Monte Carlo vs. Las Vegas Algorithms

Classical vs. Randomized Algorithms

Classical (Deterministic) Algorithms

Think of a standard algorithm you've learned, like MergeSort or Dijkstra's.

- **Input:** It takes an input .
- **Process:** It performs a fixed sequence of operations based on .
- **Output:** It produces an output .

$$\begin{tikzcd} \text{Eingabe } \mathcal{I} \arrow[r] & \boxed{\text{Algorithmus } \mathcal{A}} \arrow[r] & \text{Ausgabe } \mathcal{A}(\mathcal{I}) \end{tikzcd}$$

Key characteristics of a classical algorithm:

- **Correctness:** The output is always precise and correct for a given input .
- **Consistency:** For the same input , the output is always the same. (The algorithm behaves like a mathematical function.)
- **Deterministic Runtime:** For the same input , the runtime of algorithm is always the same.
- **Reproducibility:** The behavior of the algorithm is entirely reproducible. Give it the same input, and you get the same steps, same output, same runtime.

Randomized Algorithms

Randomized algorithms introduce an element of chance into the process.

- **Input:** They take an input .
- **Random Source:** They also have access to a source of randomness, let's call it . This could be a sequence of random bits, random numbers drawn from a distribution, etc.
- **Process:** The algorithm's operations can depend on both and the random values from .
- **Output:** The output is .

$$\begin{tikzcd} \text{Eingabe } \mathcal{I} \arrow[r] & \boxed{\text{Algorithmus } \mathcal{A}} \arrow[r] & \text{Ausgabe } \mathcal{A}(\mathcal{I},\mathcal{R}) \\ & \boxed{\text{Zufallsquelle}} \arrow[u, "\mathcal{R}"'] & \end{tikzcd}$$

Consequences of using randomness:

- The output now depends on both the input and the specific random choices made during execution.
- This means the output might be:
  - Sometimes correct.
  - Sometimes *almost* correct (e.g., a close approximation).
  - Sometimes fast (the runtime itself can be a random variable).
- **Non-Reproducibility:** Generally, if you run the algorithm twice with the same input but with different random choices from (which is typical unless you fix the random seed), you might get different outputs or different runtimes. However, the algorithm is still a deterministic function if you consider the pair as its combined input.

The Random Source

Where does this randomness come from?

- The random source provides **random, independent** bits or numbers, drawn according to some specified (often uniform) distribution.

Possible sources:

- **Physical Random Number Generators:** These leverage inherently random physical processes.
  - Examples: Lotto numbers, Geiger counters (radioactive decay), thermal noise in circuits, quantum phenomena. These are good sources of “true” randomness but can be slow or impractical for direct use in algorithms.
- **Deterministic (Pseudo-)Random Number Generators (PRNGs):** These are algorithms that produce a sequence of numbers that “look” random and pass many statistical tests for randomness, but are actually generated deterministically from an initial **seed** value.
  - Given a seed , a PRNG produces a sequence .
  - **Advantage:** If you know the seed, you can reproduce the exact same sequence of “random” numbers. This is invaluable for debugging randomized algorithms.
  - **Caveat:** PRNGs are not truly random. A sophisticated adversary who knows the PRNG algorithm and can observe enough output might predict future “random” numbers. The quality of PRNGs varies widely.

Our Assumption in Analysis

For theoretical analysis, we typically assume access to an **ideal random source** that provides perfectly random, independent bits/numbers as needed.

In practice, we use high-quality PRNGs. The potential discrepancy between theory (ideal randomness) and practice (pseudo-randomness) is something to be aware of, though often PRNGs are good enough for most applications.

Monte Carlo vs. Las Vegas Algorithms

Randomized algorithms are broadly categorized based on how randomness affects their correctness and runtime. The two main types are named after famous gambling locations: Monte Carlo and Las Vegas.

Monte Carlo [Alexander Migl]

Las Vegas [Bill Debevc]

Monte Carlo Algorithms

- **Correctness/Quality:** The correctness or quality of the output is a **random variable**. The algorithm might produce an incorrect answer or an answer of varying quality with some probability.
- **Runtime:** Typically, the runtime of a Monte Carlo algorithm is **not dependent on the random choices** (or is bounded deterministically). It usually runs for a predetermined number of steps.
- **Goal:**
    - Always (or predictably) fast.
    - *Mostly* correct or provides a *good quality* answer with high probability.

