Number Partitioning

Lam & Tao

4/21/17

1 Dynamic Programming Solution

We would like to solve the number partition problem where we are given a sequence $X = (x_1, x_2, ..., x_n)$ of non-negative integers, by dividing X into two subsets where the sums of the subsets are as close to possible to each other, or the residue of the two sets is minimized. This problem is NP-complete, but we can use a dynamic programming algorithm to solve this in $O(nb)$, where b is the sum of the numbers in X. Our DP solution will look to determine whether there is a subset in X that sums to $\lfloor b/2 \rfloor$. If such a subset, Y, exists, then we have 2 cases. 1. If b is even, then $S - Y$ sums to $\lfloor b \rfloor$ as well. 2. Else, b is odd, then $S - Y$ sums to $\lceil b/2 \rceil$ which gives us the best possible solution.

The algorithm depends on the following recurrence for $D(i, j)$, which is true if there is a subset from the first i numbers, $x_1, x_2, ..., x_i$, that sums to j, and false if such a subset does not exist. $D(i, j)$ is only true if either two conditions are true. 1.) If $x_1, x_2, ..., x_{i-1}$ has a subset that sums to j, or 2.) if $x_1, x_2, ..., x_{i-1}$ sums to $j - x_i$. The first condition follows because if there is a subset, S, that already sums to j, a bigger set that includes S will simply use S to sum to j. The second condition follows because if

$$\sum_{k=1}^{i-1} x_k = j - x_i$$

, then

$$\sum_{k=1}^{i} x_k = j$$

We can define our recursion as:

$$D(i, j) = D(i - 1, j) \ \vee \ D(i - 1, j - x_i)$$

We denote the base cases as follows, where the first row/column of the matrix is 0-indexed:

$$D(n, 0) = True, \forall n \geq 0$$
$$D(0, j) = False, \forall j \geq 1$$

Therefore, our algorithm will build a matrix table of dimension $(n + 1) \times (\lfloor \frac{b}{2} \rfloor + 1)$, where the row index is i, the index of the last element, and the column index is the sum j. We fill this table from

$D(0,0) = True$, column by column, until we return $D(n, \lfloor \frac{b}{2} \rfloor)$, or the last $True$ value in the column, if the half sum is unable to be reached. Thus, the optimal residue value will be $b - j$, where j denotes the index of the designated column.

In order to actually retrieve the two partitions we can store pointers at each cell, where we have a pointer to a previous cell used to calculate that cell and whose value was $True$. For example, consider $D(i, j)$. In the case $D(i-1, j-x_i) = True$, we know that x_i will be included in the first set. In the case $D(i-1, j) = True$, we know that x_i will be included in the second set. Note that there could be multiple optimal solutions.

This algorithm needs to fill an $(n+1) \times (\lfloor \frac{b}{2} \rfloor + 1)$ table and each entry is constant time to fill. Therefore, this algorithm is order $O(nb)$ time.

Because this algorithm takes up a table of $(n+1) \times \lfloor \frac{b}{2} \rfloor$ and the space to store the pointers is constant, this space complexity is $O(nb)$ as well. Note that if we did not have/wish to store the solution, we could simply keep track of 2 columns; thus, our space would be reduced to $O(b)$.

2 Karmarkar-Karp Runtime

We construct a max-heap from the elements in A. This is done in $O(n)$ time. We perform $ExtractMax$ twice, where $ExtractMax$ removes the element with the largest value from the heap and fixes everything (using $MaxHeapify$) so that the heap structure is maintained. Each is done in $O(\log n)$ time. We compute the difference, d, between the two numbers and $Insert$ d into the heap, making sure to maintain the heap properties. This is done in $O(\log n)$ time. We repeat this process, stopping when the two numbers we remove were the last two numbers in the heap. The difference between the two final numbers will give us our desired residue. We call $ExtractMax$ at most $2n$ times (we reduce the number of elements in the heap by 1 each time after performing $ExtractMax$ twice and inserting back) and $Insert$ at most n times. Thus, the algorithm takes $O(3n\log n)$ or $O(n\log n)$. Correctness of this algorithm follows from the correctness of the heap operations.

