Programming Assignment 3
CS 124

Artidoro Pagnoni, Nisha Swarup

April 25, 2016

Summary

1 Dynamic Programming Solution to Number Partition Problem

1.1 General Idea

Although the Number Partition Problem is NP complete, we can solve it in pseudo-polynomial time. Let b be the sum of all n integer entries of A. The solution that we will propose is polynomial in nb. In the Number Partition Problem we are essentially creating two sets with the least possible difference.

Our solution lies on the computation of all possible partitions of the initial subset of elements into two sets. We only care about the possible values that can be reached by the sum of the elements in every partition. Since the sum of all entries of A is b, we are interested in knowing all the possible values in the range $[0, b]$ that can be reached by creating a subset of the first elements of A and summing over all elements of the subset. We can imagine this information is stored in an array of length b, in which a 1 in position j means that j is the result of the sum of the elements of the first i elements of A. If j is not the sum of the elements of a subset, then the jth entry in the array will be 0. Given such an array we can find the value that is closest to $b/2$. The closer we get to $b/2$, clearly the lower the residue between the two subsets will be.

1.2 Recurrence

We can calculate this solution recursively. The recursion is done on the initial set. At every step we include one more element in the initial set, eventually covering the whole set. Given the solution array at one step, we can calculate the solution array at the next step in the following way. All the values that could be reached at the previous step can still be reached (we can choose the same subsets, and just discard the new element). In addition, we can choose subsets that include the new element. This will increase the value of the sum of their elements by the the new element.

We can formalize this by filling out a matrix M of size $n \times b$, such that the rows correspond to the solution arrays mentioned above. The first row is the base case, and corresponds to the empty set (when we don't include any element of A). As we increase the row number, we add more elements of A to the initial set. The columns correspond to the values of the sum that need to be reached. We therefore have:

Recurrence Relation:

$$
\begin{aligned}
M_{(0,0)} &= 1 && (1)\\
M_{(0,j)} &= 0 \qquad \text{for } j \neq 0 && (2)\\
M_{(i,j)} &= \max\{M_{(i-1,j)}, M_{(i-1,j-A_i)}\} && (3)
\end{aligned}
$$

$M_{(i,j)}$ is 1 if we can reach j with the sum of the first i elements of A. The base case is clear, with no elements we can only reach value 0.

The recurrence equation is also quite straightforward. The first term is 1 when you could reach j at the previous step, (without the new element). The second term is 1 when we can reach j by adding A_i (the new element) to any of the values at the previous step. The max ensures that if any of the two terms is 1, we will get 1 out (it serves the same purpose as an or, if we use booleans instead of 0s and 1s).

We fill out the matrix row by row from left to right, following the recurrence equation.

Once we have the matrix, we will only consider the last row, and find the closest reachable value to $b/2$. We can therefore run through the last row starting at $\lfloor b/2 \rfloor$ and finding the first non zero entry. One subset will have the value corresponding to the index of the column of the first non zero entry, and the other will have the complement of that index $(b-j)$.

The residue can be calculated by taking the absolute value of the difference of the values found for the two subsets.

1.3 Reconstruction of the Subsets

This algorithm finds the minimum residue but does not tell us how to partition the set A. If we want to construct the two subsets, we need to fill out an addition matrix, of the same size. The matrix will be composed of pointers to the parent entry. By parent we mean the entry that was equal to 1 in the equation $M_{(i,j)} = \max\{M_{(i-1,j)}, M_{(i-1,j-A_i)}\}$. If both terms were 1, pick the first.

To reconstruct the subset we need to find for every element if they belong to one set or the other. We will start with the last element in A. We take the last row in the Pointer's matrix, and the column that corresponds to one of the subsets,(the closest possible to $b/2$). If the pointer is pointing to the same column then the element in consideration is in set 1, if not it is in set 2. The next element to be considered is the one pointed to by the pointer, and we repeat the same reasoning until we get to the first element.

1.4 Further Optimization

We notice that we don't really need to complete the right half of either matrices. It is a question of symmetry - if one set has a total sum below $b/2$, the other will be above. Therefore, we need to make sure to fill out the matrix up to column $\lceil b/2 \rceil$. We will start at $b/2$ and go down in the last row when looking for the closest reachable entry to $b/2$. This saves half of the time and space, but will not change the asymptotic running time.

1.5 Space and Time Complexity

In terms of space complexity, this algorithm is $O(nb)$, since we need to fill out two matrices of size $n \times b/2$.

The time complexity is polynomial in nb more precisely it is $O(nb)$ since we need to fill out the entire matrix and every entry requires a constant number of operations (two comparisons and writing to both the solution and the pointer matrix). This confirms that we have a pseudo-polynomial solution for the Number Partition Problem.

2 Karma Karp Algorithm Time Complexity

The Karma Karp algorithm can be implemented in $O(n \log n)$. We will describe the algorithm that we have implemented.

We construct a binary Max Heap with the initial set. Inserting something in the Max Heap takes at most $O(\log n)$, and we are doing it for every element in the set. Therefore the total time from the construction of the Max Heap is $O(n \log n)$.

Once we have the Max Heap constructed, we need to extract the two largest elements. This corresponds to two Delete Min operations, which take $O(\log n)$ each. We then take the difference of the two elements and insert it back to the Max Heap, also with time $O(\log n)$.

We repeat the last step exactly $n-1$ times. At every step we reduce the size of the Max Heap by 1 (two delete min and one insert). We have an initial size of n and we need to get down to 1, which takes $n-1$ steps.

Now putting everything together we have a run time of:

$$\begin{aligned}
& \text{Construct Max Heap} + (n-1)(2 \text{ Delete Min} + \text{Insert}) && (4)\\
= \; & O(n \log n) + (n-1)(2 * O(\log n) + O(\log n))) && (5)\\
= \; & O(n \log n) + O(n \log n) && (6)\\
= \; & \boxed{O(n \log n)} && (7)
\end{aligned}$$

This shows that our algorithm implements Karma Karp in $O(n \log n)$.

3 General Discussion of our Implementation

3.1 Description of Algorithms

We used Python to complete this assignment.

Our KK algorithm relies on a binary Max Heap.

For each representation we had 3 functions: calculate-residue, generate-random-soln, and random-move, which are self-explanatory.

Random move returns an array of swaps to make on the random move. We then passed these functions in as parameters to the functions for each of the three algorithms that we wanted to implement. For the simulated annealing function, we used the suggested cooling schedule. This affects the probability of keeping a worst solution.

3.2 Complications

One complication we faced was that Python is neither a call-by-value nor a call-by-reference language, but a mixture of the two since it is call-by-object. We spent a while debugging one method since the immutable / mutable object dichotomy is a very different framework than that of Java or C++. This was a very valuable learning experience in how to maintain objects to write good Pythonic code.

