Comparison Of The Levenshtein Distance Algorithm In Haskell, C# And TypeScript

Wessel Tip <contact@wessel.gg> (https://wessel.gg/)

Minor Advanced Software Engineering
NHL Stenden University of Applied Sciences
Year 3, Semester 2 (Jan. 2025 - Jun. 2025)

Contents

Dictionairy

Term	Meaning
ms	Milliseconds, one thousandth of a second.
us	Microseconds, one millionth of a second.
ns	Nanoseconds, one billionth of a second.
Programming Paradigm	A style of programming.
Imperative	A programming paradigm that uses statements to change a program's state.
Functional	A programming paradigm that treats programming as mathematical functions.
Semi-Functional	A mix between functional and imperative programming.
Mutable	A variable that can be changed.
Immutable	A variable that cannot be changed.

Abstract

This report will compare three different programming languages in performance and readability. Each programming language will implement a different programming paradigm.

Haskell was used as functional language, C# as imperative language, and TypeScript as semi-functional language.

The Levenshtein distance algorithm, which measures string similarity, was implemented in each language to evaluate performance and readability.

Standardized unit tests and benchmarking tools (Criterion, BenchmarkDotNet, Deno Bench) have been used to compare each implementation.

Results revealed significant performance differences. C# outperformed both TypeScript and Haskell, executing tasks up to $1000\times$ *faster in long-string comparisons. Haskell, while slower, demonstrated way better resource efficiency, using* $2\times$ *less memory compared to C# and TypeScript.*

These results highlight the trade-offs between execution speed and resource usage.

1 Introduction

This report will focus on understanding functional programming languages and their differences compared to imperative languages. The implementation of the Levenshtein ratio algorithm is discribed in three different programming styles and compares the results on multiple fronts.

The Levenshtein ratio algorithm is a commonly used algorithm to measure the similarity between two strings. It is applied in various fields such as spell checking and plagiarism detection. By implementing this algorithm in different programming languages, the performance and code complexity can be compared to understand the differences that exist between multiple programming paradigms.

The following languages were chosen for this comparison, each representing a different paradigm:

- **Haskell (Functional)** A purely functional language that emphasizes immutability and recursion. Known for its concise and expressive code.
- **Csharp (Imperative)** An object-oriented language mainly used to develop Windows applications and games.
- **TypeScript (Semi)** A language that supports both imperative and functional constructs. Known for its leniant interperter.

The comparison will be based on the following criteria:

I. **Performance** time complexity and resource usage of the algorithm.

II. **Readability** how simple and intuitive the code is.

II. **Maintainability** is the code easy to maintain and future-proof.

2 Theoretical Framework

2.1 Functional Programming

Functional programming is one of four programming paradigms that treats computation as mathematical functions where all data is immmutable. [1] This makes functional programming declarative. The program specifies what it should do, rather than how it should do it.

Functional programming is based on the concept of pure functions, which are functions that have no side effects and always return the same output for the same input.

Haskell

Haskell (or HS) is a functional programming language that is based on the lambda calculus. [2] Haskell is a pure functional language, which means that all functions in Haskell are pure functions. It is also a lazy language, meaning that expressions are only ran when their values are needed.

2.2 Imperative Programming

Imperative programming is one of the four programming paradigms that uses statements to change the state of a program. In imperative programming, the program is a sequence of statements that are executed in order. [3]

The state of the program is determined by mutable variables. Imperative programming is the complete opposite of functional programming mentioned in subsection 2.1, promoting the use of mutable variables.

The mutable nature of imperaetive programming makes it difficult to reason the behavior of a function, as the state of the program can change at any time.

CSharp

C# (or CSharp) is an object-oriented programming language that supports imperative programming. C# is widely used for developing Windows applications, web applications, and games.

2.3 The Levenshtein Distance Algorithm

The Levenshtein Distance algorithm is an algorithm used to calculate the similarity between two pieces of text. This is done by calculating the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into another. [4]

The result of the Levenshtein Distance algorithm is a value between zero and one that represents the number of edits re-

quired to change one string into another. This number is called the *Levenshtein Ratio*.

The pseudocode for the Levenshtein Distance algorithm with a complexity of $O(m \times n)$ is shown in 1.