3 Algorithm Implementation

We used the $uniform\_int\_distribution$ library available in C++ to uniformly select a random integer in the range $[0, 10^{12}]$ to build our array A.

For each trial, we started with different random initial starting points. We fed each instance to the 7 algorithms. We let each of the 3 non-partitioned algorithms start with the same random solution. We did the same for the pre-partioned algorithms, letting them start with the same random partition. This was done to reduce the variance based on the starting point amongst the algorithms.

We implemented KK using a binary heap so as to achieve a runtime of $O(n\log n)$ for KK.

In the algorithm implementation themselves, we saved both time and space by passing in an vector by reference in various function methods and making a mutation on the desired index(es), rather than creating a whole new array and coping over all its values. Thus, going from a linear time operation to a constant time operation. We also worked to optimize on memory by avoiding unnecessary initialization and copying of vectors.

4 Algorithm Results

4.1 Mean Residue and Standard Deviation of Karmarker Karp Algorithm:

Mean Residue: 316050
Standard deviation: 375736.747177

4.2 Mean Residues of No Partition Algorithms:

Repeated Random: 286240094
Hill Climbing: 309638483
Simulated Annealing: 299349738

4.3 Standard Deviations of Residue of No Partition Algorithms:

Repeated Random: 279808687
Hill Climbing: 296807527
Simulated Annealing: 346079736

4.4 Mean Residues of Partitioned Algorithms:

Repeated Random Pre Partitioned: 165
Hill Climbing Pre Partitioned: 648
Simulated Annealing Pre Partitioned: 218

4.5 Standard Deviation of Residue of Partitioned Algorithms:

Repeated Random Pre Partitioned: 139
Hill Climbing Pre Partitioned: 874
Simulated Annealing Pre Partitioned: 197

4.6 Graphs

Karmarker Karp produces the best residue out of the three standard heuristic algorithms. This is because Karmarker Karp's residue is guaranteed to be less than or equal to the largest element due to the algorithm's selection process. This means that the final residue for KK is smaller than the range of potential residues for the standard heuristic algorithms. While Repeated Random is the best residue amongst the 3 standard heuristics, it still has to search out of 2^{100} possible solutions, a very large solution space; this applies to the other 2 nonpartioned algorithms as well.

Considering the nonpartioned algorithms, we note that Repeated Random produces the best residue. From this we can deduce that the solution space does not have one local optimum, but rather many local optima—a result of the set up of neighborhoods, and our results confirm this theory. The Hill Climbing algorithm, which looks at neighbours and moves if it finds a lower residue, will get stuck at one of these

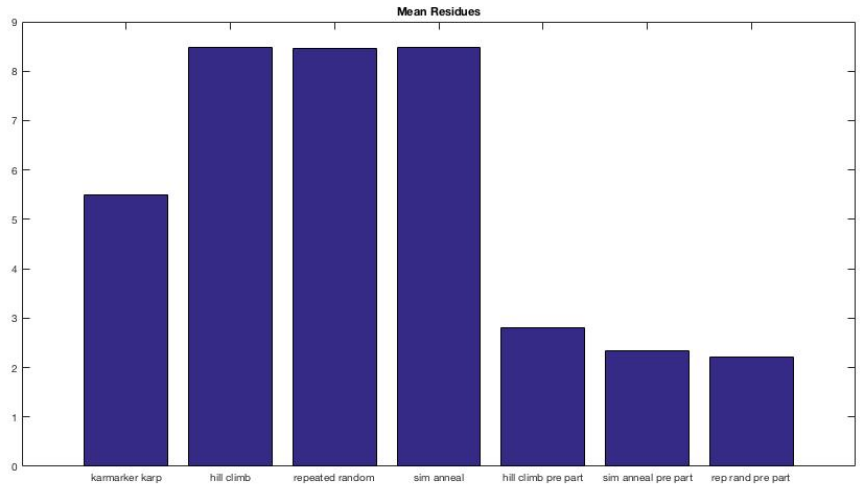

Figure 1: Mean Residues (on a log 10 scale). Notice KK is a middle ground between PP and NP algorithms