4 Results and Analysis

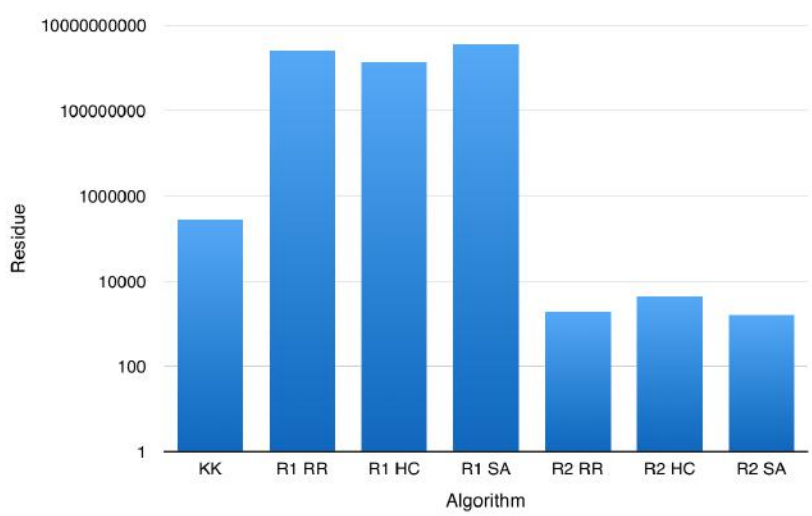
4.1 Relative Performances

We see that representation two takes significantly more time to run, but also has significantly better performance among all three of the algorithms. KK is very fast because we are only running it once. It makes sense that repeated random takes slightly longer than the other two algorithms since we are generating more random numbers, which makes a difference over 25,000 trials. It makes sense that representation 2 takes more time than representation 1, since representation 1 just adds numbers to calculate residues which takes linear time, whereas representation 2 has to run KK which will take $O(n \log n)$ every time.

Averages

Algorithm	Residue	Time
KK	272399.4	0.000631371
Representation 1: Repeated Random	2540197762	0.064256511
Representation 1: Hill Climbing	1364200936	0.031132083
Representation 1: Simulated Annealing	3626559951	0.032871208
Representation 2: Repeated Random	1907.88	2.309893794
Representation 2: Hill Climbing	4404.28	2.223660889
Representation 2: Simulated Annealing	1607.8	2.235860085

4.2 Differences in Solution Space



The above graph shows the average residues for each algorithm on a logarithmic scale.

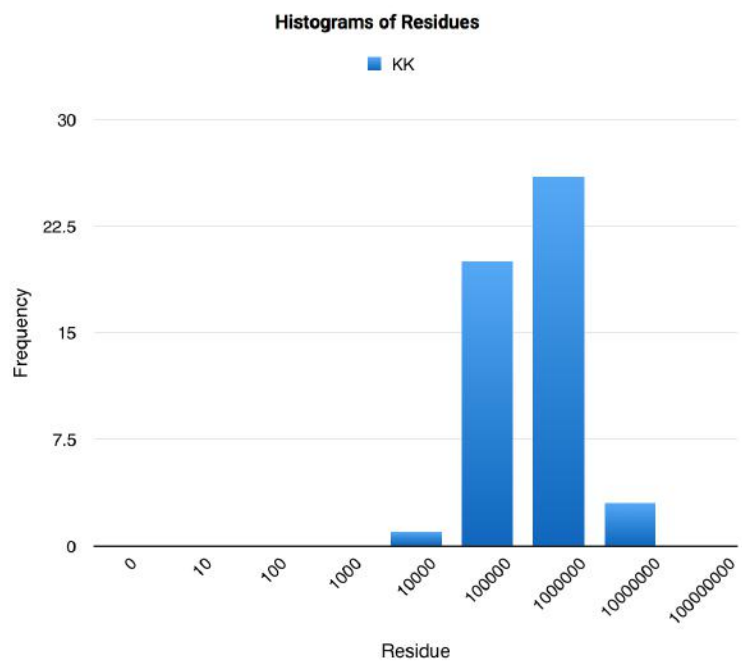
We see that KK performs better than the first representation, but worse than the second representation. This shows us that a random algorithms performance is heavily affected by the representation of solutions or in other words, how we set up the problem. This matters more than the actual algorithmic method that we use.

In the standard representation a neighbor's residue will vary significantly from its ancestor. On average, a change consists in swapping one element from one side to the other, and since we are considering extremely large elements (order of 10^12) we have very large differences. Even when we are swapping two elements, the expected difference in residue is still of almost the same order of magnitude as the elements (one less). This makes it harder to find a local improvement. We can understand this as the solution space being more bumpy when using the standard representation.

By setting up the problem with representation 2, we change how we choose moves between solutions and hence change how many local optima there are. In this case, the difference in residue between two neighbors can be smaller. It may seem counter-intuitive since prepartition will create individual elements that are very large. However, their size will be compensated within the algorithm by putting more larger elements on the other side. This creates a smoother solution space.

This means that the solution space for representation 1 would be more bumpy, whereas the solution space for representation 2 would have less bumps.

4.3 Karma Kart Residue Frequency Plots



Here is a histogram of residues for the Karmarker-Karp algorithm. We see that the algorithm mainly produces residues less than 100,000, although there will be outliers on the right. Residues of 100,000 are pretty good for random integers on the interval [0 to 1,000,000,000,000].

4.4 Frequency Tables in the Appendix

In the Appendix you will find frequency tables of residues for each of the algorithms and each of the representations. We used raw number frequencies instead of proportions, since there was an equal number of trials for each algorithm (50 instances). In the legends for the graphs, R1 means representation 1, R2 means representation 2, RR means repeated random, HC means hill climbing, and SA means simulated annealing.

4.5 Residue Plots

The graphs in the Appendix show how the residue changes by iteration on a new generated instance. We show the iteration on the x-axis and the residue on the y-axis. The residue is close to its final value after the 15,000th iteration, and this is because the solution space gets flatter as we get closer to optimal value. This is a characteristic of constraint optimization problems.

We also see that the residue for the second representation RR in this particular instance that we generated was under 100. This goes to show how much results vary by the instances that are generated, due to randomness.

Finally, we also notice that for the Standard Representation the improvements in the residue values are always very large. In the Prepartition representation the improvements are large at first but get to less than 1000. The smaller improvements allow for a lower overall residue. This confirms that the solution space in the Prepartition Representation is smoother and allows for small improvements.

4.6 Distribution of Repeated Random Results

The Repeated Random algorithm has similar looking distributions across both representations. This makes sense because it just generates a bunch of random options and chooses the optimal one.

5 Karma Karp as Starting Point for the Randomized Algorithms

The implementation of Karma Karp that we have provided returns a residue but does not construct the corresponding partition. We could easily construct it with a slight variation, as suggested in the prompt.

Supposing that we get the partition from the Karma Karp algorithm, we can use that partition as a starting point instead of generating a random one.

The first thing to notice is that the three algorithms always return the best solution found. Using the Karma Karp partition as a starting point ensures that we will never get a worse solution. Starting with Karma Karp will have different effects for the two representations.

Standard vs Prepartition: In the Standard Representation starting with Karma Karp would start our optimization from that point in solution space. We therefore expect our solution to slightly improve from the Karma Karp solution, but only slightly. It would only slightly improve because it would quickly get stuck in local minima. Here, the solution space is not very well defined.

