Programming Assignment 3

Rita Bogdanova-Shapkina

April 23, 2022

Late days used on this pset: 2
Total late days used: 12

1 Dynamic Programming Solution to Number Partition

The Number Partition problem takes as input a set A of size n containing nonnegative integers $\{a_1, a_2, \ldots, a_n\}$ and outputs a set S size n of signs $\{s_1, s_1, \ldots, s_n\}$, $s_i \in \{-1, 1\}$, and an integer ("residue") $u = |\sum_{i=1}^{n} s_i a_i|$ such that the residue u is minimized. Note that assigning the signs s_i to minimize u is equivalent to partitioning A into two sets A_1 and A_2 such that $u = |\sum_{a_i \in A_1} a_i - \sum_{a_j \in A_2}|$ is minimized. In this representation, without loss of generality those a_i that are "assigned" positive sign $s_i = 1$ are placed in A_1, and those assigned negative sign $s_j = -1$ are placed in A_2. Let $b = \sum_{i=1}^{n} a_i$. Then, note that minimizing u is equivalent to choosing A_1 such that $\sum_{a_i \in A_1} a_i$ approaches $\lfloor b/2 \rfloor$ as closely as possible (wlog from below), as $\sum_{a_i \in A_1} a_i = \lfloor b/2 \rfloor - \epsilon \implies \sum_{a_j \in A_2} a_j = \lceil b/2 \rceil + \epsilon \implies u = \lfloor b/2 \rfloor - \lceil b/2 \rceil + 2\epsilon$ (if b is even, $\lfloor b/2 \rfloor - \lceil b/2 \rceil 0$, otherwise it equals 1), so minimizing ϵ, the difference between $\lfloor b/2 \rfloor$ and $\sum_{a_i \in A_1} a_i$, optimizes the residue u. With this, consider the following algorithm:

0. For convenience, reindex the a_is to be 0-indexed
1. Create a table T of size $n \times \lfloor b/2 \rfloor$, such that $T(i, k) = 1$ if there is a subset of $\{a_0, \ldots, a_i\}$ such that the a_j in that subset sum to k, and 0 if otherwise. Also create a corresponding $n \times \lfloor b/2 \rfloor$ table of pointers P such that $P(i, k)$ stores the subset of $\{a_0, \ldots, a_i\}$ that adds to k, if $T(i, k) = 1$.
2. Set $T(i, 0) = 1$ for all i and $T(0, k)$ for all $k \neq 0 \in [\lfloor b/2 \rfloor]$. Correspondingly, set $P(i, 0)$ to the empty set for all i.
3. Set $T(i, k) = 1$ if $T(i-1, k) = 1$ or if $T(i-1, k-a_i) = 1$. Correspondingly, if the first is true, set $P(i, k)$ to $P(i-1, k)$, and if the second is true set $P(i, k)$ to $P(i-1, k-a_i) \cup \{a_i\}$
4. Iterate from $T(n, \lfloor b/2 \rfloor)$ down to $T(n, 0)$ until the first m such that $T(n, \lfloor b/2 \rfloor - m) = 1$ is reached; return the set $A_1 = P(n, \lfloor b/2 \rfloor - m)$ and residue $u = (\lfloor b/2 \rfloor - \lceil b/2 \rceil) + 2m$.

Proof of correctness:

We proceed by induction to show the correctness of steps 1-3. First, consider the base cases $T(0, k)$ and $T(i, 0)$ (and the corresponding P values). When $i = 0$, the only subset of A possible is the empty set of no a_i values, which naturally sums to 0; so, setting $T(0, k)$ to false (0) for all $k \neq 0$ in Step 2 is correct. Meanwhile, when $k = 0$, any subset of A contains the empty set, which naturally sums to 0; so, setting $T(i, 0) = 1$ and correspondingly pointing $P(i, 0)$ to the empty set in Step 2 is correct.

