CS 124, Programming Assignment 3

80940475 and 10977908

April 2017

1 Dynamic Programming

A is our original array = $\{a_1, a_2....a_n\}$. Let's refer to the summation of all the values of A by the variable b. Basic idea: Find a subset of A that tries to sum up to b/2 or as close to b/2 as possible. We do this by finding every sum that is possible to exist with different subsets of the elements in A and filling an array M with that.

We have a 1D array called M which is size [b/2]. This array is of type booleans. M[0] is set to True and everything else is set to False.

We will go along all the elements in A (let's call all the elements in A, a_i where $1 \leq i \leq n$). Then we will iterate through array M and for all j that M[j] is true, we will set M [a_i + j] true as well. We will stop when M[b/2] becomes True or when we run out of elements in A.

Let's try an example, if A = $\{5, 7...\}$. M[0] is already set to True. When we look at a_1, we set M[5+0] to true. Now we consider a_2 and set both M[0+7] and M[5+7] (M[12]) to true. We keep doing this until we set M[b/2] to True.

However, to obtain the two subsets we will need to have another array. Let's call that array S. This is also size b/2. Whenever we set M[i] to be true, we store a_i in S[j+a_i]. E.g if $a_i = 7$ and adding 7 results in 12, we store 7 in S[12].

How do we find the residue and 2 subsets from M and S? We look in M and if M[b/2] = True, we know that the residue is 0. If not, we keep traversing through M backwards from M[b/2] until we hit the first True. Let's say this is at value y. b/2 - y is the value of our residue.

Now let's look at how we reconstruct the subsets while using S. Look in S[y] and see the number stored there. Add this to the subset. Eg our obtained value of y (closed approximation to b/2) is 12 and in S[12], 7 is stored. Now we know that 7 is in the subset. Now do 12-7 and look in S[5] and see what is stored there and add this to the subset. Keep doing this until you reach S[0] to obtain all the elements in one subset. Automatically, all the other elements in A are in the second subset.

Space efficiency is O(2 * b/2) because of array M and array S of size b/2. This results in a space of O(b).

The running time is O(n * b) where n is the number of elements in A (the original array). This is because we iterate through all the elements n and for each element we go through the two arrays (b/2 + b/2 = b).

2 KK in O(n log n)

We use a max heap, the creation of which is O(n). At each point, we delete the top two values, take their difference and insert the difference into the max-heap. Insertion and deletion takes O(log n). We keep doing this until the heap is empty and we insert the last difference. We do the deletion 1.5n times and the insertion n/2 times. Hence, 2n insertion and deletions are performed.

The total running time is O(2nlogn + n) which approximates to a running time of O(nlogn).

Proof of correctness: As Karmarkar-Karp says, we repeatedly take the maximum 2 values and insert their difference into the data structure until only one value is left (the residue).

3 Functions

Please run the program by typing ./kk.py *input_file*

Note: In my write up, I refer to the standard solution as the binary solution.

- instance: creates an initial array with 100 elements
- *randsolgen_binary*: creates a random solution in the form of the binary representation
- randsolgen: creates a random solution in the form of the prepartition solution
- karmKarp: returns the residue of an array by repeatedly sorting the list, taking the top 2 maximum values, deleting them and inserting their difference into the array until we are left with only one value (the residue). I did not implement this using a heap as this version of Karmarkarp was really fast (running in fractions of a millisecond)
- *randadj_bin*: produces a neighbour of the binary solution
- randadj: produces a neighbour of the prepartition solution
- randtrans: produces A' using the prepartition solution
- residue: calculate the residue using an array and the binary solution
- *reprand_bin*: the repeated random algorithm on the binary representation
- *hill_bin*: the hill climbing algorithm on the binary representation
- *sim_bin*: the simulated annealing algorithm on the binary representation
- *reprand*: the repeated random algorithm on the prepartition representation
- *hill*: the hill climbing algorithm on the prepartition representation
- *sim*: the simulated annealing algorithm on the prepartition representation

4 Data

We ran our program on 100 instances and calculated the average times and the average residues by 1. Karmarkar Karp 2. repeated random, hill climbing and simulated annealing (on binary representation) 3. repeated random, hill climbing and simulated annealing (on pre-partition representation).