With the Prepartition representation, the optimization from the Karma Karp solution would still reduce the residue significantly. We expect the solution to be lower than the solutions obtained without starting with Karma Karp. Here it is harder to get stuck in local minima because the solution space is defined better.

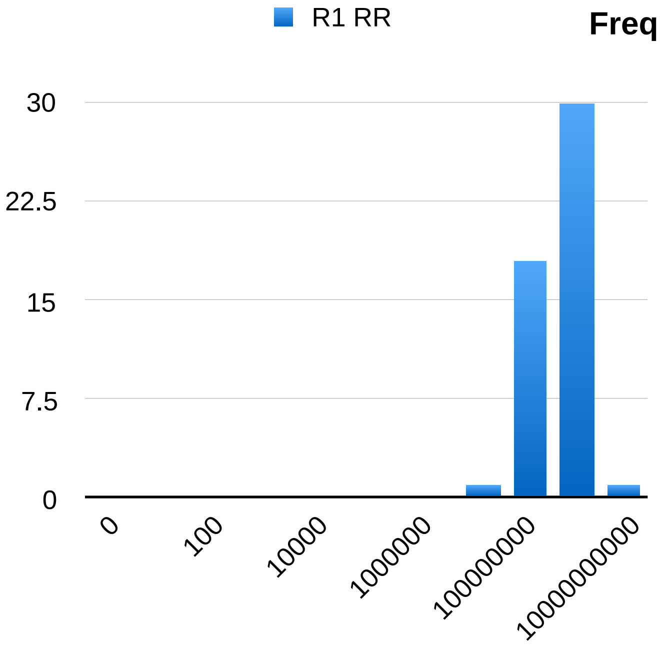
Repeated Random: Starting Repeated Random with Karma Karp has the effect of only adding to the randomly generated solution the Karma Karp solution. As we have seen in the previous graphs, Karma Karp usually offers a significantly better solution than the Repeated Random Algorithm for the standard representation. Therefore, on average, we will get the Karma Karp solution back when using the variation of Repeated Random with the standard representation.

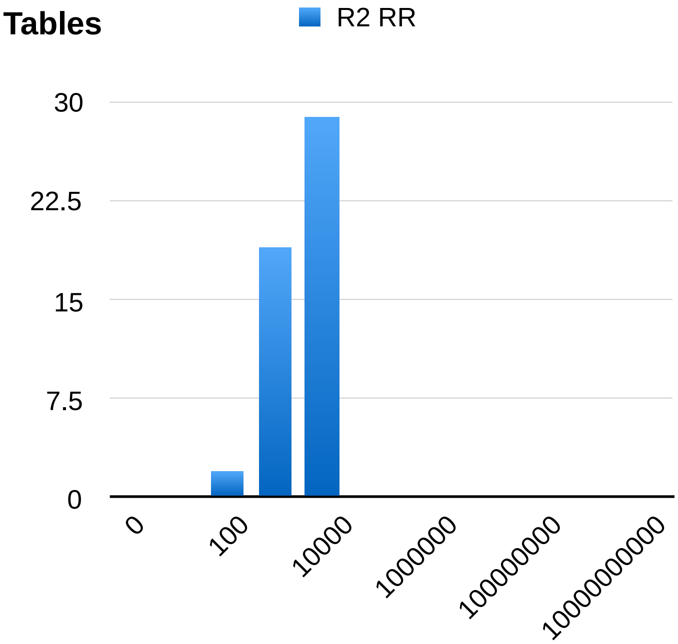
With the prepartition representation instead, Repeated Random usually generates better results than Karma Karp. So adding the Karma Kart solution to the solution set won't affect the result of the algorithm.

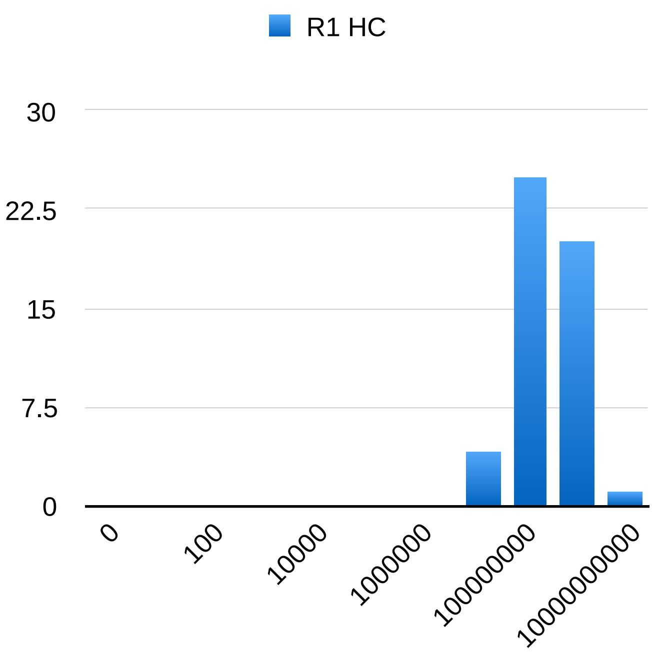
Hill Climbing: When we start Hill Climbing with Karma Karp we are actually starting with an already fairly optimized solution. What will happen is that improvements will be less frequent. This means that the curve of residue in time will be flatter. As we get closer to the optimized value the solution space gets flatter and flatter.

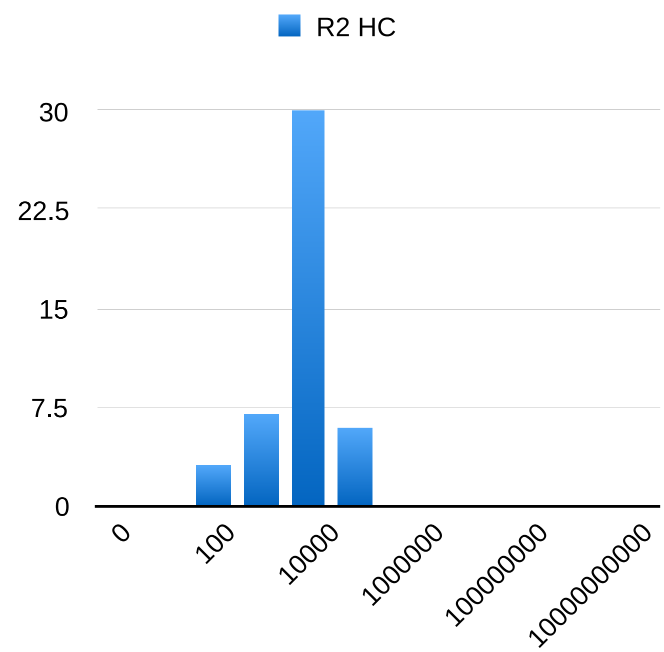
Simulated Annealing: Similarly, starting with the Karma Karp solution corresponds to starting with an already fairly optimized solution. Here as well, the frequency of the improvements will be lower. This means that the curve of residue in time will be flatter.

