, , pp. 1–4

doi:
20:1:2025
Rsearch Paper

RSEARCH PAPER

An Overview of Randomized Algorithms: Applications

Huzaifa Tariq and Iqra Batool

Abstract

Randomized algorithms leverage randomization to improve efficiency and provide near-optimal solutions in diverse problem domains. In contrast to deterministic algorithms, randomized approaches introduce probabilistic behavior, enabling faster execution or approximate results in scenarios where deterministic methods are computationally expensive. This paper discusses the classification, historical development, and practical applications of randomized algorithms.

Key words: randomized algorithms, Monte Carlo methods, Las Vegas algorithms, probabilistic ,

Introduction

Randomized algorithms are computational methods that incorporate random decisions during execution to achieve efficiency and simplicity. They were initially developed for problems in computational number theory and have since found applications in various fields such as cryptography, data analysis, and optimization. This paper provides an in-depth exploration of the types, history, and design principles of randomized algorithms, as well as their practical relevance in addressing complex computational challenges.

Background and Key Contributions

Randomized algorithms trace their origins to Monte Carlo methods, first introduced in solving problems related to numerical analysis, statistical physics, and simulations. [**?**]

Pioneering Work

- **Michael Rabin (1963)** and **John Gill (1977)** built on this idea to explore how randomness could be used in solving problems efficiently.
- The **first practical randomized algorithms** were developed in the 1970s:
  - **Berlekamp (1970)**: Created a randomized algorithm for factoring polynomials.
  - **Rabin (1976)**: Proposed randomization as a general tool, applying it to problems in **geometry** and **number theory**.
  - **Solovay and Strassen (1977)**: Designed a randomized algorithm for **primality testing**.

Development and Research

After these early successes, researchers developed many techniques to create and analyze randomized algorithms. Notable surveys and research include:

- **Karp (1991)**: Focused on various methods for designing randomized algorithms.
- **Maffioli (1985)** and **Welsh (1983)**: Reviewed key developments in the field.

graphicx

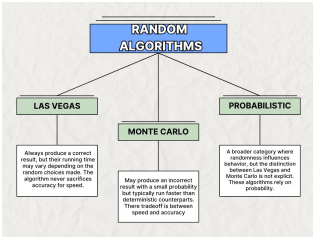

Fig. 1. Randomized algorithm

.

Classification of Randomized Algorithms

Randomized algorithms are typically categorized into three main types based on their behavior and use of randomness:

[4]

Las Vegas Algorithms

- Always produce a correct result, with variability in execution time.

- Example: Randomized Quicksort.
- Use Cases: Data sorting, where accuracy is prioritized. [**?**]

```
function randomizedQuicksort(arr, low, high):
if (low < high):
 // Randomly select pivot
 pivotIndex = random(low, high)
 // Place pivot at the end
swap(arr[pivotIndex], arr[high])
 pivot = arr[high], i = low
  for (j = low to high - 1):
if (arr[j] <= pivot):
swap(arr[i], arr[j]), i++
 // Final pivot placement
swap(arr[i], arr[high])
 // Sort left part
randomizedQuicksort(arr, low, i - 1)
// Sort right part
randomizedQuicksort(arr, i + 1, high)
```

Listing 1. Randomized Quicksort Algorithm

Monte Carlo Algorithms

- May produce incorrect results with a small probability.
- Example: Miller-Rabin primality test.
- Use Cases: Cryptography and simulations.

Probabilistic Algorithms

- Use probability distributions to guide execution.
- Example: Simulated Annealing.
- Use Cases: Optimization and machine learning.

Application of Randomized Alogrithm

. Key strategies include:

Foiling an Adversary

Now let's consider a two-player game in which one is the algorithm attempting to solve a specific problem, and the other is an adversary who may try to cheat or confuse the algorithm by giving it wrong or misleading input data. The goal of the adversary is to make the algorithm's execution time as long as possible. Thus, the adversary's "payoff" is how long it takes for the algorithm to finish its process, which is really just the algorithm's running time on the inputted data. For a randomized algorithm, this means that the process of decision is stochastic in nature. Thus, rather than following a predictable sequence of operations that would make it predictable, the algorithm acts in multiple random ways, much like a player who uses many different strategies. This natural indeterminism makes it more difficult for an adversary to know what to expect from the algorithm and how to input data to prevent its optimal functioning. In short, randomness in the algorithm forces the adversary to have a much tougher time fooling it.

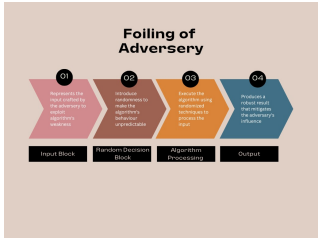

Fig. 2. foiling an adversary

Random Sampling

Random sampling is a method of picking a group from a larger set of things where each item has an equal chance of being picked. It is like picking names out of a hat or drawing cards randomly.

