Seminar Report

Python Performance Optimization leveraging Native Implementations

Valerius A. Mattfeld

MatrNr: 11580056

Supervisor: Lars Quentin

Georg-August-Universität Göttingen
Institute of Computer Science

June 4, 2025

Abstract

High-Performance Computing (HPC) is important for modern research across disciplines. Python being a native programming language, due to its accessibility, popularity, and large ecosystem, poses a cornerstone in that field. However, Python is an interpreted language, which makes it slow by nature. This can create a bottleneck for the programming language before even running Python programs on such a cluster, which leads to inefficient resource utilization and execution times. Stand-art approaches to enhance performance in Python programs include algorithmic optimization as well as built-in libraries, and even complete rewrites of critical parts in a faster language. These solutions require a great deal of development effort and often expertise in that area, which researchers may lack.

This report evaluates various implementation approaches of the A* pathfinding algorithm. Starting with pure, naive Python, over standard library optimization, Nuitka compilation to binary files, Numba JIT compilation, Rust extensions with PyO3, and C++ extensions via pybind11. This tries to identify practical performance improvement strategies with several degrees of implementation complexity.

Benchmarking and analysis reveal that different problem sizes have different solutions in terms of measurement variations. Bigger workloads are better processed by C++ (6.51x speedup), and smaller workloads are better processed by Rust (14.2x speedup), solutions like Numba show inconsistent scaling behavior, and surprisingly, standard library optimizations offered no improvement over naive implementations.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts (basic autocomplete assistance and docstrings with `https://supermaven.com/`)
- ☐ For optimizing or restructuring software source texts
- ☐ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC	*High-Performance Computing*
API	*Application Programming Interface*
BFS	*Breadth-First-Search*
DFS	*Depth-First-Search*
JIT	*Just-In-Time*
FFI	*Foreign-Function Interface*

1 Introduction

1.1 Motivation

The four pillars of modern science consist of *theory*, *simulation*, *big data analytics* as well as *experimentation.* Computers are used for the latter three of those pillars.[1]

With reconstructing complete simulations of car accidents from a few photos, analyzing multiple magnitudes of parameters for weather predictions, the factor of computational performance is crucial.[8]

One of the most used languages in the area of machine learning is Python.[16]

Python has the benefits of being a mature language, is easy to learn and capable[16] These attributes of Python make it a fitting programming language which helps to implement and test thoughts and theories in high-velocity research areas, such as machine learning[16]

To quickly test and visualize ideas, researchers tend to use a library suite called `Jupyter`.[10] `Jupyter Notebooks`[2] is a feature of the `Jupyter Lab` suite, which allows the researcher to quickly scratch down some blocks of python code and markdown notes, and execute it in blocks.

Blocks of scratch code accumulate and can make up a whole research codebase.[20]

This is far from a performance-optimized solution.

To extend on this thought, the previously mentioned high-velocity of machine learning research community tends to publish python-only packages.

The same logic implemented in different ways in Python can have a significant impact on performance.[15].

It can not be assumed that researchers also have the ability to write performance-optimized code in addition to their professional skill set.

In a scenario where this software is to be run on HPC, those performance bottlenecks become a crucial factor in the efficacy of collecting information.

The lack of proper scripting, fast iterations, and the inability of the software produced to leverage the benefits of HPC could leave room for optimization.

In this report, we evaluate different approaches to enhance the performance of Python code, with several degrees of refactorization and reimplemantion to gain performance improvements.

Those factors influence the usage in both a computational factor and in a manpower factor, which influence the overall cost of using an HPC cluster to perform computations.

We will explore which solutions offer the best computational performance benefit, without involving a significant amount of manpower.

The chosen test set on which we measure the said improvements and factors consists of a maze generator and an implementation of the A*-Pathfinding algorithm. We test the efficacy of serveral implementations and frameworks, which are logically identical.

[1]`https://hps.vi4io.org/_media/teaching/autumn_term_2022/hpda22-01.pdf`, visited June 4, 2025.

[2]`https://jupyter.org/`, accessed June 4, 2025.

Those frameworks and implementations consist of:

1. a naive pure-pythonic version, Section 3.1
2. a `nuitka`[3] compiled version of the former, Section 3.3
3. a standard library extended version of 1., Section 3.2
4. a version leveraging the benefits of `numba`[4]
5. an implementation, which leverages the *Rust*[5] programming language in conjunction with `PyO3`[6], Section 3.5
6. and bindings written in C++, Section 3.6

The order of the listed implementations increases with the participation of software engineers and developers.

The A * algorithm has been chosen because it is well-studied[22] and non-trival, while being minimal enough to be used as a test.

1.2 Contributions

One goal is to examine the development experience of a researcher using the tools listed in Section 3. The implementations of the said variations can be found on GitHub, and PyPi.com for the published packages. Which are listed in Appendix A.3

The published packages double as a performance-enhanced A* Python library in general.

Furthermore, during a search for a maze generator, which products are used as a test data set are also published on GitHub[7]. It utilizes a modified version of Prim's maze generation algorithm[2], as well as some verification and maze generation parameters, e.g. number of valid paths.