Our results, that are shown in the table and graphs below (there are more graphs in the Appendix, two for each type of algorithm) show that binary representation results in the worst residues (9 digits long) for all 3 algorithms. The prepartition representation was the best as it produced residues of 3 digits long for all 3 algorithms. Karmarkar Karp produced average residues in between these two representations, which were 6 digits long.

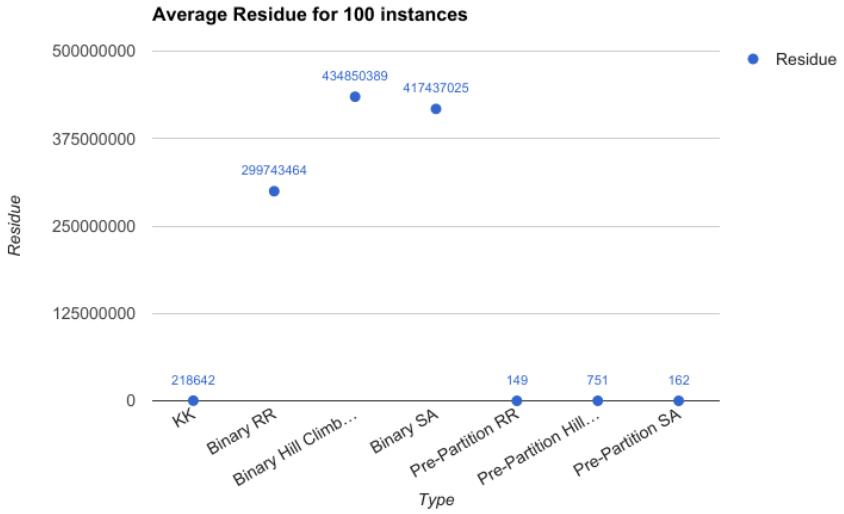
However, the times showed the opposite scenario: Karmarkar Karp did the best (0.00045 seconds = .45 milliseconds), followed by the hill climbing algorithm on the binary representation (.9 seconds). Running repeated random and simulated annealing on the binary representation was also quite fast (3.6 and 2.3 seconds respectively). However, running the algorithms on the prepartition representations took much longer e.g. repeated random took an average of 26 seconds, hill climbing took slightly less (22 seconds) and simulated annealing took a whooping 62 seconds.

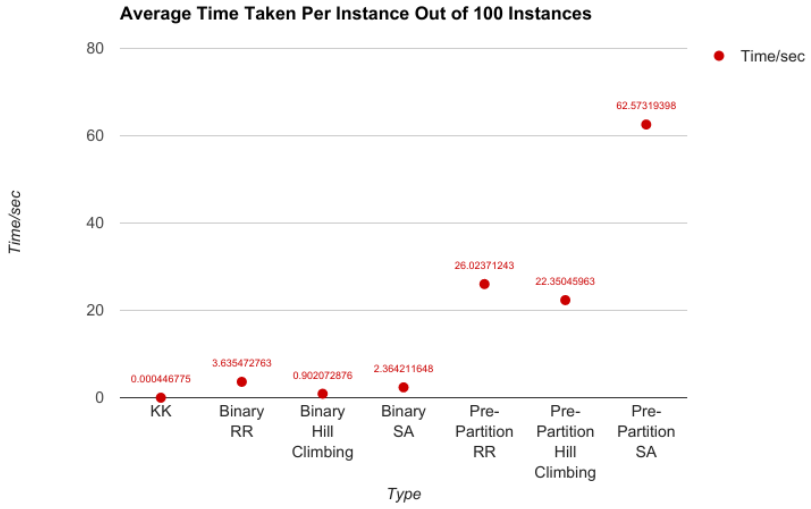
The scatter plots in the appendix show the values of time and residue for all 7 types of tests and for 100 instances. The graphs show that the values of time and residue do cluster together for each type of algorithm. There are some outliers, but for the most part, the residue and times fall in between certain bounds.