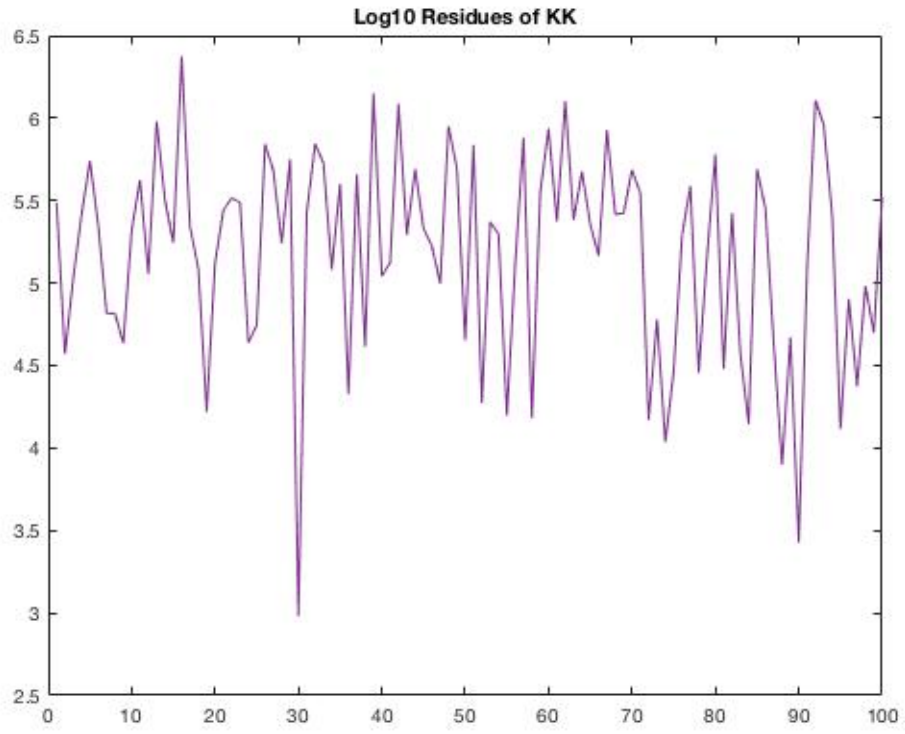

Figure 2: KK Residues

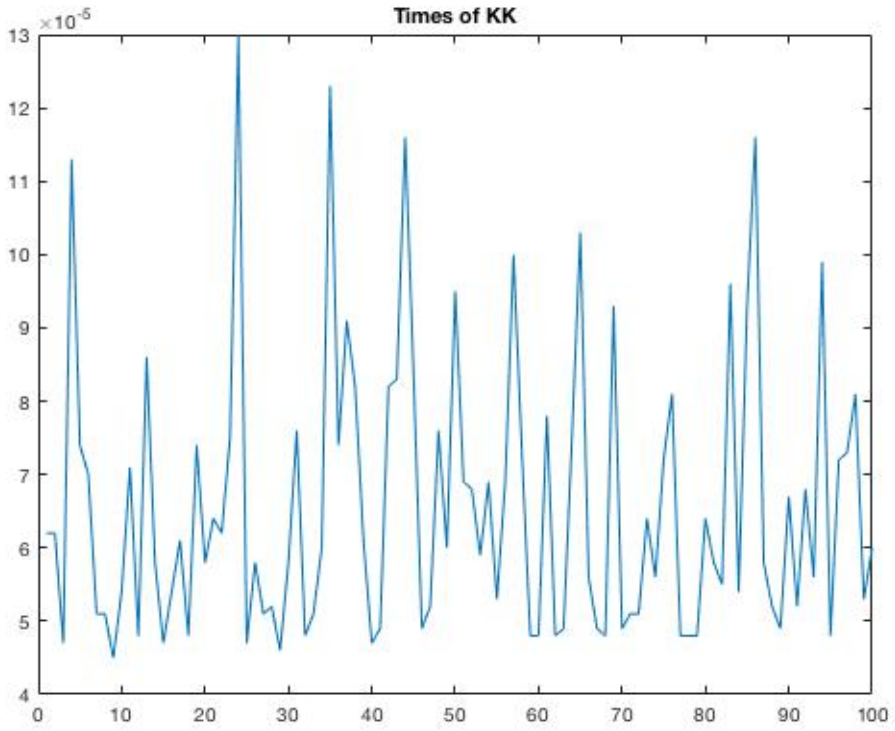

Figure 3: KK Times

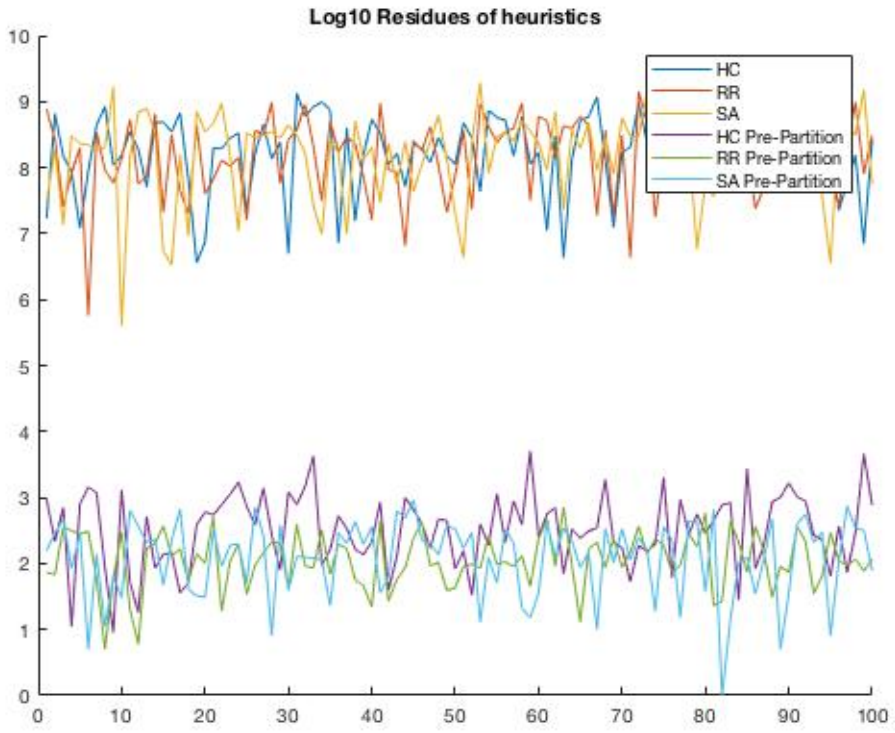

Figure 4: Heuristic Residues, both pre-part and normal (on a log 10 scale)