Algorithm 1 Levenshtein Distance Algorithm [4]

function LEVENSHTEINDISTANCE(s, t)
  $n \leftarrow$ length of s.
  $m \leftarrow$ length of t.
  if $n = 0$ **then**
    return m
  end if
  if $m = 0$ **then**
    return n
  end if
  Create matrix d of size $0..m \times 0..n$.
  Initialize first row to $0..n$.
  Initialize first column to $0..m$.
  Examine each character of s (i from 1 to n).
  Examine each character of t (j from 1 to m).
  if $s[i] = t[j]$ **then**
    $c \leftarrow 0$
  else
    $c \leftarrow 1$
  end if
  $d[i,j] \leftarrow \min(\begin{cases} \text{Cell above} = [i, j-1] + 1 \\ \text{Cell left} = [di, j-1] + 1 \\ \text{Cell diagonal} = d[i-1, j-1] + c \end{cases})$
  return $d[n,m]$.
end function

3 Methodology

3.1 Unit Tests

To test the implementations, identical test datasets will be used across C#, Haskell and TypeScript. The following test cases will be used to test the correctness of the implementations:

- **#1: Empty** = (””,””)$->1$
- = (””,”x”)$->0$
  = (”x”,””)$->0$
- **#2: Nulls** = (””,$null$)$->0$
  = ($null$,””)$->0$
  = ($null$,$null$)$->0$
- **#3: Identical** = (”x”,”x”)$->1$
- **#4: Long** = ($lStringA$,$lStringA$)$->1$
- **#4: Long** = ($lStringA$,$lStringC$)$->0.571...$
- = ($lStringA$,$lStringB$)$->0$
- **#5: Different** = (”x”,”y”)$->0$
- **#6: Similar** = (”$kitten$”,”$sitting$”)$->0.571...$

3.2 Benchmarking

Criterion will be used to measure the performance of the Haskell implementation.

BenchmarkDotNet will be used to measure the performance of the C# implementation.

Deno Bench will be used to measure the performance of the TypeScript implementation.

perfcollect Will be used to collect performance data from all three implementations in the form of flame graphs. This will allow for optimisation of the implementations by changing the parts that take the longest time to execute.

3.3 Readability

To assess code quality, a scoring system from one to five will be used based on the following criteria:

- **Cyclomatic Complexity [5]**How easy is it to follow the code path.
- **Code Structure** Language-specific conventions (modularity in C#, purity in Haskell as example).
- **Code Depth** Based on nesting depth. it is best practice to have the lowest nesting depth possible.
- **Code Length** How concise the code is.

Scores were averaged across three developers familiar with all three languages.

4 Results

4.1 Benchmarks

This section will present the benchmarks of the *Levenshtein Distance algorithm* mentioned in subsection 2.3 across all three implementations. Each benchmark was executed on an *AMD Ryzen 5 5600X* CPU under the same controlled runtime environment.

4.1.1 C# (.NET)

Running C# with the DOTNET runtime consistently outperformed both TypeScript and Haskell, having the fastest execution times across all benchmarks. It was on average $3\times$ faster than TypeScript and over $1000\times$ faster than Haskell in some cases.

The performance gap to Haskell and TypeScript remained consistent for *short different strings* in *112.1 ns* and *short similar strings* in *193.0 ns*.

Long string comparisons saw minimal performance degradation compared to other implementations. *long different strings* and *long partially similar strings* both completed in *3.23 ms*, showing no performance difference inbetween completely and partially different strings. both cases were $2\times$ faster than TypeScript and over $300\times$ faster than Haskell.

4.1.2 TypeScript (Deno)

Running TypeScript in the Deno runtime showed an average improvement of 2× in execution time compared to Haskell, but TypeScript was still 3× slower than C# for all cases.

TypeScript showed a big performance gap between intirely different strings and partially similar strings. The function executed *short different strings* in *144.9 ns*, whilst *short partially different strings* averaged for *323.3 ns*. Showing a *2.23x* increase in execution time.

The time exponentially increased for long strings, with *long identical strings* taking *7.0 ms* and *long partially similar strings* taking *7.5 ms*. This is a 48275× (Or 194× per character) increase in execution time compared to short strings.

But the performance difference between completely and partially different strings did close down to a mere 1.1× increase when using long strings. Showing that the algorithm is more efficient when comparing long strings variating strings.

4.1.3 Haskell (GHCup)

Running Haskell with the GHCup compiler showed the slowest performance across all benchmarks. It was on average 10× slower than TypeScript and over 1000× slower than C# in long string comparisons.

The function executed *short identical strings* in *20.15 ns*, which was significantly slower than both TypeScript (*5.1 ns*) and C# (*1.41 ns*).

Execution times for *short different strings* (*992.3 ns*) and *short similar strings* (*2.41 s*) demonstrated inefficiencies in Haskell's string processing.

4.2 Resource Usage

Although Haskell scored the worst in execution time mentioned in subsection 4.1, it did use significantly less memory and CPU than both C# and TypeScript. This can be seen in Figure 1 and Figure 2.