Type	Residue
KK	218642
Binary RR	299743464
Binary Hill Climbing	434850389
Binary SA	417437025
Pre-Partition RR	149
Pre-Partition Hill Climbing	751
Pre-Partition SA	162

Type	Time/sec
KK	0.000446775
Binary RR	3.635472763
Binary Hill Climbing	0.902072876
Binary SA	2.364211648
Pre-Partition RR	26.02371243
Pre-Partition Hill Climbing	22.35045963
Pre-Partition SA	62.57319398

Average residues and time taken for 100 instances





5 Analysis

- It makes sense that KK was the fastest as it was only run on one iteration while for everything else, the iterations were 25000. It also makes sense that running the algorithms on the prepartition representation took so much longer than running them on the binary representation as the former also involves transforming the solution and deriving A' while dealing with very large numbers. Moreover, it also involved repeatedly calling

Karmarkar Karp which resulted in the times adding up.

It makes sense that hill climbing took the least amount of time, since the way that neighbors are generated for hill climbing involves randomly changing just one to two numbers in the solution, which is close to (if not) constant time. Simulated annealing on binary was second and then repeated random on binary was third. Repeated random is slower because of how many times it has to generate a random solution from scratch. Simulated annealing was slower than hill climbing since it calculates residues a couple of times and does a few more probability calculations, and is faster than repeated random, because it uses neighbors as opposed to completely new solutions. This reasoning holds for both binary and prepartition representation.

In prepartition, hill climbing was the fastest followed by repeated random and then simulated annealing. Simulated annealing took the most time as expected because it involved a few probability calculations and comparisons between residues, the latter done by running Karmarker-Karp on large numbers.

- We were able to find better solutions using prepartitioning, because it forces elements together as opposed to setting them apart. Thus, it feeds into KK (which is already a pretty good heuristic) as opposed to providing a completely random solutions.

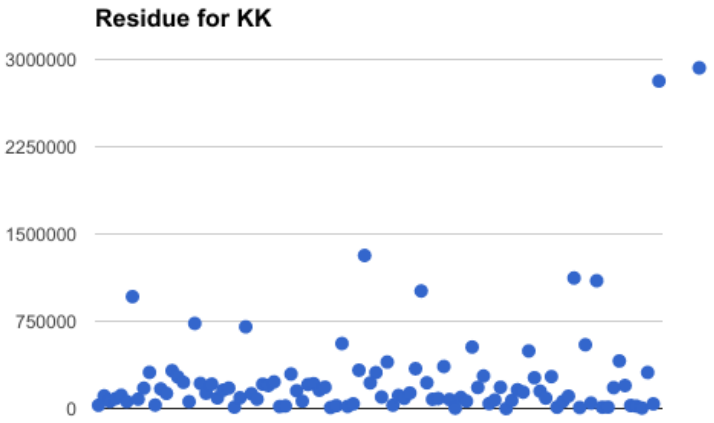
  In prepartitioning, repeated random is the best at finding a solution, again because of its greater variety of random options, followed by simulated annealing (which unlike hill climbing is designed to avoid getting stuck in a local optimum) and then hill climbing (which not only uses neighbors with very little difference between each other, but can also easily get stuck in a local optimum). This reasoning applies to both representations.

