor of $S$
  **if** residue($S'$) < residue($S$) **then** $S = S'$
return $S$
```

2.1.4 Simulated Annealing

Pseudocode as follows:

```
  Start with a random solution $S$
$S'' = S$
**for** iter = 1 to max_iter
  $S'$ = a random neighbor of $S$
  **if** residue($S'$) < residue($S$) **then** $S = S'$
  **else** $S = S'$ with P exp($-$(res($S'$)-res($S$))/T(iter))
  **if** residue($S$) < residue($S''$) **then** $S'' = S$
return $S''$
```

2.1.5 T(iter) cooling schedule choice

We used the suggested $\text{T(iter)} = 10^{10}(0.8)^{\lfloor \text{iter}/300 \rfloor}$ function.

For the simulated annealing algorithm, we wanted a T(iter) temperature function that would decrease over time. At the start of the simulated annealing algorithm, we are willing to accept a greater number of transitions that lead to higher-residue solution, for the sake of exploring more of the state space. During later iterations, we want to accept fewer transitions that lead to a higher-residue solution, but we still want to accept a few transitions so that we have a chance to escape local minima.
We define T to decrease exponentially, with the greatest decrease occurring from iteration 0 (T(0) approaches ∞) to iteration 6000 (T(6000) = 1.304). This decrease corresponds to a decrease over time of the chance exp($-$(res(S')-res(S))/T(iter) of choosing a state-space with higher residue.
Alternatives: We spent some time modifying and testing the constants in T(iter), but did not find constants yielding significantly lower residue on average. Experimenting with a polynomial cooling schedule did not yield improved results.

2.1.6 Further Improvements

The Karmarkar-Karp algorithm returns a good approximation for the minimum residue, although its output is not necessarily optimal. We can use the output of Karmarkar-Karp to improve the repeated random, hill climbing, and simulated annealing algorithms.
For each algorithm, instead of starting with a random solution S–a list of 1's and -1's assigning each element to one of the two sets–we can start with the solution given by the output of the Karmarkar-Karp algorithm. For the repeated random algorithm, this upper-bounds the residue of the returned solution, by ensuring that S will have residue less than or equal to the output of Karmarkar-Karp. For hill-climbing and simulated-annealing, the returned residue is similarly upper-bounded.
Additionally, since hill-climbing only moves toward improved solutions (with lower residue) starting from the initial solution, this would result in more searches of closer to optimal states. Simulated annealing can move to worse states, so it is not guaranteed to only explore states at least as good as the starting state; however, it would also be more likely to explore more states closer to optimal. Both algorithms search for local minima of the residue function. Thus, the starting point of the algorithm is important (though, as noted, not determinative for simulated annealing)
Using Karmarkar-Karp as a starting point allows more searches of the state-space closer to the optimal solution.
We can upper-bound the return value of the pre-part representation heuristics. However, as mentioned by a TF in OH, it seems less natural/useful to use K-K to improve the subset of the state-space explored by the pre-part repeated random, hill-climbing, or simulated annealing algorithms.

3 Results

Give tables and/or graphs clearly demonstrating the results. Compare the results and discuss.

We test the karmarker-karp, repeated random, hill-climbing, and simulated-annealing algorithms, where the last 3 are implemented using both a standard and a pre-part representation. We show the mean and standard deviation of residues returned across 50 trials where each trial has a max iteration value of 25000.

Alg	Mean	Std Dev
K-Karp	247,583	324,950
Rep Rand	297,828,080	224,211,173
Hill Climb	252,690,514	328,898,210
Sim Anneal	193,935,938	182,096,779
PreP Rep Rand	197	175
PreP Hill Climb	711	582
PreP Sim Anneal	315	324

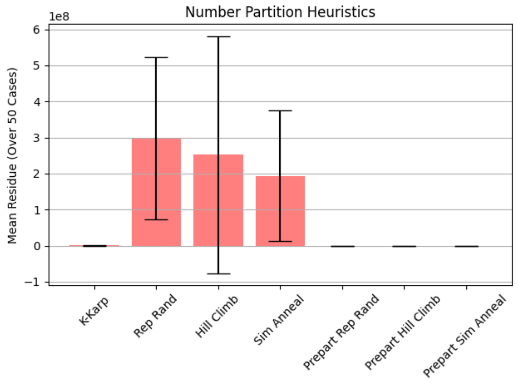
Table 1. Algorithm Residue Results (Over 50 Iterations)

The heuristics implementing a pre-part representation produced results significantly closer to optimal than the standard representation heuristics or Karmarkar-Karp. Karmarkar-karp also significantly outperformed the standard representation heuristics. There is, however, a time-complexity trade-off.
Karmarkar-Karp is much faster than the heuristics, since each heuristic runs Karmarkar-Karp at least 25,000 times. We conclude that Karmarkar-Karp outperforms the standard representation heuristics in terms of both time complexity and expected quality of approximation.
It is worth noting that across our trials for the prepart representation heuristics, the repeated random heuristic obtains the closest to optimal results. The hill-climb and simulated annealing algorithms perform relatively worse, despite exploring more of the state space surrounding closer to optimal solutions. It is possible that both of these heuristics get stuck in local minima. Simulated annealing performs better on average than hill-climbing, perhaps because it is better able to escape local minima by accepting some solutions that have relatively higher residue.
For the standard representation, simulated annealing outperforms hill climbing, which outperforms repeated random. This is likely because simulated annealing explores more of the lower-residue states while having a chance to escape local minima, while repeated random performs relatively worse because it cannot build off of solutions with lower residue.

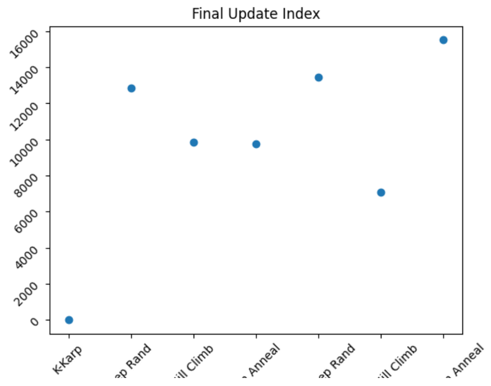
The mean and standard deviation of residue across 50 trials for each approach is shown below.

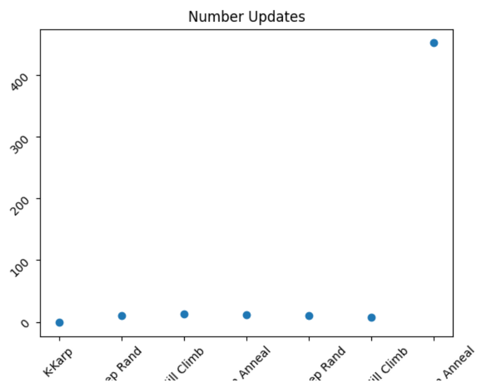
Alg	Num Updates	Last Upd Index
Rep Rand	10	12,839
Hill Climb	13	9,828
Sim Anneal	11	9,739
PreP Rep Rand	10	13,461
PreP Hill Climb	8	7,066
PreP Sim Anneal	452	15,507

Table 2. Algorithm Update Metrics (Over 50 Iterations)



We measure the average number of updates to the best yet encountered solution for each heuristic. We also measure the average of the index of the last iteration on which an update occurred. From the graphs below, it is clear that the simulated annealing algorithm with a pre-part representation updates the best yet encountered solution most frequently.




Number Updates
400
300
200
100
0
K-Karp
ep Rand
ill Climb
Anneal
ep Rand
ill Climb
Anneal