Haskell only used on average 39 MiB of memory and 8.88% CPU, whilst C# used 61 MiB and 38.14% CPU, and TypeScript used 60 MB and 60.27% CPU.

4.3 Readability

Criteria	HS	C#	TS
Cyclomatic Complexity	4	3	3
Code Structure	5	4	4
Code Depth	4	3	4
Code length	5	4	3
Average	4.5	3.5	3.5

I. HS stands for Haskell
II. TS stands for TypeScript
III. Scores range from 1 - 5

The Haskell implementation scores highest due to its conciseness, functional purity, and minimal nesting.

The C# and TypeScript implementations are comparable to eachother in code readability.

5 Conclusion

C# has had the best execution speed, outperforming TypeScript by 3× and Haskell by over 1000× for long-string comparisons, showcasing that imperative programming is highly efficient for intensive tasks.

C# did consume considerably more memory (61 MiB) and CPU usage (38.14%).

Haskell, whilst being the slowest in execution time by far, did use 2× less memory (39 MiB) and 5× less CPU usage than C# and TypeScript. Making it an excellent choice for low resource environments where speed is not as important.

Its immutable nature and functional syntax also earned it the highest readability score (4.5), making it an excellent choice for maintainability.

TypeScript did not excel at anything compared to C# or Haskell, but also did not perform poorly. Making it a good middleground for projects on the web due to its ease of use.

These findings show that the choice of language depends on the project's priorities:

- **C#** for performance-sensitive systems
- **TypeScript** for small, balanced utilities
- **Haskell** for resource efficiency and code maintainability

Acknowledgements

The author would like to thank the following people for their contribution to this report:

Foppele, J.	Supervisor
Luciano, P.	Haskell code review
Ralph, D.	TypeScript code review
Siegmar , B.	CSharp code review

References

[1] P. Hudak, "Conception, evolution, and application of functional programming languages," *ACM Computing Surveys (CSUR)*, vol. 21, no. 3, pp. 359–411, 1989.

[2] G. Hutton, *Programming in haskell*. Cambridge University Press, 2016.

[3] D. Syme, A. Granicz, A. Cisternino, D. Syme, A. Granicz, and A. Cisternino, "Introducing imperative programming," *Expert F#*, pp. 69–100, 2007.

[4] D. K. Po, "Similarity based information retrieval using levenshtein distance algorithm," *Int. J. Adv. Sci. Res. Eng*, vol. 6, no. 04, pp. 06–10, 2020.

[5] C. Ebert, J. Cain, G. Antoniol, S. Counsell, and P. Laplante, "Cyclomatic complexity," *IEEE software*, vol. 33, no. 6, pp. 27–29, 2016.

6 Appendice

6.1 Graphs

Figure 1: Memory Usage whilst executing the algorithm 1000000 times.

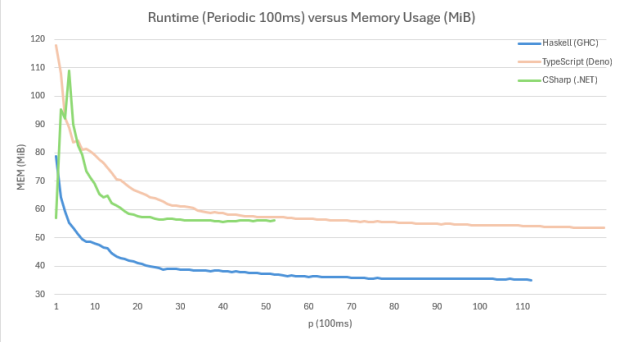
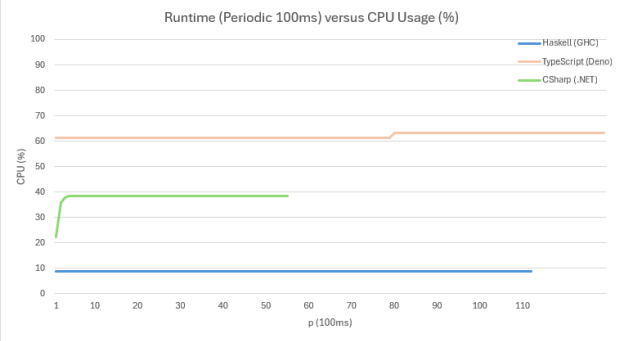


Figure 2: CPU Usage whilst executing the algorithm 1000000 times.