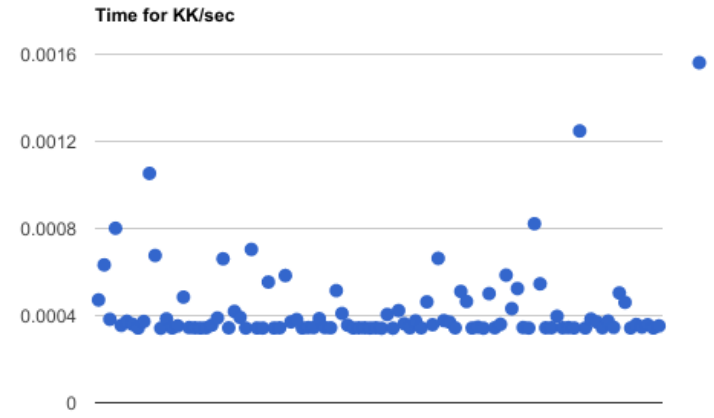
  In the standard representations, as in the binary the hill climbing did worst, followed by simulated annealing and then repeated random (which still was pretty bad).

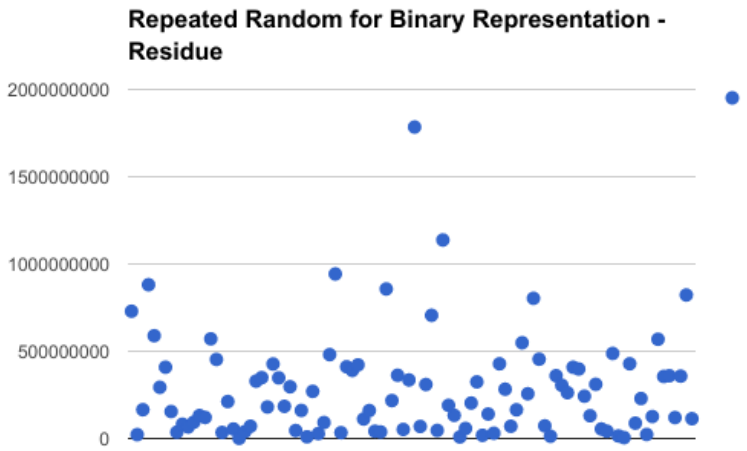
6 Using the solution from KK

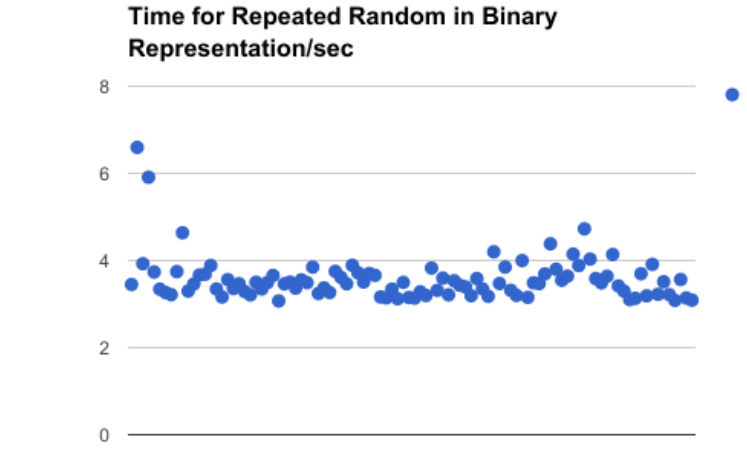
KK produces a pretty good residue as shown by the results in the table above (it is in between binary and prepartition representation). Hence, if we start with the subsets produced by KK as the starting points of our local search algorithms (like hill climbing and simulated annealing for example), that will automatically improve our answers as we start with something that is more accurate than a random array and would keep improving the residue. It is still possible to end up stuck in a local optimum in hill climbing, however it would still be a better solution than a completely random-neighborhood hill-climbing one (as proven by our results).