Las Vegas Algorithms

- **Runtime:** The runtime is a **random variable**. It might finish quickly on some random choices, and slowly on others.
- **Correctness/Quality:** The correctness or quality of the output is **not dependent on the random choices**. If the algorithm produces an answer, that answer is guaranteed to be correct.
- **Goal:**
    - Always correct/good (when an answer is given).

- *Mostly* fast (i.e., the expected runtime is good, or it finishes quickly with high probability).

Alternative View of Las Vegas Algorithms

Sometimes, a Las Vegas algorithm is designed such that it might explicitly output ”???” (or “I don’t know”) instead of a correct answer if it gets “unlucky” with its random choices within a certain time bound. The guarantee is that if it *does* give an answer other than ”???”, that answer is correct.

This leads to two common operational modes for Las Vegas algorithms:

1. **Repeat until an answer:** If the algorithm can output ”???”, you simply run it repeatedly until it gives a definite (and thus correct) answer. The analysis then focuses on the expected number of repetitions.
2. **Abort after fixed time:** Run the algorithm for a predetermined maximum time. If it produces an answer within this time, great. If not, it aborts and outputs ”???“.

A Las Vegas algorithm whose runtime is a random variable can be converted into one that sometimes outputs ’???’ by imposing a runtime cutoff (e.g., based on Markov’s inequality applied to its runtime).

Reducing Error Probability

One of the powerful features of randomized algorithms is that we can often decrease the probability of an undesirable outcome (like getting no answer, or getting a wrong answer) by simply running the algorithm multiple times.

Reducing “Failure to Answer” for Las Vegas Algorithms

Recall that a Las Vegas algorithm is always correct if it provides an answer, but it might sometimes fail to do so, perhaps by outputting a special symbol like ”???” or by exceeding a time limit.

Goal

Suppose we have a Las Vegas algorithm . For any input , it gives a correct answer with a probability of at least . This means the probability of it outputting ”???” is at most . Our goal is to construct a new algorithm, , that gives a correct answer with an even higher probability, specifically at least , where is some small target failure probability.

Idea: Repetition

The natural approach is to run the original algorithm multiple times. If any of these runs gives a definite answer, we can use it. But how many repetitions are enough?

Theorem: Amplifying Success for Las Vegas Algorithms

Let be a randomized algorithm that never gives a false answer. However, it might sometimes output ”???” (indicating no answer found). Let for any input .

For any desired overall failure probability :

Construct a new algorithm as follows:

1. Repeatedly call .
2. If any call to returns a value different from ”???”, then immediately stops and returns that value.
3. If has been called times and all calls have resulted in ”???”, then stops and outputs ”???“.

Then, the probability that provides a correct answer for input is at least . That is, .

Proof

The algorithm fails to provide a correct answer if and only if *every one* of its independent calls to results in ”???“.

The probability that a single call to results in ”???” is .

Since the calls are independent, the probability that all calls fail is: .

We want this failure probability to be at most . So we need to find such that .

We know that for any real number , . Applying this with : .

So, we need .

Taking the natural logarithm of both sides (which is a monotonically increasing function, so it preserves the inequality):

Multiplying by (and reversing the inequality sign):

.

By choosing , we satisfy this condition.

Therefore, .

The probability that provides a correct answer is . This completes the proof.

Example: Number of Iterations

Let’s see how many iterations are needed for some practical values. Suppose our base Las Vegas algorithm has a success probability (it finds an answer 1 out of 4 times on average). Then .

Target Failure Prob. ()	**Min. Iterations ()**	
0.1	10	0.9
0.01	19	0.99

Target Failure Prob. ()	Min. Iterations ()	
0.001	28	0.999
0.0001	37	0.9999
0.00001	47	0.99999
0.000001	56	0.999999

Key Observation: To decrease the failure probability by a constant factor (e.g., from to), the number of required iterations increases only by an additive constant amount (). This makes boosting the success probability of Las Vegas algorithms very efficient.

Reducing Error for Monte Carlo Algorithms

For Monte Carlo algorithms, which may produce incorrect answers, reducing the error probability by simple repetition isn't always straightforward.

General Case Limitation:

If a Monte Carlo algorithm is no better than random guessing (e.g., it flips a coin to decide between "JA" and "NEIN", and its probability of being correct is exactly), then simply repeating it and, say, taking a majority vote won't improve things. The error probability remains .

Error reduction for Monte Carlo algorithms *is* generally possible under two main conditions:

1. The algorithm exhibits **one-sided error**.
2. The algorithm's probability of being correct is **strictly greater than** (i.e., it has some "edge" over random guessing).