6.2 Benchmark Summaries

Figure 3: Benchmark results for the TypeScript implementation using the Deno runtime.

```
CPU | AMD Ryzen 5 5600X 6-Core Processor
Runtime | Deno 2.0.0 (x86_64-unknown-linux-gnu)

benchmark                                          time/iter (avg)        iter/s      (min ... max)           p75      p99     p995
-------------------------------------------------- ------------------------------ -------------------- -------------------------
levenshteinRatio - short identical strings                 5.1 ns   195,400,000 (  5.0 ns ...  24.6 ns)    5.0 ns   7.5 ns   9.2 ns
levenshteinRatio - short different strings                144.9 ns     6,904,000 (137.0 ns ... 182.7 ns) 148.7 ns 173.1 ns 182.5 ns
levenshteinRatio - short similar strings                  323.3 ns     3,093,000 (299.4 ns ... 613.6 ns) 325.8 ns 465.5 ns 613.6 ns
levenshteinRatio - long identical strings                   5.3 ns   187,100,000 (  5.1 ns ... 145.6 ns)    5.1 ns   7.9 ns   9.0 ns
levenshteinRatio - long different strings                   7.0 ms         142.0 (  6.5 ms ...  12.3 ms)    6.9 ms  12.3 ms  12.3 ms
levenshteinRatio - long partially similar strings           7.5 ms         133.6 (  7.2 ms ...   8.1 ms)    7.6 ms   8.1 ms   8.1 ms
```

Figure 4: Benchmark results for the C# implementation using BenchmarkDotNet.

```
Runtime = .NET 9.0.3 (9.0.325.11113), X64 RyuJIT AVX2; GC = Concurrent Workstation
Mean = 3.233 ms, StdErr = 0.004 ms (0.14%), N = 14, StdDev = 0.017 ms
Min = 3.204 ms, Q1 = 3.221 ms, Median = 3.233 ms, Q3 = 3.246 ms, Max = 3.259 ms
IQR = 0.025 ms, LowerFence = 3.184 ms, UpperFence = 3.283 ms
ConfidenceInterval = [3.214 ms; 3.252 ms] (CI 99.9%), Margin = 0.019 ms (0.59% of Mean)
Skewness = -0.19, Kurtosis = 1.65, MValue = 2

BenchmarkDotNet v0.14.0, Debian GNU/Linux 12 (bookworm) (container)
AMD Ryzen 5 5600X, 1 CPU, 12 logical and 6 physical cores
.NET SDK 9.0.201

| Method                       | Mean             | Error          | StdDev         |
|----------------------------- |-----------------:|---------------:|---------------:|
| ShortIdenticalStrings        |         1.410 ns |      0.0510 ns |      0.0793 ns |
| ShortDifferentStrings        |       112.147 ns |      1.4472 ns |      1.3537 ns |
| ShortSimilarStrings          |       193.013 ns |      1.8653 ns |      1.7448 ns |
| LongIdenticalStrings         |         1.097 ns |      0.0221 ns |      0.0196 ns |
| LongDifferentStrings         | 3,234,542.032 ns | 34,662.1297 ns | 32,422.9776 ns |
| LongPartiallySimilarStrings  | 3,233,473.960 ns | 18,980.6706 ns | 16,825.8701 ns |

  Mean   : Arithmetic mean of all measurements
  Error  : Half of 99.9% confidence interval
  StdDev : Standard deviation of all measurements
  1 ns   : 1 Nanosecond (0.000000001 sec)

Run time: 00:02:27 (147.1 sec), executed benchmarks: 6

Global total time: 00:03:08 (188.25 sec), executed benchmarks: 6
```

Figure 5: Benchmark results for the Haskell implementation using Criterion.

```
benchmarking levenshteinRatio - short identical strings
time                 20.15 ns   (19.87 ns .. 20.38 ns)
                     0.999 R²   (0.998 R² .. 0.999 R²)
mean                 20.15 ns   (20.00 ns .. 20.36 ns)
std dev              604.4 ps   (497.4 ps .. 753.7 ps)

benchmarking levenshteinRatio - short different strings
time                 966.6 ns   (953.2 ns .. 985.3 ns)
                     0.996 R²   (0.991 R² .. 0.999 R²)
mean                 992.3 ns   (970.7 ns .. 1.036 us)
std dev              97.98 ns   (50.78 ns .. 187.7 ns)

benchmarking levenshteinRatio - short similar strings
time                 2.410 us   (2.386 us .. 2.436 us)
                     0.999 R²   (0.999 R² .. 1.000 R²)
mean                 2.391 us   (2.374 us .. 2.417 us)
std dev              69.63 ns   (50.30 ns .. 104.4 ns)

benchmarking levenshteinRatio - long identical strings
time                 3.786 us   (3.765 us .. 3.815 us)
                     1.000 R²   (0.999 R² .. 1.000 R²)
mean                 3.815 us   (3.795 us .. 3.842 us)
std dev              79.62 ns   (63.85 ns .. 103.2 ns)

benchmarking levenshteinRatio - long different strings
time                 1.178 s    (847.8 ms .. 1.411 s)
                     0.991 R²   (0.970 R² .. 1.000 R²)
mean                 1.174 s    (1.121 s .. 1.245 s)
std dev              68.38 ms   (32.52 ms .. 85.07 ms)

benchmarking levenshteinRatio - long partially similar strings
time                 1.159 s    (816.3 ms .. 1.381 s)
                     0.990 R²   (0.968 R² .. 1.000 R²)
mean                 1.158 s    (1.120 s .. 1.214 s)
std dev              53.67 ms   (17.17 ms .. 70.20 ms)
```