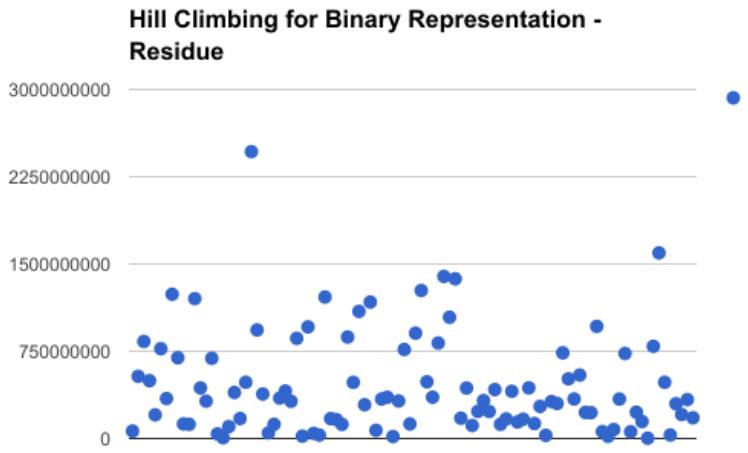
7 Appendix: Graphs

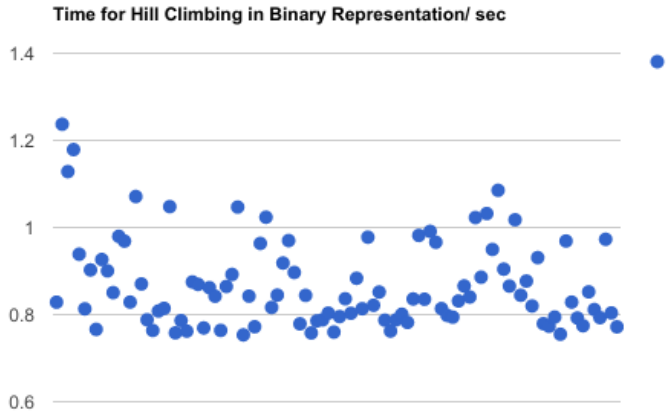


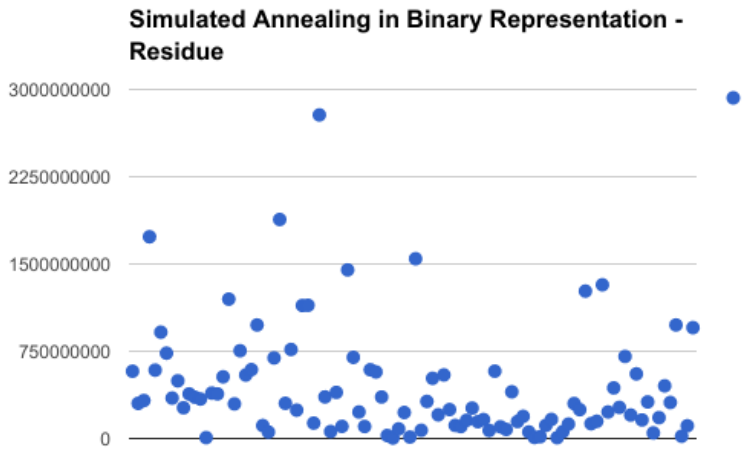


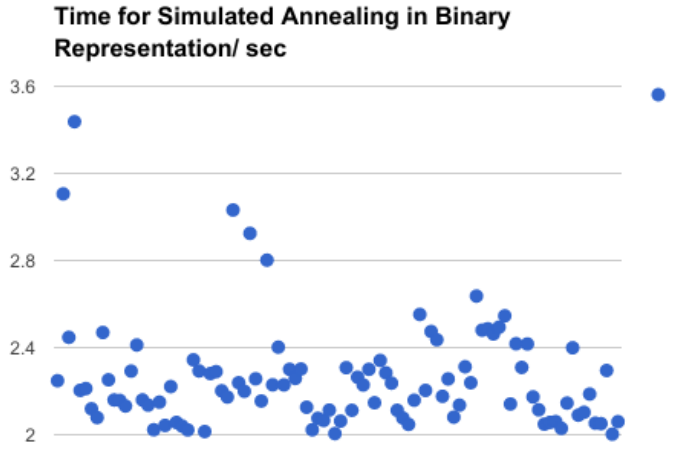

Repeated Random for Binary Representation - Residue
2000000000
1500000000
1000000000
500000000
0