Now, we assume $T(p, q)$ and $P(p, q)$ are correct for $p \leq i-1$, $q \leq k$, and consider $T(i, k)$. If there is a subset S of $\{a_0, \ldots, a_i\}$ that sums to k, then either that subset is also a subset of $\{a_0, \ldots, a_{i-1}\}$ (i.e. a_i is not needed for the sum), or a_i is needed for the sum and so there is some subset of $\{a_0, \ldots, a_{i-1}\}$ that adds to $k - a_i$ (since $x + a_i = k \implies x = k - a_i$). The first case is checked by the $T(i-1, k) = 1$ condition in Step 3, which correctly returns whether there is indeed a subset of $\{a_0, \ldots, a_{i-1}\}$ that sums to k since $T(i-1, k)$ is correct by inductive hypothesis. Then, if the check passes, since $P(i-1, k)$ is correct by the

inductive hypothesis, setting $P(i,k)$ to $P(i-1,k)$ is also correct as the subsets are the same. In the latter case, similarly since $T(i-1,k-a_i)$ is assumed correct, there is s subset of $\{a_0,\ldots,a_{i-1}\}$ that sums to $k-a_i$ if and only if $T(i-1,k-a_i)=1$, so that check is correct; if the check passes, the assumed correctness of $P(i-1,k-a_i)$ shows that setting $P(i,k)=P(i-1,k-a_i)\cup\{a_i\}$ is correct, as this is by definition of "preceding" subset with only the addition of a_i. This proves the correctness of steps 1-3 by induction.

Lastly, it was shown in the paragraph before the algorithm that minimizing the difference between the sum of A_1 and $\lfloor b/2\rfloor$ yields the correct output. Since per the above $T(n,\lfloor b/2\rfloor-m)$ correctly states whether or not some subset of $\{a_1,\ldots,a_n\}$, i.e some subset of A, adds to $\lfloor b/2\rfloor-m$, searching for the first "yes" with respect to decreasing m as done in Step 4 and then returning that set, which is stored in $P(n,\lfloor b/2\rfloor-m)$, yields the correct partition of A, i.e. output of signs S as explained in the first paragraph. Also, returning the residue by the formula $(\lfloor b/2\rfloor-\lceil b/2\rceil)+2m$ is correct as shown in the first paragraph, as m is exactly the difference ϵ between the sum of A_1 (i.e. $P(n,\lfloor b/2\rfloor-m)$) and $\lfloor b/2\rfloor-m$. So, the algorithm is correct.

Proof of time complexity:

In total T and P are both $n\times\lfloor b/2\rfloor$ arrays, and steps 2-3 iterate through each entry $T(i,k)$ exactly once. Further, during each iteration, either constant-time assignment (Step 2), or constant-time assignment, comparisons, and arithmetic are performed (Step 3) - since the entries of P are pointers, assigning the pointers (when setting $P(i,k)=P(i-1,k)$) or copying a pointer and adding an element (setting $P(i,k)=P(i-1,k-a_i)\cup\{a_i\}$) takes constant time if using a data structure like a linked list. So, steps 2 and 3 takes $O(n\times\lfloor b/2\rfloor)=O(bn)$ time, as does Step 1, the creation of two $n\times\lfloor b/2\rfloor$ tables. Step 4 iterates through at most $\lfloor b/2\rfloor$ elements (from $m=0$ to $m=\lfloor b/2\rfloor$), and so takes $O(b)$ time. So, total time complexity is $O(nb)$, which is pseudopolynomial with respect to n (as a true polynomial algorithm would be polynomial with respect to the length of the input in bits, i.e. $\log(n)$).

2 Karmarkar-Karp algorithm

The Karmarkar-Karp algorithm provides a heuristic for Number Partition by repeatedly taking the two largest elements, a_i and a_j, from the input set A and "differencing" them, i.e. replacing the larger with $|a_i-a_j|$ and the smaller with 0. The final remaining nonzero number is returned as the residue, as the above 'differencing' takes advantage of the fact that placing elements in different sets is equivalent, in the final summation, of placing their difference in one of the two sets.

The Karmarkar-Karp algorithm can be implemented in $O(n\log n)$ time as follows: first, insert all n elements of the set into a max heap; second, repeatedly extract the two largest elements, a_i and a_j from the heap and insert $|a_i-a_j|$ and 0 into the heap; third, return the last nonzero element (i.e. when extracting the largest two elements results in one of them being 0).