Monte Carlo with One-Sided Error

This is common in decision problems where an error can only occur for one type of instance (e.g., it might falsely identify a "NO" instance as "YES", but never a "YES" instance as "NO").

Definition (One-Sided Error Example)

An algorithm has one-sided error if, for a decision problem (JA/NEIN):

- If the input is a JA-instance: . (It's always correct).
- If the input is a NEIN-instance: . (It's correct with probability at least). This implies it erroneously outputs JA with probability at most .

(The roles of JA and NEIN can be swapped depending on the specific algorithm).

Theorem: Error Reduction for One-Sided Monte Carlo

Let be a Monte Carlo algorithm with one-sided error as defined above: . .

For any desired overall error probability :

Construct as follows:

1. Repeatedly call .
2. If any call to returns NEIN, then immediately stops and returns NEIN.
3. If has been called times and all calls have resulted in JA, then stops and outputs JA.

Then, the probability that gives the correct answer for input is at least . That is, .

Proof

We analyze the two cases for the true nature of input :

- **Case 1: is a JA-instance.** According to the properties of , every call to on a JA-instance will output JA. Therefore, will either see consecutive JAs and output JA (rule 3), or it would have stopped earlier if NEIN was possible (but it's not for JA-instances). So, correctly outputs JA. .
- **Case 2: is a NEIN-instance.** The algorithm makes an error if it outputs JA when is a NEIN-instance. This happens if and only if all independent calls to outputted JA. For a NEIN-instance, the probability that a single call to outputs JA (an error) is . So, . Using the same reasoning as in the Las Vegas proof, if we choose , then . Thus, .

Since is correct with probability 1 for JA-instances and with probability at least for NEIN-instances, its overall probability of being correct is at least .

Monte Carlo with Two-Sided Error (Correctness Probability)

Now, consider a Monte Carlo algorithm that always gives a JA or NEIN answer, but it can be wrong in either direction. However, it's better than a random guess: , for some .

Theorem: Majority Vote for Two-Sided Error Amplification

Let be a Monte Carlo algorithm such that for some . For any desired overall error probability :

Construct as follows:

1. Call a total of times independently.
2. outputs the answer (JA or NEIN) that occurred most frequently among the trials (the majority vote). If there's a tie, it can break it arbitrarily, e.g., output JA.

Then, .

Proof

We analyze a Monte Carlo algorithm that returns the correct answer with probability at least for some known . We define a new algorithm which runs independently times and returns the majority answer.

We show that:

Setup

Let:

-
-
- Let be the number of correct answers in independent runs of .
- So
- We want to bound:

Step 1: Expected Value of X

The expected number of correct runs is:

We now show that:

Step 2: Verifying the Chernoff Condition

We expand the right-hand side of (★):

So:

This holds for all , so (★) is valid.

Step 3: Applying Chernoff Bound

Using the Chernoff bound for the lower tail:

By (★), we have:

Step 4: Bounding the Exponent

We use:

So:

We want this to be at most :

Thus, choosing:

ensures:

Final Conclusion

With this value of , the amplified algorithm satisfies:

This shows that any Monte Carlo algorithm with correctness probability can be boosted to failure probability at most using:

independent repetitions and majority vote.

Randomized Algorithms for Optimization Problems

Randomized algorithms can also be applied to optimization problems (e.g., finding a maximum clique, minimum spanning tree, etc.).

Typical Scenario:

- The algorithm always produces a *feasible* (valid) solution.
- The *quality* of this solution is a random variable; it's not necessarily the optimal one.

Suppose for a maximization problem, we want a solution with value at least (our target quality for input). Assume our base randomized algorithm achieves this target quality with at least probability : .

Goal: Design an algorithm that achieves quality with a higher probability, at least .

Theorem: Amplification for Optimization by Repetition

Let be a randomized algorithm for an optimization problem (assume maximization without loss of generality). Suppose .

For any desired success probability (where is the failure probability):

Construct as follows:

1. Call the original algorithm a total of times independently.
2. outputs the solution that has the **best value** among all solutions obtained.

Then, .

Proof

The algorithm fails to achieve a solution of quality at least if and only if *every one* of the independent trials of produces a solution with value less than .

The probability that a single trial of fails to meet the target quality is .

Since the trials are independent, the probability that all trials fail is: .

This is the same mathematical situation as in the Las Vegas amplification. By choosing , we ensure that .

Therefore, the probability that succeeds (i.e., at least one of the trials achieves the target quality, and thus the best of them does) is .

This "repeat and take best" strategy is a common and effective way to boost the performance of randomized optimization algorithms.