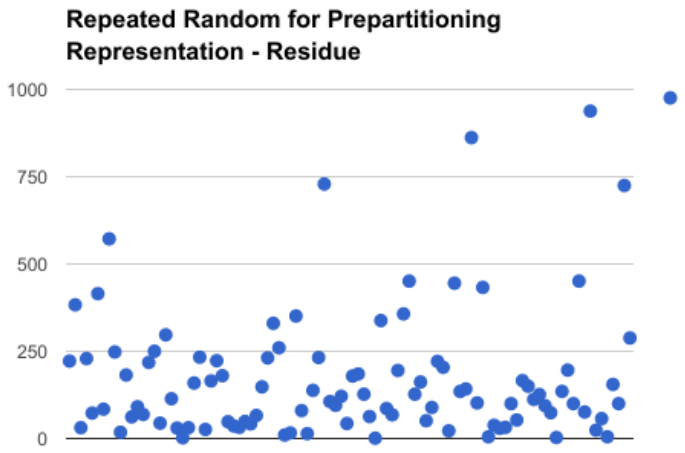

Time for Repeated Random in Binary Representation/sec
8
6
4
2
0

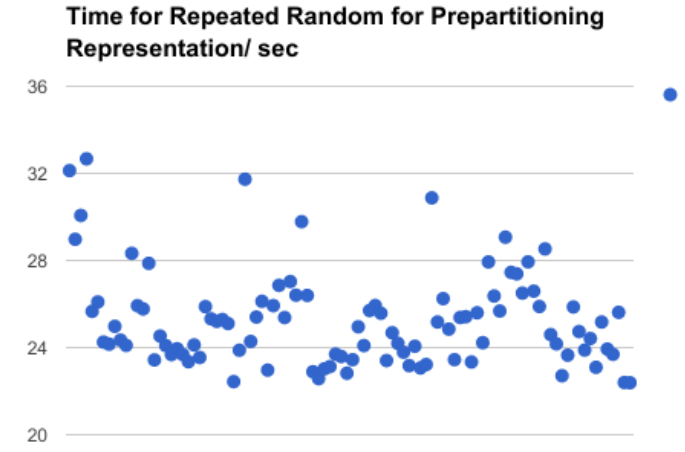

Hill Climbing for Binary Representation - Residue
3000000000
2250000000
1500000000
750000000
0

