Project Proposal

Team: Aashutosh S. Sharma

October 13, 2025

Premise

This project intends to study and compare randomized algorithms with respect to their deterministic counterparts. Often, introducing an element of randomness into an algorithm can yield on average better results than deterministically exhausiting all possibilities. Of course, there are downsides to using such algorithms, which manifest mainly in two different ways, which result in a classification of randomized algorithms into two *named* categories.

Randomized algorithms can be Las Vegas algorithms (those which are guaranteed to yield the correct answer but in a significantly longer amount time in the worst case) or Monte Carlo Algorithms (those which always terminate in no more than a certain amount of time, but have a chance of failing). While runtime is used to compare and analyse Las Vegas algorithms, error rate is used to compare Monte Carlo algorithms, since their likelihood of being correct is dependent on how long (as in, for how many iterations) they are run. This project shall analyse algorithms of both kinds.

Choice of algorithms

Bogosort

In all honesty, this one's more of a meme. This is an oft-cited example as one of the worst randomized algorithms (indeed one of the worst algorithms of any kind). It attempts to sort a given array by generating a random permutation and checking whether it's sorted, and repeating if not.

This is an example of a Las Vegas algorithm, and hence has a worst-case complexity of never terminating (it keeps generating incorrect permutations). On average (assuming distinct elements), the expected runtime grows in $\mathrm{O}(n \cdot n!)$.

Randomized QuickSort

This algorithm attempts to sort an array by randomly picking a pivot and partitioning the array into two subarrays with elements greater than and less than the pivot, then recursively sorting those subarrays.

This is an example of a Las Vegas algorithm, where even though it definitely terminates, in the worst case it is actively worse than most naïve algorithms. However, it still delivers in the average runtime, which is $O(n \log n)$.

Miller-Rabin Primality Test

This algorithm attempts to improve on the $O(\sqrt{n})$ algorithm to test whether a given number is prime. It does so by leveraging a property known to hold for prime inputs but also for *some* composite inputs, similar to the Fermat Primality Test (which uses the fact that $a^{p-1} \equiv 1 \mod p$).

This is an example of a Monte Carlo algorithm, where each iteration of this algorithm makes it more likely to be correct. This algorithm never fails for prime inputs, but has an error rate of 2^{-2k} when run for k iterations.

Karger's Min Cut Algorithm

This algorithm attempts to compute the *Min Cut* of a simple undirected graph. It operates in a way quite similar to Kruskal's algorithm, particularly when it is used in the context of clustering, where vertices are repeatedly "grouped" by randonly choosing an edge and merging its vertices until only two vertices (*i.e.* vertex "groups") remain and the remaining edges make up the min cut. This works since it is far more likely for edges that are not part of the min cut to be selected when sampled uniformly.

This is another example of a Monte Carlo algorithm, since multiple runs make it more likely to hit upon the min cut. For a graph with n vertices, this algorithm generates the correct min cut with probability $\binom{n}{2}^{-1}$, which is significantly better than randomly choosing cuts.

Randomized Incremental Construction for Convex Hulls

This algorithm attempts to generalize methods to construct the Convex Hull of a set of points in multidimensional space (usually 3D) to similar methods to construct Delaunay Triangulations (and therefore Voronoi Diagrams). It does so by randomly choosing points in the set and updating the best-so-far convex hull in a greedy style.

This is a Monte Carlo algorithm which runs in $O(n \log n)$ time which is on par with the best deterministic algorithms like QuickHull and Graham scans. In fact, certain linear programming methods can optimize this to linear time in n, but with a heavier constant factor through the parameters chosen. Interestingly, what we understand of this algorithm allows a nil error rate, since it is effectively a greedy algorithm with random sampling.

Proof and verification

As part of this project, we shall implement the above algorithms, as well as provide their proofs and runtime and error analyses. Furthermore, we shall verify the theoretical results empirically by repeatedly testing these algorithms on procedurally generated inputs. The empirical verification shall be done sufficiently many times in order to obtain a reasonably robust estimate of the average runtime and/or expected error.