How it work

For example, for a setting that contains 100 students, in which 10 of the students are going to be randomly sampled to participate in a questionnaire, it is assured that via random sampling, every student is equally probable to be picked. Random sampling is conducted based on a principle that ensures an equal chance and lack of bias in the selection from a population where each sampled item or person has a fair chance to be selected. Sometimes, to solve a problem, you need to find a "witness" or "certificate" that proves something is true. For example, to demonstrate that a number is not prime, you need to find some other number which divides it exactly. In some cases, the search effort to find such a witness has to go through a large set of possibilities, which costs too much in terms of time. However, when there is a large population of potential witnesses in the domain of search, the random choice of one provides a good enough chance of selecting a valid witness. The success probability of this search increases as the number of repetitions of process increases

Why use randomness? Randomized algorithms improve performance by reducing the amount of data that needs to be processed, but still provide strong evidence about the identity of two objects.

l

Random Sampling Algorithm

Below is an implementation of the random sampling algorithm:

```
function randomSampling(population[], sampleSize):
sample = [] // Initialize an empty sample
while (size(sample) < sampleSize):
// Select a random index
index = random(0, size(population) - 1)
// Ensure no duplicates
if (population[index] not in sample):
sample.append(population[index]) // Add to sample
return sample
```

Listing 2. Random Sampling Algorithm

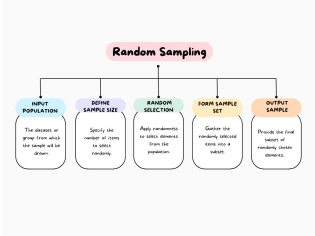

Fig. 3. Random sampling

Fingerprintings and Hashing

fingerprinting uses randomness to create fingerprints of data in a way that reduces the likelihood of two different objects having the same fingerprint

How it works

In a randomized fingerprinting scheme, you may randomly select parameters or use random sampling techniques when creating the fingerprint. For instance, instead of verifying each individual bit or byte in the computation of the fingerprint of a large file, a probabilistic algorithm may verify only random pieces of the data. Hence it can do a quite good job at determining whether two data objects are equal without using very much time. The inherent randomness makes it rather unlikely that any two distinct objects will have the same fingerprint, even when scanning a small part of the data. A popular example is the **Randomized Rabin Fingerprinting** used in pattern matching. It uses a random hash function to create a fingerprint of a string. Since the hash function uses random bits, it helps reduce collisions and increases the chance that the fingerprints of two different strings will not match.

Randomized Algorithms in Hashing:

Randomized hashing enables the construction of hash functions that are faster, more efficient, and better at reducing collisions, which in turn improves the reliability of the hashing process.

Working principles

Universal hashing is a method involving randomness for enhancing the efficiency of hashing. In universal hashing, the hash function is randomly selected from an available set of hash functions. The randomness makes sure the hash function spreads out the data uniformly across the hash table to create less possibility of hash collision. Use of a random selection of hash functions for every operation will make a randomized algorithm capable enough to promote more uniform distribution in the hash table for data. Thus, it avoids clustering or collision, which can deteriorate the performance of the hash table. For example, suppose you use a hash table to store data, and by coincidence two keys hash to the same location, or collision. In this case, the randomized choice of hash functions can minimize this possibility and allow faster lookups. Consistent hashing is just another example where randomness enables distribution of data to be made very efficient across a distributed system, such as when distributing load across servers or nodes on the network. Randomized hashing ensures that the loads are balanced even when nodes are added or removed.

Probabilistic Methods

The probabilistic method is one of the important methods used within combinatorics and theoretical computer science to prove the existence of certain objects that satisfy given properties, even when the actual construction of such objects turns out to be hard or impossible.

How It Works:

The probabilistic method's basic idea is extremely simple: rather than construct an object having some specified property, we prove that if we pick a random object from a set that is sufficiently large, then there is some positive probability that the selected object has the desired property. If that probability is nonzero, then at least one such object must exist with the property that is being imposed. This technique is particularly helpful in showing the existence of an object, but it may lack the ability to clearly identify or define it. The randomness feature is essential to prove that the object under study is likely to exist because of chance.

Randomized Hash Algorithm for Fingerprint and Hashing

Below is an implementation of the randomized hash algorithm used in fingerprinting and hashing:

```
function randomizedHash(input, hashFunctions[]):

// Randomly select a hash function
hashFunction = randomChoice(hashFunctions)

// Apply the selected hash function to input
hashValue = hashFunction(input)
    return hashValue
```

Listing 3. Randomized Hash Algorithm

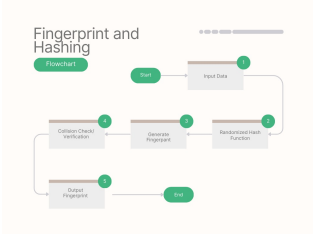

Fig. 4. finger print and hashing

Conclusion

Randomized algorithms have transformed computational paradigms, offering efficiency and simplicity in solving

problems across various domains. As computational challenges grow, their relevance and application will continue to expand.

References

1. GeeksforGeeks. - a randomized binary search tree.
2. GeeksforGeeks. Quicksort using random pivoting. Accessed: 2025-01-17.
3. GeeksforGeeks. Randomized algorithms. Accessed: 2025-01-18.
4. Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. *ACM Computing Surveys (CSUR)*, 28(1):33–37, 1996.

[4] [3] [1] [2]