The insertion step takes$O(n\log n)$ time in total since inserting each element takes at most $O(\log n)$ time. Each extraction and insertion takes $O(\log n)$ time individually, so each extraction step (2 extractions, a subtraction and absolute value, and two insertions) takes $O(\log n)$ time. Since exactly one of the elements in the set is changed to 0 during each step, and until the last step only nonzero elements of the list undergo extractions/insertions (since if one of the two largest elements is 0, the algorithm is complete), so a total of $\lfloor n/2\rfloor$ insertion/extraction steps are performed, resulting in $O(n\log n)$ time. So, $O(n\log n)$ time is needed for the Karmarkar-Karp algorithm.

3 Implementation

C++ was used to take advantage of its speed for testing.

To implement the Karmarkar-Karp algorithm, first, max heaps were implemented almost identically to my implementation of min heaps on PA 1 (which were used for Prim's algorithm), essentially only modifying the checks for the smaller element to checks for the larger element and changing the datatype from `double`

and `int` to C++'s native `long long int`, which can take on 64-bit integer values; `long long int` was also used as the datatype for all variables dealing with the actual integer contents of the Number Partition instances. In the code, max heaps were implemented by creating a `MaxHeap` class with the heap itself built on a `std::vector` and the `insert` and `extractMax` functions being public class functions implemented with private functions `maxHeapify`, `getParent`, etc.

To implement the randomized partition algorithms, the helper functions `randomSolution`, `genNeighbor`, and `solResidue` were fist implemented, which respectively generate a random Number Partition solution, generate a neighbor of a Number Partition solution per the specifications, and return the residue of a given Number Partition solution. Since two solution formats - Standard and Prepartition - were examined, each function included a `part` flag that indicates whether the function should proceed win the context of the Standard or the Prepartition solution format. This flag is inherited from the original call of the algorithms for Repeated Random, Hill Climbing, and Simulated Annealing.

The Standard version of `randomSolution` outputs a list (`vector`) of length n such that each entry is a 1 or -1, with an equal probability of 1/2 (done via an if statement depending on a `rand()/INT_MAX` call that generates a number from 0 to 1; a given index in the solution is set to -1 if below .5, and otherwise to 1). The Prepartition version generates a list of size n such that each entry is randomly chosen from $[0, n-1]$ (i.e. the indexes of input A), similarly via a call to `rand() % n`.

The Standard version of `genNeighbor` first copies the input (Standard solution), then generates a first index i in $[n]$ (via `rand() % n` as above) and sets the sign at index i to its negative in the neighbor list. Then, a check for probability 1/2 is done (identically to the `rand()`-generated check done to generate Standard solutions, as above), and if it passes a second index $j \neq$ `sol`$[i]$ (which is assured by putting the generation of j in a `while` $i == j$ loop) is generated and the sign at index j is set to its negative in the neighbor list.

The Standard version of `solResidue` takes as input a Standard solution `sol` and the original input list A, and simply returns the sum of the values at each index of A multiplied by their respective "signs," i.e. the value at the same index of `sol`, as per the original definition of residue. In the Prepartition version, the input Prepartition is iterated over, and the ith index of a new array is set to the sum of all numbers at the input indexes where the integer at that same index in the partition solution equals i.

Then, the randomized algorithms themselves were defined in accordance with the pseudocode given in the assignment. Repeated Random was implemented in a for loop (up to `max_iter`) where in each iteration, a random solution was generated (via `randomSolution`) its residue was calculated (via `solResidue`) and compared to that of the current smallest residue (initially set to the residue of the first random solution), replacing it if smaller. Hill Climb was similarly implemented in a for loop where, after an initial random solution was generated, in each iteration a neighbor is generated and replaced the current solution and current residue with itself and its residue (if smaller) if its residue is smaller than the residue of the current solution. Simulated Annealing was implemented by storing the current and best solution (both initially set to a randomly generated solution), and during each for loop iteration, a random neighbor replaced the current solution if it had smaller residue or with given probability $e^{-(res(S')-res(S))/T(i)}$, and then the current solution replaced the best if it has smaller residue. The probability cooling function for Simulated Annealing was $T(i) = 10^{10}(.8)^{i/300}$, for $i \in$ [`max_iter`], which was chosen to be 25000. All three randomized algorithm function accepted a `part` boolean flag that they passed to the helper functions `randomSolution`, `genNeighbor`, and `solResidue` to indicate whether the Standard or Prepartition version of each should be used - changing this flag is how the Prepartition algorithms were called.