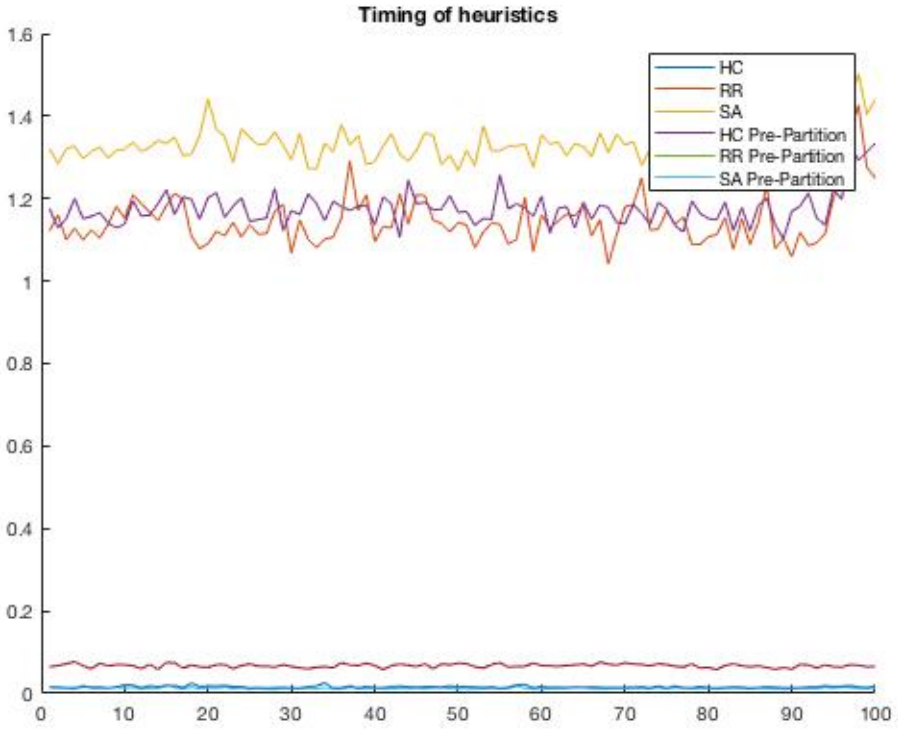

Figure 5: Heuristic Times, both pre-part and normal (on a log 10 scale). NP algorithms take less time due to PP algorithms calling KK to compute residues