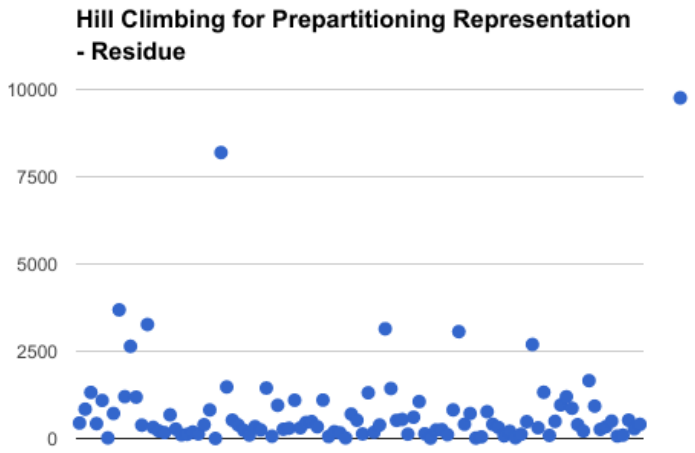

Time for Hill Climbing in Binary Representation/ sec
1.4
1.2
1
0.8
0.6

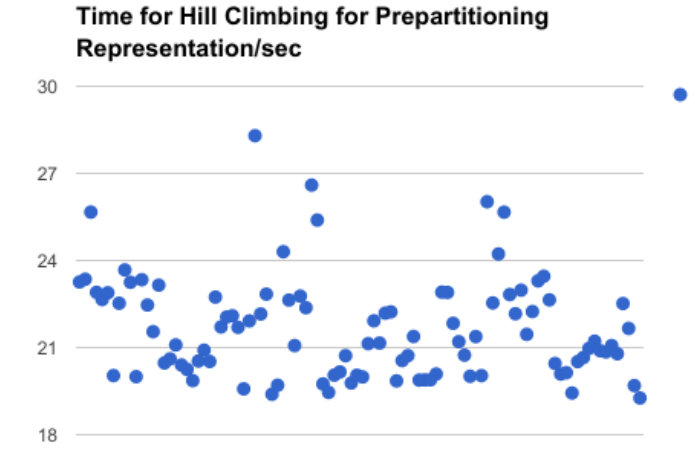

Simulated Annealing in Binary Representation - Residue
3000000000
2250000000
1500000000
750000000
0

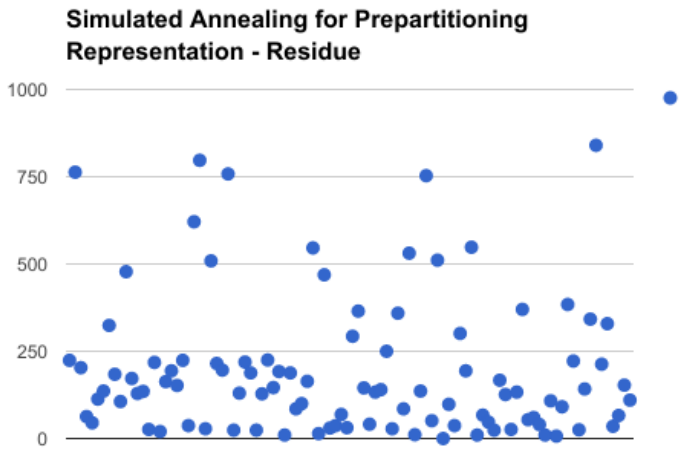

Time for Simulated Annealing in Binary Representation/ sec
3.6
3.2
2.8
2.4
2

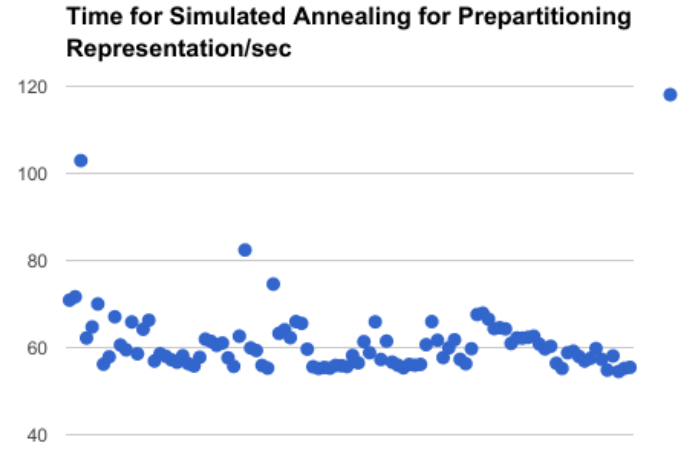

Repeated Random for Prepartitioning
Representation - Residue
1000
750
500
250
0


Time for Repeated Random for Prepartitioning
Representation/ sec
36
32
28
24
20


Hill Climbing for Prepartitioning Representation - Residue
10000
7500
5000
2500
0


Time for Hill Climbing for Prepartitioning Representation/sec
30
27
24
21
18


Simulated Annealing for Prepartitioning Representation - Residue
1000
750
500
250
0


Time for Simulated Annealing for Prepartitioning Representation/sec
120
100
80
60
40