4 Results & Discussion: Residues

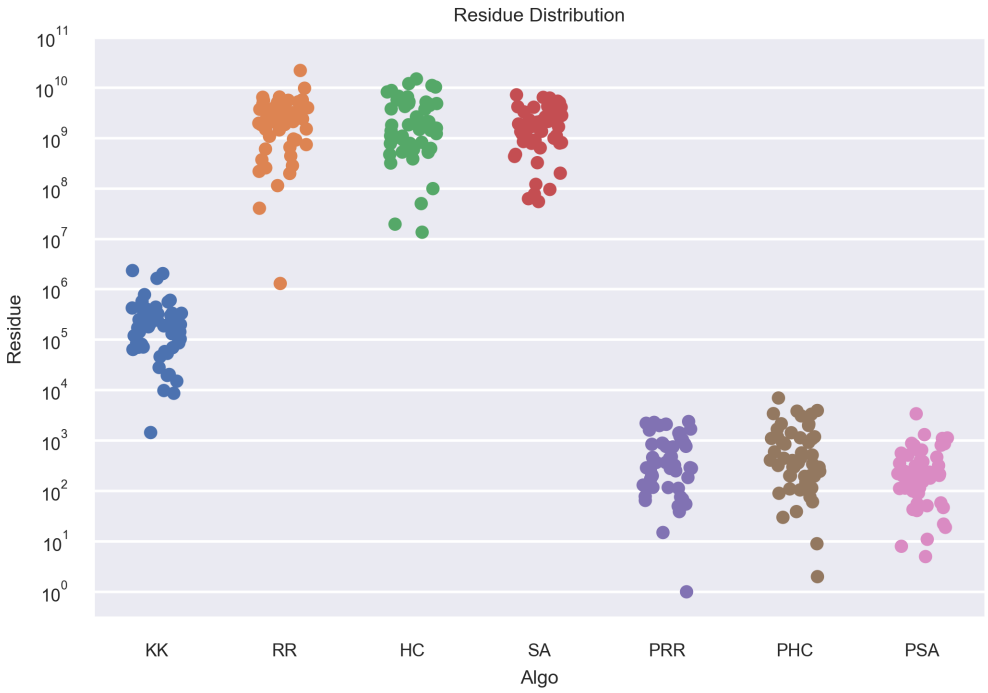
To test the 7 algorithms, a new Number Partition instance of size 100, which in the implementation is a `std::vector` of size 100, was generated 100 times (simply by setting each value in the vector to a random `long long int` in the range $[1, 10^{12}]$, and the residue yielded by each of the algorithms was calculated for each instance, as well as the algorithms' runtime. The randomized algorithms were run with `max_iter`, i.e. the number of randomized iterations, set to 25000 (with the exception of the iteration testing - see below).

Below is a table of the average residue of size-100 Number Partition problems with integers in $[1, 10^{12}]$ returned by the 7 algorithms; here and throughout, KK indicates the Karmander-Karp algorithm, RR the Repeated Random algorithm, HC the Hill Climb algorithm, and SA the Simulated Annealing algorithm; the prefix P indicates the Prepartition version of the randomized algorithms. Residues are rounded to the nearest whole number.

Average Residue, by algorithm

KK	RR	HC	SA	PRR	PHC	PSA
316721	3108073340	3178815436	2227916398	708	1032	392

In addition, the following is a graph of all 50 residues obtained from the tests of each algorithm (note that the y axis is on a log scale):



These results indicate show that using the Prepartition solution format causes the randomized algorithms to significantly (approximately by a factor of 10^6 for all three randomized algorithms) outperform the same algorithms using the Standard solution format. A potential contributing factor to this is the greater 'randomness' offered by the Prepartition format when calculating residue: the Standard format solution does the equivalent of immediately assigning each number to one of the two ("final", with respect to Number Partition) sets, the +1 indexes being one set and the -1 indexes being the other. Therefore, calculating neighbors is only ever equivalent to switching (at most two) elements between sets, and in general residue is "fixed" to the current set division. In contrast, the Prepartitioning format, by giving as input to the Karmander-Karp algorithm a set obtained by adding all elements in the original input that are randomly assigned the same integer in $[n]$, effectively divides the input into up to n sets (however many distinct integers the Prepartition solution randomly generated), allowing greater flexibility (and the same with neighbor generation). Since the Prepartitioning format uses the KK algorithm to calculate residue this 'division into up to n sets' serves to lower residue as the KK algorithm then divides up to n 'randomly shuffled' sets into the final two, increasing result variety; the fact that the KK algorithm on a given input also returns much