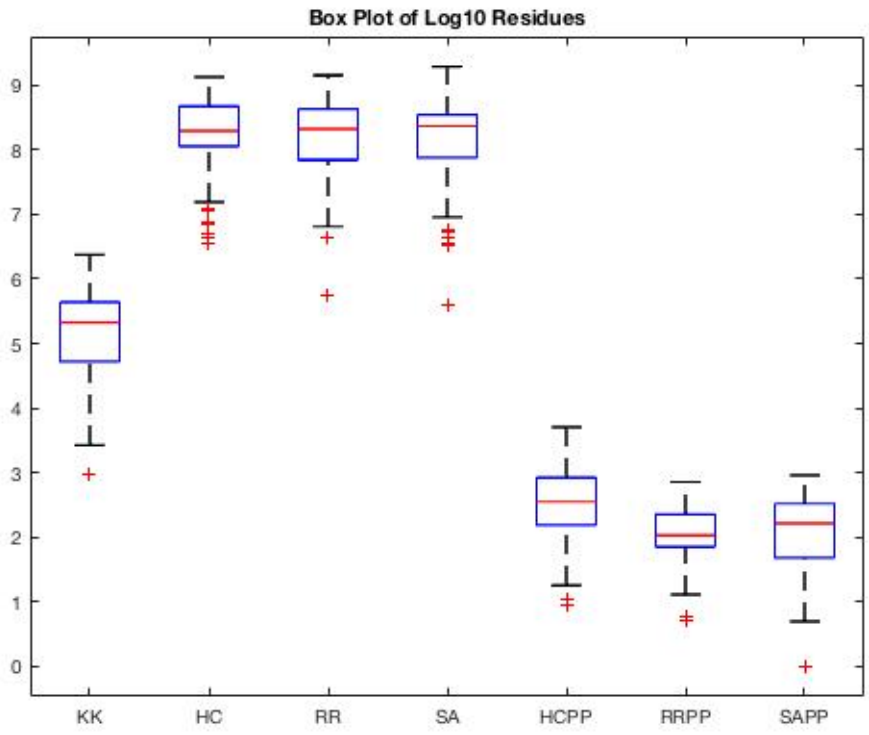

Figure 6: Box plots of the residues, showing medians and outliers

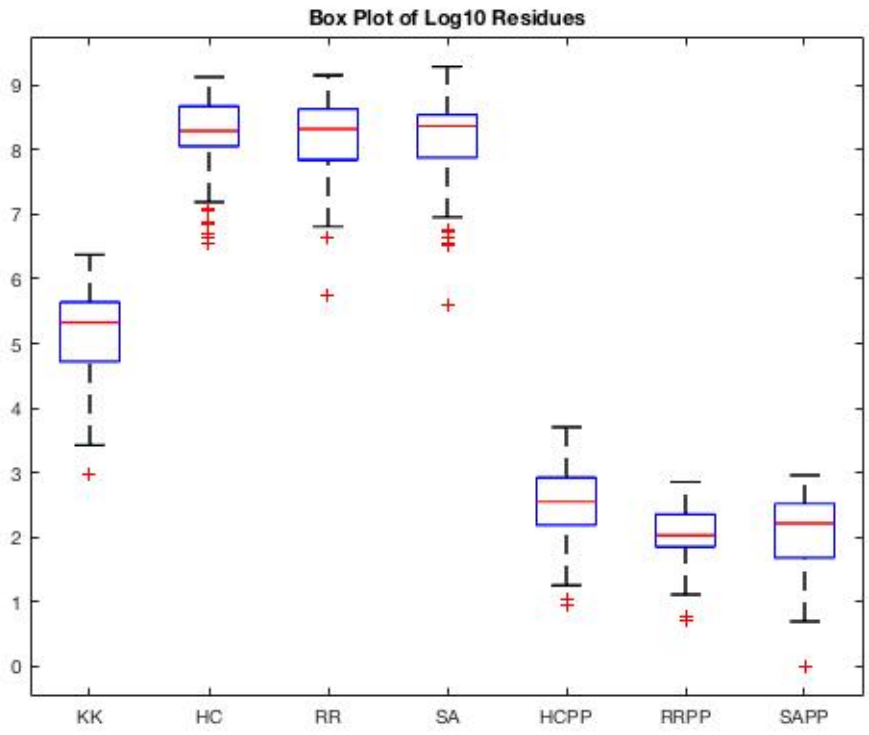

Figure 7: Box plots of the times, showing medians and outliers

local optima and has no chance of escape. In the Simulated Annealing algorithm, there is a probability of escape to a different optima hence its better result, and in the standard Repeated Random algorithm, solutions do not get stuck in local optima but nor do we continue searching down the path of the local optima due to the randomly generated solution.

We notice after these trials that the pre-partitioned heuristic algorithms perform better than the non pre-partitioned algorithms. We note that because the pre-partition algorithms run KK after the initial partition, the final residue value has an upper bound of the largest element in the pre-partitioned array. From the experiments, we can deduce that a random move on S is more constricting than a random move on P. The former partitions into 2 sets while the latter partitions into n sets and lets KK handle the rest. Consequently, because we are using the partition representation of the number list, we can move elements into n partitions, while in the no partition representation, we can only move an element into 1 different set. We can speculate that prepartioning sets up neighborhoods in such a way that the solution space is a lot smoother, smaller than that of the non-partioned algorithms, and easier to move towards optima. Additionally, as the neighborhood space for the prepartioned algorithms is much larger, we are more likely/able to find a path leading to a better solution.