Source Code

Figure 6: TypeScript implementation of the Levenshtein distance algorithm.

```typescript
export function levenshteinRatio(target: string, source: string): number {
  if (source === null || target === null) return 0.0;
  if (source === target) return 1.0;
  if (source.length === 0 || target.length === 0) return 0.0;

  const distanceMatrix = new Array(source.length + 1);
  for (let i = 0; i < distanceMatrix.length; i++) distanceMatrix[i] = new Array(target.length + 1);

  for (let i = 0; i <= source.length; distanceMatrix[i][0] = i++);
  for (let i = 0; i <= target.length; distanceMatrix[0][i] = i++);

  for (let i = 1; i <= source.length; i++) {
    for (let j = 1; j <= target.length; j++) {
      const cost = target.charAt(j - 1) === source.charAt(i - 1) ? 0 : 1;

      distanceMatrix[i][j] =
        Math.min(
          Math.min(
            distanceMatrix[i - 1][j] + 1,
            distanceMatrix[i][j - 1] + 1
          ),
          distanceMatrix[i - 1][j - 1] + cost
        );
    }
  }

  const totalDistance = distanceMatrix[source.length][target.length] / Math.max(source.length, target.length);

  return 1.0 - totalDistance;
};
```

Figure 7: C# implementation of the Levenshtein distance algorithm.

```
namespace LevenshteinRatio;

public static class Levenshtein {
  public static float Ratio(string target, string source) {
    if (source == target) {
      return 1;
    }

    if (string.IsNullOrEmpty(source) || string.IsNullOrEmpty(target)) {
      return 0;
    }

    var distance = new int[source.Length + 1, target.Length + 1];

    for (int i = 0; i <= source.Length; i++) {
      distance[i, 0] = i;
    }

    for (int i = 0; i <= target.Length; i++) {
      distance[0, i] = i;
    }

    for (int i = 1; i <= source.Length; i++) {
      for (int j = 1; j <= target.Length; j++) {
        int cost = target[j - 1] == source[i - 1] ? 0 : 1;

        distance[i, j] =
          Math.Min(
            Math.Min(
              distance[i - 1, j] + 1,
              distance[i, j - 1] + 1
            ),
            distance[i - 1, j - 1] + cost
          );
      }
    }

    var distanceAsFloat = Convert.ToSingle(distance[source.Length, target.Length]);

    return 1.0f - distanceAsFloat / Math.Max(source.Length, target.Length);
  }
}
```

Figure 8: Haskell implementation of the Levenshtein distance algorithm.

```haskell
module Lib (levenshteinRatio) where

import Data.Array

levenshteinRatio :: String -> String -> Double
levenshteinRatio source target
  | source == target = 1.0
  | otherwise =
      let distance  = fromIntegral (calculateDistance source target)
          maxLength = fromIntegral (max (length source) (length target))
      in 1.0 - distance / maxLength

calculateDistance :: String -> String -> Int
calculateDistance source target = finalDistance sourceLength targetLength
  where
    sourceLength = length source
    targetLength = length target

    distanceMatrix = array
      ((0, 0), (sourceLength, targetLength))
      [
        ((i, j), distanceAtIndices i j) | i <- [0 .. sourceLength],
        j <- [0 .. targetLength]
      ]

    distanceAtIndices 0 j = j
    distanceAtIndices i 0 = i
    distanceAtIndices i j =
      minimum
        [ distanceMatrix ! (i - 1, j) + 1,
          distanceMatrix ! (i, j - 1) + 1,
          distanceMatrix ! (i - 1, j - 1) + cost i j
        ]

    cost i j
      | source !! (i - 1) == target !! (j - 1) = 0
      | otherwise = 1

    finalDistance i j = distanceMatrix ! (i, j)
```