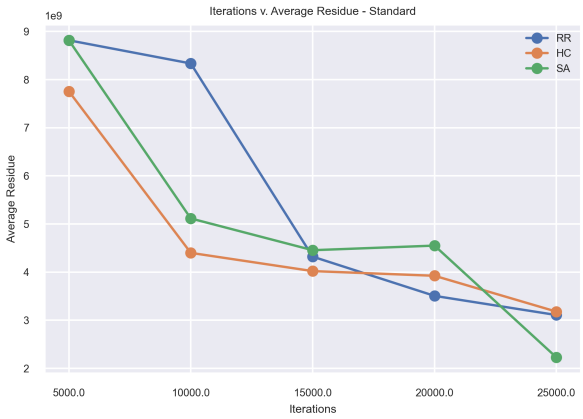
lower residue (approximately by a factor of 10^3) estimates than the Standard randomized algorithm further contributes to performance.

Within the Standard randomized algorithms, the Repeated Random and Hill Climbing algorithms had similar performance (HC's being slighly worse), with Simulated Annealing beating them out by approximately 33% in the average, though having a smaller range of residues than the other two. This is likely because the Simulated Annealing algorithm has the benefit of the Hill Climb algorithm over Repeated Random in that it gradually moves towards a minimum, "saving" previous changes, instead of potentially choosing wildly different solutions and "losing" previous progress (which is exactly done by Repeated Random), yet avoiding HC's pitfall of becoming potentially stuck at a solution that is a "local minimum" in that it is more optimal than (most of) its neighbors, but neighbors' neighbors (or further 'steps' away) would be more optimal, as HC never moves away from such a local minimum but SA transitions to even larger-residue neighbors in accordance with the probability function $e^{-(res(S')-res(S))/T(i)}$, $T(i) = 10^{10}(.8)^{i/300}$ where i is the iteration. Note also that the SA residues have less low outliers than the RR residues, likely because finding totally new random solutions has a chance of resulting in a highly optimal one unreachable by taking neighbors.

The same patterns of (average) residues given by HC being slightly larger than those given by RR, which themselves are larger than those given by SA, also hold for the Prepartition format, with the addition that the small scale makes differences more pronounced. Indeed, the change to Prepartition format does not change the algorithms' overall treatment of solutions, so the same facts about their processes (HC but not SA getting stuck, etc.) apply here. The same can be said of the continued presence of low outliers in the results of the RR algorithm.

Lastly, note that while Karmander-Karp generally outperforms the Standard random algorithms, it is soundly beaten by all three Prepartition random algorithms.

Besides the 50 tests at 25,000 iterations from each algorithm, each randomized algorithm's response to changes in the number of iterations done was also tested by running a tests of each on (the same, for all algorithms) 50 randomly generated size-100 Number Partition instances for each value of `max_iter` in $\{5000, 10000, 15000, 20000\}$ in addition to the previous tests at 25000 iterations. The changes in average residue with respect to number of iterations are plotted below:



Note that the scale on the Prepartition graph (right) is logarithmic.

Quite naturally, the residue yielded by the randomized algorithms decreased as the number of iterations increased, as more iterations allow more opportunities to generate a better randomized solution / neighbor. Note also that for both the Standard and Prepartition formats, the impact of increasing iterations had the largest impact on the output of the Simulated Annealing algorithm (especially when usin the Prepartition format). This is likely explained by the SA probability function $e^{-(res(S')-res(S))/T(i)}$, $T(i) = 10^{10}(.8)^{i/300}$ where i is the iteration, that decides whether SA will 'try' less a less optimal neighbor for following iterations. Note that since if this check is performed $res(S') > res(S) \implies -(res(S') - res(S)) < 0$, and T decays

over the iterations, this likelihood of SA switching to a suboptimal neighbor approxmately decreases over the iterations. Therefore, when less iterations are available, this probability falls to ~ 0 more rapidly, and SA more rapidly loses access to its main optimizing feature and becomes akin to a less efficient version of Hill Climb.