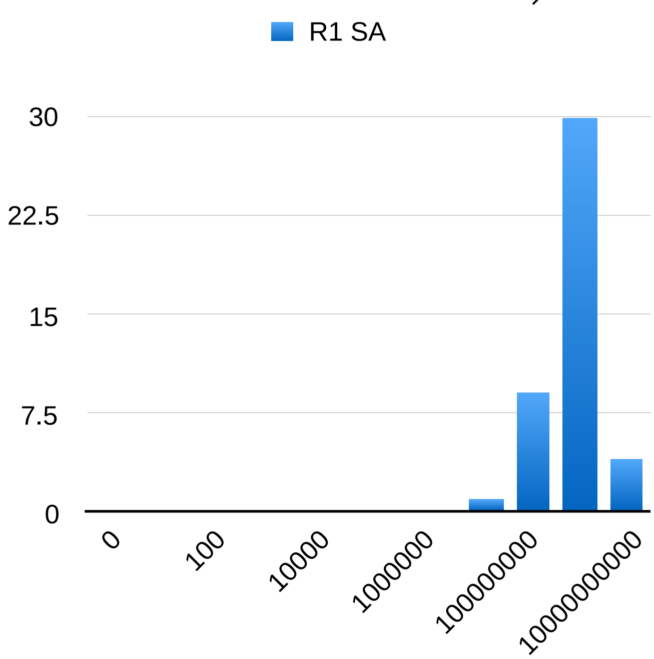
Frequency Tables

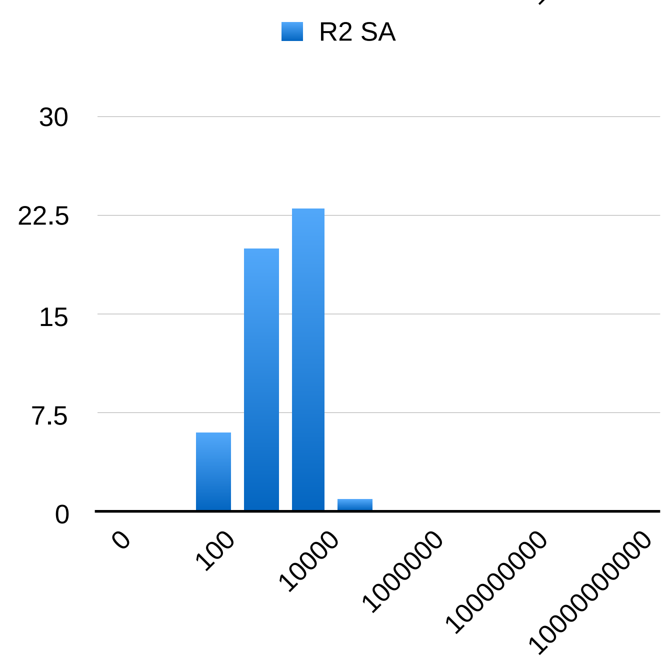


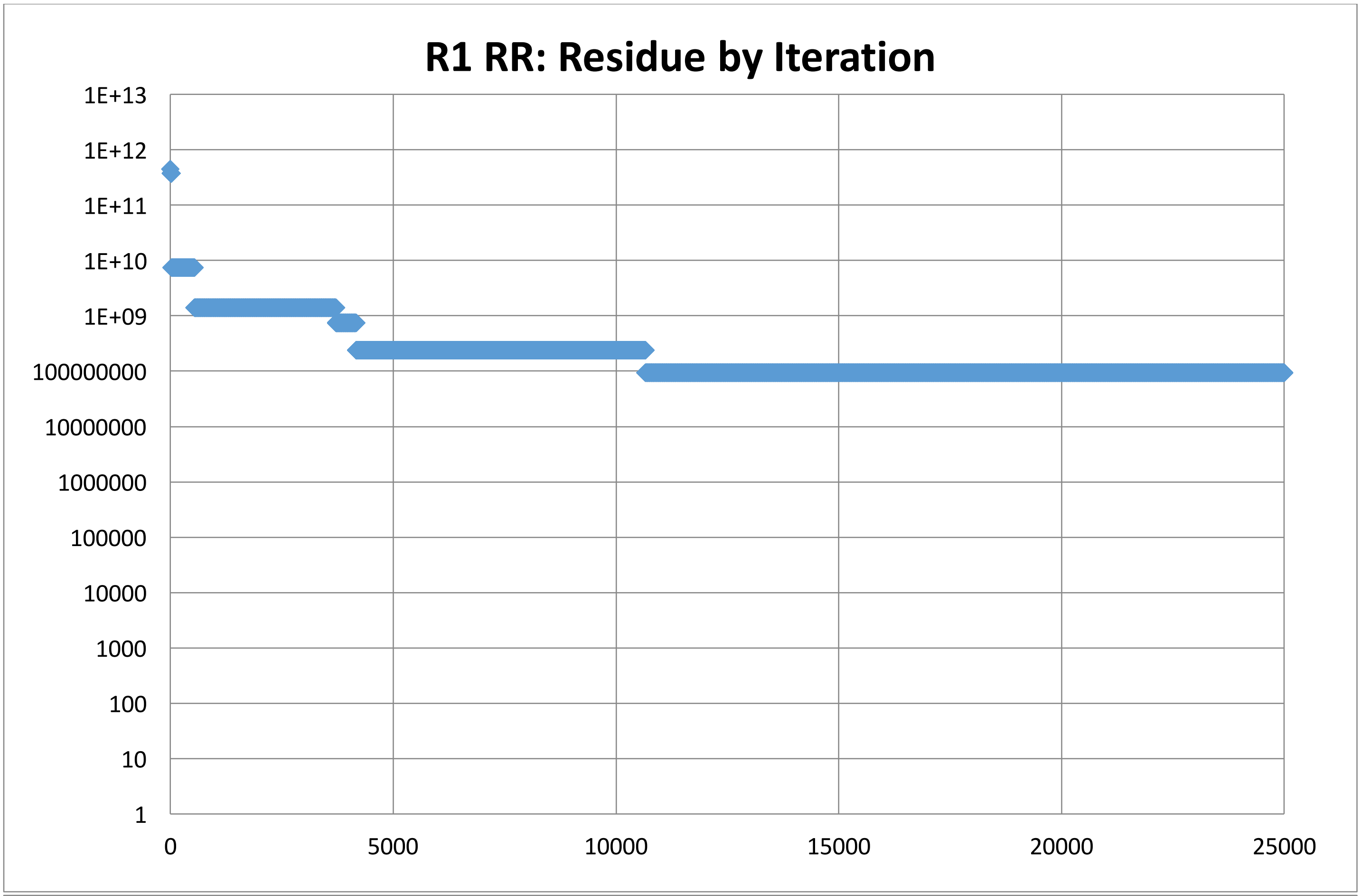


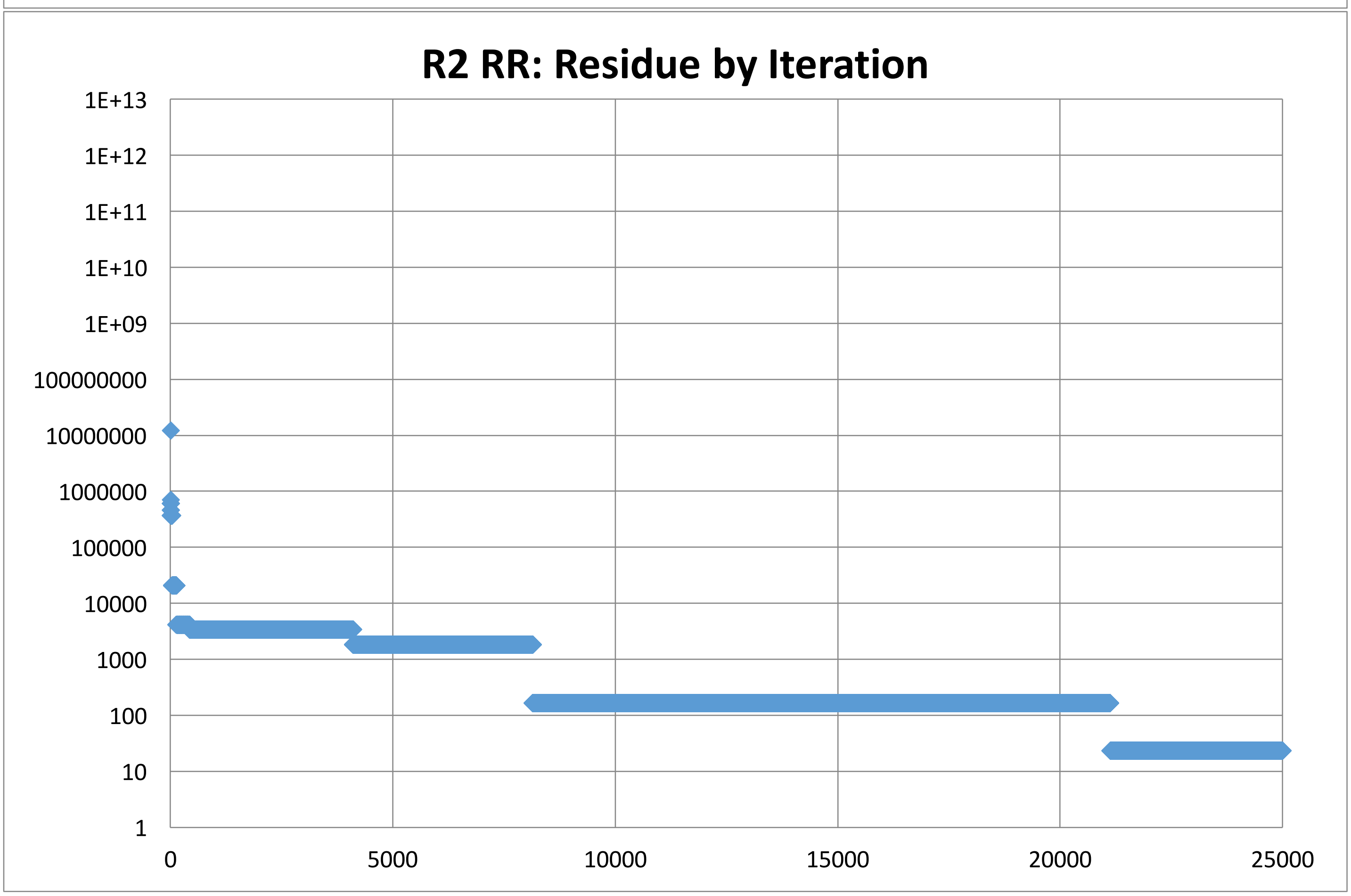


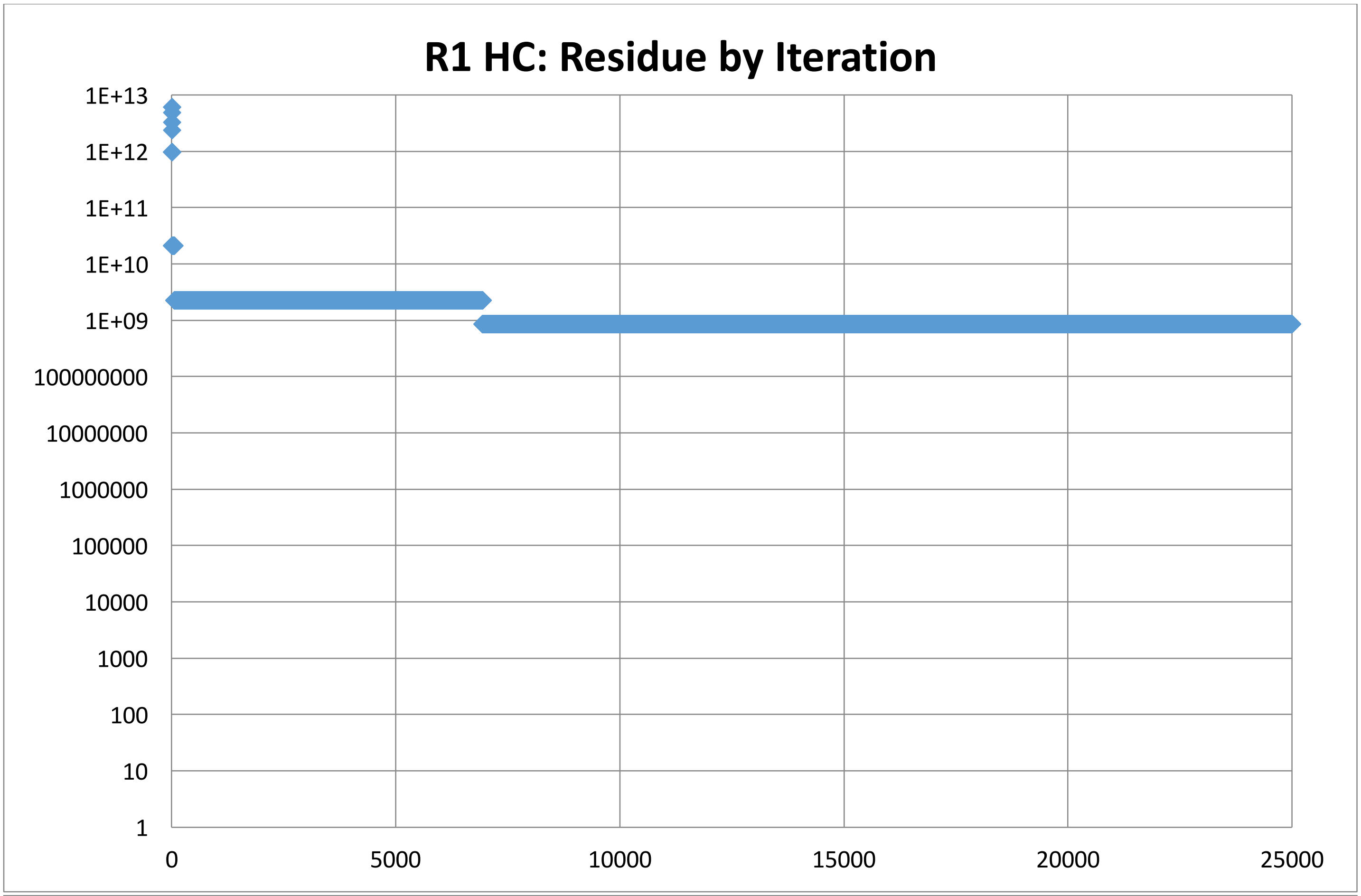


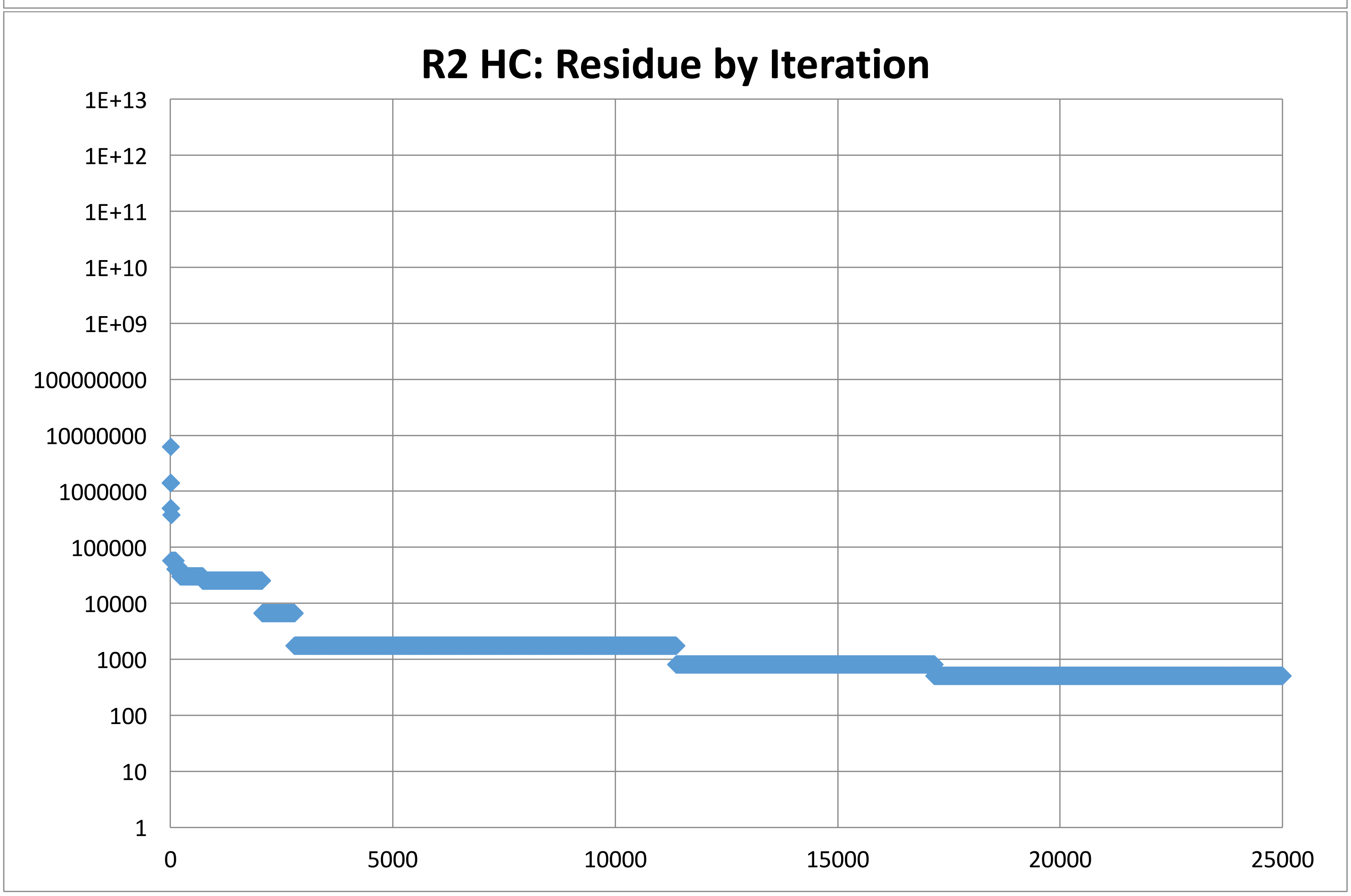


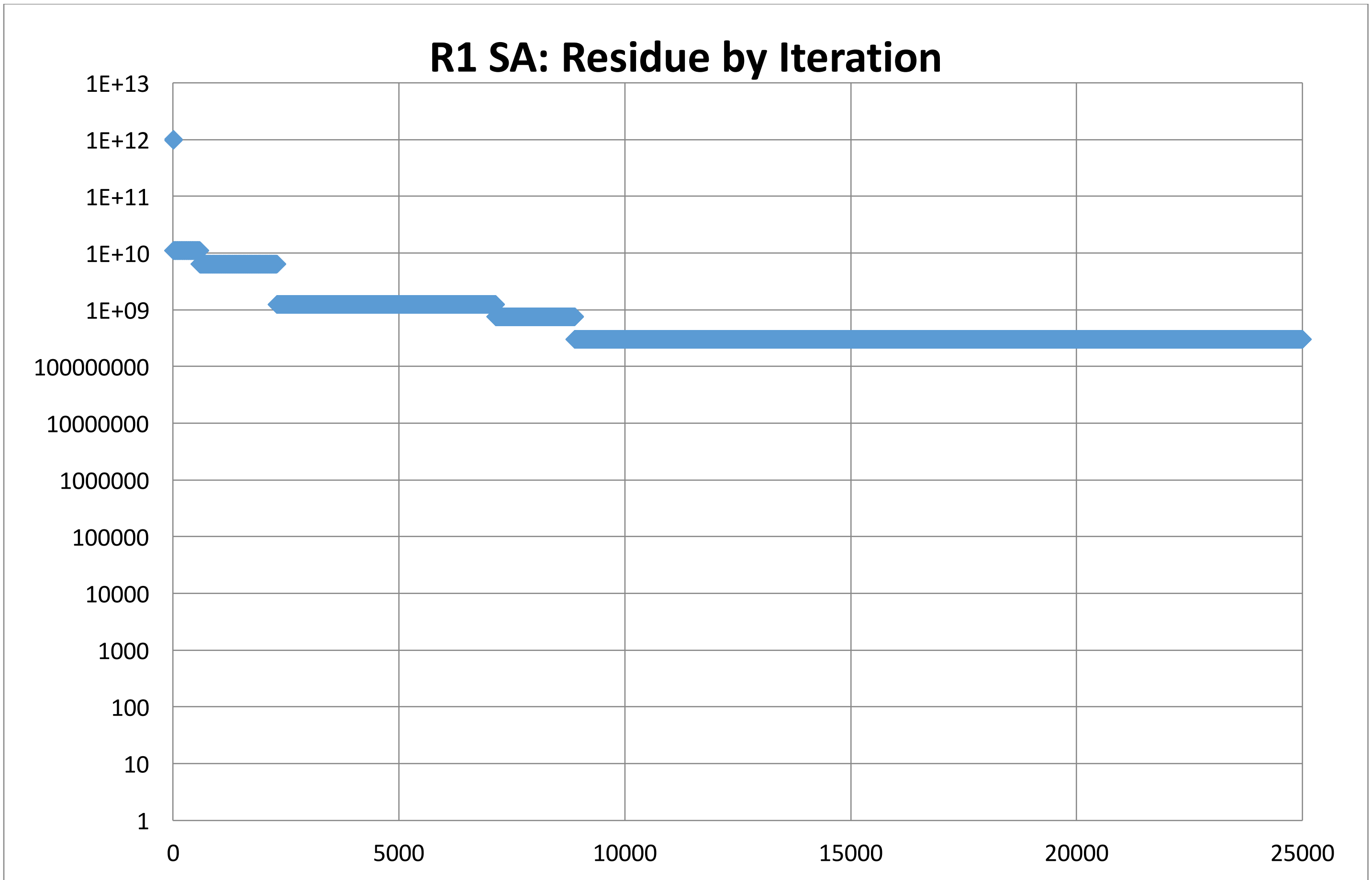


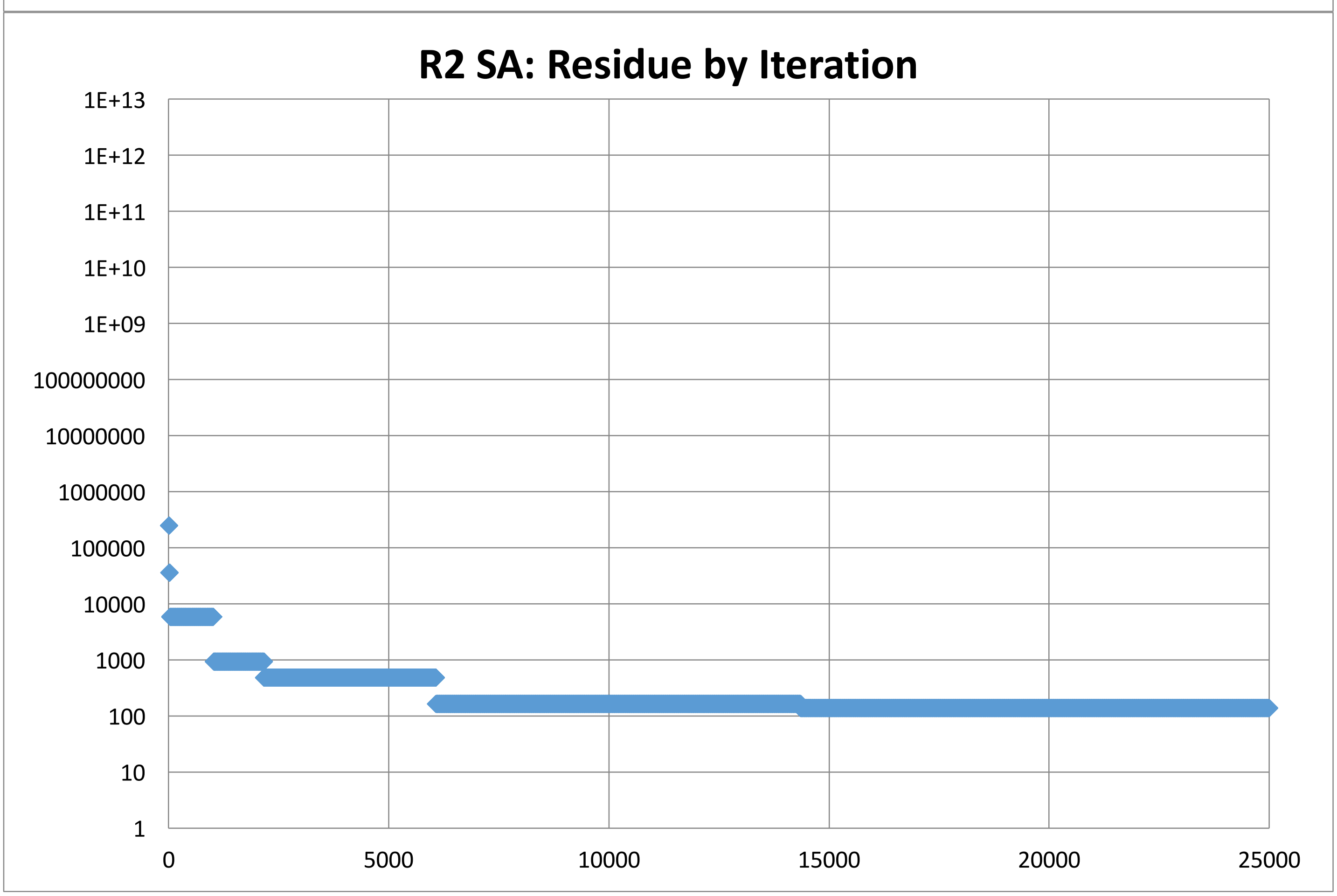

R1 RR: Residue by Iteration
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
5000
10000
15000
20000
25000


R2 RR: Residue by Iteration