When analyzing the data for the prepartioned algorithms, the distinction between the 3 is made clear. We see that the Simulated Annealing algorithm performs better than the Hill Climbing algorithm as it can avoid getting stuck in a local optimum. We also see that Repeated Random performs well and even a little better than Simulated Annealing. We deduce that the solution space is small and when running for 25000 iterations, it is likely that a small solution can be found; this applies to all 3 prepartioned algorithms. We note that Simulated Annealing may take some time to detrench itself from a local optimum, while Repeated Random has the flexibility of jumping around to avoid getting entrenched in the first place. However, the results between the two are very similar, which shows that both algorithms can be promising.

To provide some analysis on the time complexity, we see that the prepartioned algorithms take longer than the nonpartitioned algorithms. This makes sense as for each of the 25000 iterations for the prepartioned algorithms we must convert an array based on a given partition solution, $O(n)$, and run KK which takes $O(n\log n)$ as versed to just one linear search, $O(n)$, to compute the residue for the nonpartioned algorithms. Finding neighbors for both pre and nonpartioned algorithms is done in constant time, except for repeated random which is done in $O(n)$ time. If we were to analyze time based on the number of iterations, i, the nonpartioned algorithms would run in $O(in)$. The prepartioned algorithms, on the other hand, run in $O(i * n\log n)$.