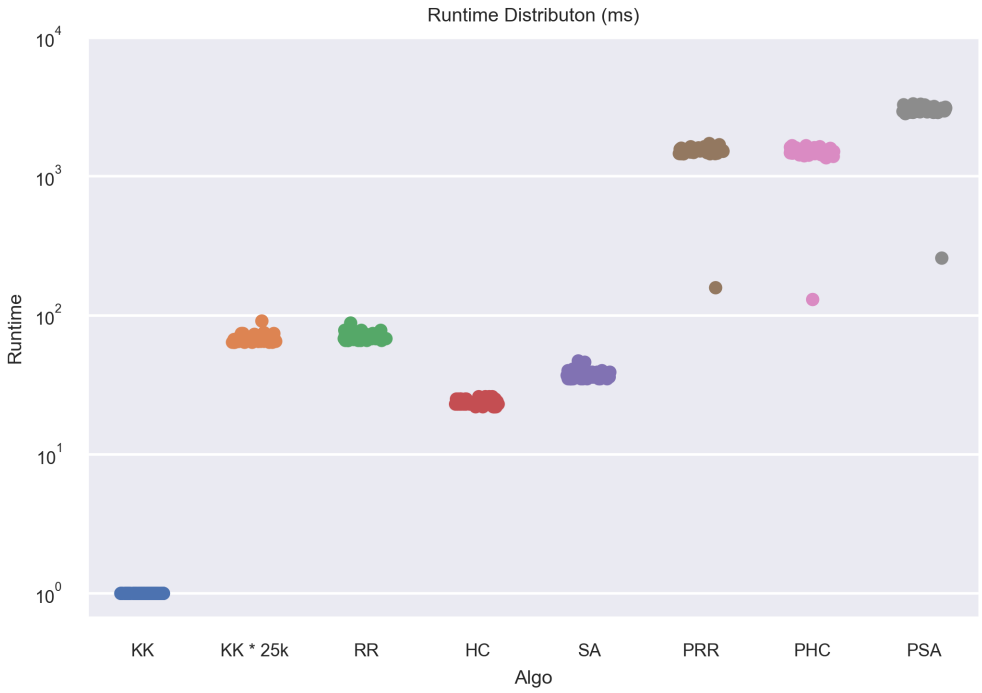
5 Results & Discussion: Runtimes

During the testing, the average and individual runtimes of the 50 trials on random size-100 Number Partition instances were noted for each algorithm. Below is a chart of the average times (KK x 25k indicates running KK 25,000 times, to more directly compare with the iterations of the randomized algorithms):

Average Runtime, by algorithm (milliseconds)

KK	KK x 25k	RR	HC	SA	PRR	PHC	PSA
0	66	70	24	37	1517	1476	2994

And below is a graph of all runtimes for all trials:



Clearly, the Standard format randomized algorithms significantly outperform the Partition format random algorithms in terms of runtime, by a factor fo approximately 10. Asymptotically (assuming $O(1)$ arithmetic), the Standard algorithms take $O(n \cdot \texttt{max\_iter})$ time, as copying neighbors, generating solutions, and summing / multiplying the sets to find residues takes $O(n)$ time. Since the Partition algorithms call on KK to find residue, their asymptotic runtime becomes $O(n \log n \cdot \texttt{max\_iter})$ time. However, since running KK 25000 times is still over 10 times faster than each of the Prepartion algorithms (and KK is certainly not called by them over 10 times per iteration), the other ostensibly $O(n)$ steps needed to calculate residue and generate

solutions in the Prepartition format must also contribute; a likely major contributor is the additions needed to create the input to KK, which are likely not $O(1)$ in reality since the integers are large. Therefore, choosing the Parititon format is a tradeoff of taking lower residues in exchange for worse time performance.

6 Discussion: Improving Randomized Algorithms

If the version of the Karmander-Karp algorithm that returns not only the residue, but the actual set division it approximated, is used, then the neighbor-based randomized algorithms (HC and SA) would likely be improved, as now instead of starting at a random solution (which may have very large residue) and moving away from it in a sequence of neighbors, with an already approximately "good" input, HC and SA would begin searching at a solution guaranteed to be somewhat close to optimal, and therefore have a higher probability of finding an even more optimal result. Further, as shown by the residue/trial distribution graph in the first Results section, (especially if the Prepartition format is used) there is certainly "room" for optimization of the Karmander-Karp solution by the randomized algorithms, which can outperform it even with totally random start points.