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
5000
10000
15000
20000
25000


R1 HC: Residue by Iteration
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
5000
10000
15000
20000
25000


R2 HC: Residue by Iteration
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
5000
10000
15000
20000
25000


R1 SA: Residue by Iteration
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
5000
10000
15000
20000
25000


R2 SA: Residue by Iteration
1E+13
1E+12
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
5000
10000
15000
20000
25000

Karmarkar-Karp algorithm

Answer	Time
229226	0.000708103
194876	0.001968145
16722	0.000598907
555880	0.000597954
83118	0.000598907
786710	0.000596046
464228	0.00062108
501557	0.000594854
84974	0.000607014
426572	0.00058794
250107	0.000597954
191298	0.000627995
1080644	0.000597
92256	0.000602007
64104	0.000603914
75208	0.000590086
71921	0.000593901
71605	0.000625849
533079	0.000596046
60091	0.000592947
87817	0.000596046
129799	0.000601053
225823	0.000595808
169894	0.0006001
12711	0.000597954
10266	0.000597954
1617090	0.00063014
186309	0.000588894
78468	0.000623941
125932	0.000627041
217444	0.000596046
1649	0.000602007
177981	0.000597954
15190	0.000593901
54880	0.000594139
988117	0.000597
136419	0.000602007
107022	0.000604153
148308	0.000603914
38829	0.000599146
269257	0.000594854
212943	0.000602961
766059	0.000598907
192227	0.0006001
277401	0.000607967
28794	0.00059104
56341	0.000603914
15873	0.000593901
1401383	0.000620127
35568	0.000598907