From the results, we see that the additional linear search (also note the additional calls to the *rand* function) that nonpartioned RR takes is reflected in its time being slightly higher than the other nonpartioned algorithms. Additionally, we see that the prepartioned algorithms take significantly longer than the nonpartioned. We note that constant factors that big O complexity ignores are at work here (KK taking $O(3n\log n)$). Overall, the time from our data corroborates our analysis.

We subsequently ran 40000 iterations and obtained the following results:

4.7 Mean Residues for 40K Iterations:

Repeated Random: 191601961.64
Hill Climbing: 212065624.44
Simulated Annealing: 161645961.28
Pre-Repeated Random: 108.62

Pre-Hill Climbing: 530.58
Pre-Simulated Annealing: 133.5

We can see that with more iterations, all of the algorithms improve their residue value. From these results, one interesting thing that we can deduce is that the reg-HC and reg-SA seem to have most likely been stuck in local optimum bound which took a considerable amount of time/iterations to move towards the local optimum itself. Overall, these results further corroborate our analysis and discussion above.

5 Karmarkar-Karp as a Starting Point

We note that for the standard representation algorithms KK performs significantly better. Thus, KK as a starting point would improve the final return residue. We see that the pre-partioned representations perform better than KK. We consider KK as a starting point for each algorithm.

5.1 Repeated Random

Based on the experimental results, starting RR with the KK starting point would most likely be the best solution that RR will find amongst all the 25000 iterations. However, this starting point will have no influence/effect on future iterations as we are finding a completely new, random solution at each iteration.

5.2 Hill Climbing

By itself, we see that Hill Climbing gets stuck in a local optimum that is no where close to the global optimum. Thus, using KK as a Starting Point would be a significant improvement, and could further lead this algorithm to an even better solution. Consider the example $(10, 8, 7, 6, 5)$. KK would yield a split of $(10, 7)$ and $(8, 6, 5)$. Switching 8 and 7, a valid random neighbor move, would yield the optimal solution. However, in general, this does not guarantee that using KK as a starting point will lead to finding the global optimum. As we are essentially doing a prepartition where elements are either grouped in 1 or 2, there is no guarantee that this prepartition will lead to the global optimum. Thus, we could still get stuck in a local optimum.

5.3 Simulated Annealing

This will have the advantages mentioned above in Hill Climbing, except that now we can allow moves that will potentially move us out from getting stuck in a local optimum. However, as we will be fairly entrenched in this local optimum, there is no guarantee that we will be able to get out by the time the iterations have finished and that finding the next/subsequent local optimums will be better than the one we find when using KK as a starting point.

5.4 Pre-Repeated Random

Using KK with a starting point will most likely make no difference, as we see that this algorithm will randomly and eventually find a more optimal solution on its own.

5.5 Pre-Hill Climbing

Starting with partitions in strictly 2 sets as versed to n sets could initially restrict the flexibility of the pre-partioned algorithm by itself, which based on the experimental results was not necessarily optimal. This algorithm can eventually make neighbor changes that will move it a better solution; however, again it faces the danger of getting stuck in a local optimum that was set when using KK as a starting point. We see from the experimental results that Pre-HC gets stuck in a local optimum solution.

5.6 Pre-Simulated Annealing

Again, the same considerations apply for this algorithm as did for Pre-Hill Climbing, except now it is possible to move out of local optimums in search of a more optimal solution. We see based on the experimental results, that Pre-SA performs better than that of Pre-HC, as it avoids getting stuck in the first local optimum that it finds. Using KK as a starting point may entrench Pre-SA in a local optimum bound that could take some time to get out of. This algorithm, and Pre-HC as well, may be better off not using KK as a starting point to allow for greater initial flexibility.