Representation 1

Repeated Random		Hill Climbing		Simulated Annealing	
Answer	Time	Answer	Time	Answer	Time
2816405844	0.069067001	739193708	0.030761003	2561314908	0.032793999
750008290	0.077694893	756993706	0.030636072	4915241778	0.032267094
4168913384	0.062952042	1516968008	0.031197071	10866071594	0.032176971
3109280290	0.06302309	33614508	0.030861139	8317866310	0.032112837
330474608	0.062843084	283867114	0.031086922	4699256156	0.032284021
4455871056	0.062260866	82920162	0.033584118	96468390	0.032284021
3862986690	0.062721014	787869768	0.03095293	3931747384	0.033190966
2490957073	0.062957048	1176789307	0.030821085	6385048391	0.032396078
5166248898	0.069345951	1527025798	0.030838013	644540588	0.032179117
2717778108	0.062526941	4522350532	0.030801058	671081726	0.032156944
3507855067	0.062752008	184263517	0.030723095	2193268243	0.032175779
350366350	0.062951088	1294473584	0.031116962	4510946526	0.032273054
5554424940	0.062793016	704326576	0.030972004	742019826	0.03222394
745176680	0.063199997	1664778784	0.030920029	2160145878	0.032270908
7607459758	0.063374043	370275224	0.030804873	2722662474	0.035181999
1062099490	0.063088894	175066500	0.031221151	2016188738	0.042587996
923192087	0.063127995	140832081	0.030910015	6713055809	0.033161879
377057403	0.062252045	612513153	0.031049967	1033008809	0.032075167
638740827	0.06249094	622381095	0.030632973	1555492513	0.032270908
8605253269	0.06262207	195085617	0.031358004	5584349075	0.032253981
2036905313	0.062551022	16104163	0.030855894	951519755	0.032194138
7166235659	0.062653065	1213590561	0.031196833	4169585267	0.032270908
1086788225	0.062805176	139336229	0.030835867	3829720205	0.032196999
1195766544	0.062861919	166432462	0.030898094	3936781386	0.032315969
171575963	0.062508106	1220831635	0.031982899	11925197875	0.032277822
1216954168	0.062847853	757629792	0.030997992	10168461972	0.032531977
166181748	0.062896967	3425817666	0.031258106	654766152	0.032266855
714646607	0.062875986	292373819	0.031002045	6072438367	0.032259941
1425111896	0.06280303	985818610	0.031172037	4725723668	0.032305956
1283450032	0.063474894	1034611560	0.031086922	1876608032	0.032333136
2229746620	0.062716961	3830881836	0.030919075	10762709114	0.032420874
4960199103	0.062733889	2385234885	0.03083396	3800230715	0.032265186
917073739	0.064403057	283782427	0.030903101	904737409	0.032464981
84103822	0.08921814	4051777454	0.031176805	2185471460	0.038887024
8023592496	0.062592983	623112464	0.032388926	3536507262	0.032261133
1992699473	0.062859058	419453049	0.031406164	3970501363	0.032416821
1448120045	0.062703133	419138337	0.031286001	2556405979	0.032431841
671467848	0.062962055	1846284474	0.031533957	1817775036	0.033226967
120246490	0.062747002	134768134	0.031230211	5600704620	0.032319784
299004929	0.062878847	980468033	0.031026125	7960757495	0.032279015
12560344323	0.06242609	2103036935	0.03104496	2817336815	0.032279015
2368693241	0.065798044	1781729069	0.031286001	1869307043	0.032284021
1487734489	0.073971033	13945033725	0.031401157	207464909	0.035461903
338593111	0.06338501	1353611341	0.031185865	2390589413	0.032287121
1265884259	0.063238144	26609251	0.030866861	208517481	0.03255105
166942668	0.063468933	1749565764	0.03105998	2068279354	0.032441139
577056179	0.063051939	166989097	0.031185865	1364611193	0.032294989
5019168555	0.063286066	183434767	0.031049013	3294384627	0.032234192
6103556203	0.063070059	1168778383	0.031049967	560896039	0.032361031
671494250	0.06299305	4112222156	0.03123498	2820232450	0.035120964

Representation 2

Repeated Random		Hill Climbing		Simulated Annealing	
Answer	Time	Answer	Time	Answer	Time
660	2.313399076	6704	2.280812979	3478	2.220596075
5464	2.306154013	2932	2.307091951	642	2.241355896
116	2.297328949	14406	2.26941514	66	2.217646837
2686	2.281932116	150	2.219672918	444	2.213941097
2432	2.299727917	2108	2.150285959	3218	2.211498022
834	2.286860943	72	2.184720993	2294	2.231050015
5334	2.29982686	3666	2.228904009	434	2.223781109
791	2.307342052	5511	2.273067951	5825	2.235877991
944	2.299906969	9512	2.153842926	846	2.233578205
374	2.291710138	2460	2.239284992	12256	2.275161028
303	2.292513132	10223	2.205576897	2367	2.273689985
4538	2.376430035	6348	2.165063143	308	2.243432045
1632	2.289139986	376	2.309262991	424	2.231279135
480	2.291491032	17690	2.154083014	522	2.225110054
1058	2.326239824	9762	2.196033001	394	2.238559008
4850	2.305636883	334	2.310106039	94	2.217540979
71	2.309085131	5255	2.199488878	6723	2.220463991
3611	2.311796904	1325	2.222308159	1613	2.223433971
811	2.330661058	2897	2.268926144	955	2.227865934
2353	2.322247982	4761	2.219180107	1595	2.235929012
3061	2.316725969	19	2.165024996	511	2.224714994
263	2.295964003	317	2.246432066	35	2.244106054
1221	2.292189121	2811	2.256707907	49	2.226128101
256	2.291589975	5446	2.173305035	524	2.212344885
2959	2.304033995	5165	2.197100878	1107	2.256728172
94	2.300533056	4080	2.210428953	1578	2.211266041
1564	2.316329956	2626	2.144999981	1008	2.222245216
7433	2.309983015	227	2.249521017	21	2.220259905
1428	2.292896986	2364	2.260319948	1184	2.233510971
916	2.289746046	2536	2.221441984	766	2.218199015
232	2.305505037	680	2.382018089	352	2.216674089
1533	2.293994904	4031	2.238296032	509	2.220947027
187	2.314954996	17523	2.232171059	487	2.246906996
858	2.348347902	5850	2.216403961	2418	2.221212864
2532	2.299618959	8624	2.195088863	1222	2.287572861
223	2.30272007	2437	2.227801085	883	2.236484051
1923	2.307859182	2203	2.252671957	3693	2.222406864
1752	2.340453863	2756	2.302246094	3002	2.227596998
3190	2.305228949	62	2.159313917	510	2.223443985
847	2.302381992	1071	2.168496132	3153	2.230854034
1801	2.293009996	13257	2.090248108	47	2.270902872
2999	2.314376831	1999	2.259596109	2347	2.218893051
877	2.301594019	1815	2.20531702	1711	2.232429981
2181	2.301867962	3913	2.222611904	535	2.235788822
771	2.288982153	1247	2.24209404	819	2.23220396
2882	2.318780899	1202	2.216160059	1578	2.222800016
2615	2.296329021	307	2.256944895	1219	2.233275175
1483	2.354251862	5693	2.213942051	2681	2.206131935
1651	2.343328953	11423	2.162466049	1311	2.316601038
6320	2.411679029	2038	2.256746054	632	2.378583908