1.3 Structure of this report

This report starts with Section 2, where we will introduce the A* Pathfinding algorithm, along with similar algorithms used in the project. In this chapter, Section 2.1.2 describes what sets the A* Pathfinding algorithm apart from the others. Also, for generating the test data, as mentioned in Section 1.2 - are mazes. A brief introduction to a published maze generator, which was created along with this project, is written in chapter Section 2.2. Furthermore, the technologies and frameworks used to enhance python performance outside its standard library are briefly elaborated in Section 2.3, Section 2.4, and Section 2.5.

Afterwards, Section 3 starts with a *naive implementation* in Section 3.1, which will serve as the baseline for benchmarking and performance comparison. Then, Section 3.2, describes how the standard library tries to optimize the program without using external Python libraries. Section 3.3 elaborates on how this alternative compiler bundles the

[3]`https://nuitka.net/`, visited June 4, 2025.
[4]`https://numba.pydata.org/`, visited June 4, 2025.
[5]`https://www.rust-lang.org/`, visited June 4, 2025
[6]`https://pyo3.rs/v0.15.1/`
[7]`https://github.com/valerius21/Stara-Maze-Generator`

naive program into an executable, and Section 3.4 explores the possibilities of *Just-In-Time* (JIT) features. Additionally, we also use Python extensions, which are written in Rust (Section 3.5) and C++ (Section 3.6).

We benchmark each implementation and look at its performance in Section 4. In Section 4.1, we describe the benchmark environment and the benchmark metrics, Section 4.2. The way in which the test data are generated and the results are stored is elaborate Section 4.3. The previous chapters including the benchmarking will yield the results for each framework, which are shown in Section 4.4. Those results are compared with each other in Section 4.5.

In Section 5, we discuss the probable causes of the observed performance gains and losses and close the report with a conclusion in Section 6.

2 Background

2.1 The A* Pathfinding Algorithm

With the issue of finding connected paths or grids, pathfinding algorithms are of the essence. Its use cases range from the navigation of streets with GPS[26] to programming movement algorithms in video game development [5].

Among those pathfinding algorithms, *Breadth-First-Search* (BFS), *Depth-First-Search* (DFS), and the A* Pathfinding algorithm are well studied.[9]

2.1.1 Other Pathfinding Algorithms

BFS searches all its neighbors in a graph before moving deeper, thereby exploring in a horizontal direction at first and yielding the shortest possible path.[9] This has the downside that this algorithm is not suitable for large spaces.

The maze generation in chapter 2.2 utilizes BFS to find at least one valid path in a given maze.

DFS In contrast, the DFS algorithm follows the connected nodes as deep as possible into the graph, before backtracking.[9] It does not guarantee the most efficient solution if multiple fitting paths are available, and can be stuck if it encounters an infinite solution. [9]

2.1.2 Heuristics of A*

What sets A* apart from classical pathfinding algorithms like the ones mentioned above is that it utilizes heuristics to prioritize paths that lead to the goal, thus balancing the following fundamental factors:

- $g(n)$ - The cost from the current node to the start node (**known cost**)
- $h(n)$ - The heuristic for estimating the cost from the current node to the end node (**future cost**)
- $f(n) = g(n) + h(n)$ - The total estimated cost of the path

To extend on the properties of A* and its heuristic function, A* itself becomes admissible if its heuristic is.[6][8]

In this experiment, we chose the *Manhatten distance* as our heuristic function:

$$h(n) = |x_1 - x_2| + |y_1 - y_1|$$

where, respectively, (x_1, x_2) and (y_1, y_2) are the current node and goal node.

This heuristic is fitting for our usecase, since we are dealing with a grid-like pattern, where the movements are limited to four, nondiagonal directions, namely up, down, left, and right.

2.2 Maze Generation

For this benchmark to work, we need a grid in which a path must be found. A grid, where some paths lead to a goal, is a maze. Therefore, the idea came to build a maze generator, which creates test data, fitting to our benchmarking requirements, but also can serve as a general maze generator for various sizes. The features include:

1. Batch generation of random mazes using a modified version of Prim's algorithm[2]
2. A configurable maze size with a specified minimum number of valid paths
3. Reproducable maze generation leveraging seeds
4. Exportable mazes as HTML visualization including solution paths and JSON serialization
5. A data structure, in which the maze can be represented, e.g. NumPy arrays

The maze generator's source code is on GitHub[9].

Although usable with a command line interface, as shown in Listing 1, it also provides a Python *Application Programming Interface* (API) (Listing 5), which we will import into the implementations mentioned in Section 3.

```
# Generate a default 40x40 maze
generate-maze

# Generate a 20x20 maze with custom start/goal positions
generate-maze --size 20 --start 0 0 --goal 19 19

# Generate a maze with a specific seed and show solution
generate-maze --seed 123 --draw-solution

# Generate a maze with more paths
generate-maze --min-valid-paths 5
```

Listing 1: Command Line usage of the maze generator

[8]https://courses.cs.duke.edu/fall11/cps149s/notes/a_star.pdf

[9]https://github.com/valerius21/Stara-Maze-Generator/, visited June 4, 2025.

The maze data structure is a 2D NumPy array, where its cells represent either a wall with the value `0` or a passage with `1`.

The tool has the additional feature of exporting Mazes in a visual representation having HTML, including color-coded cells for walls, passages, start, and goal; containing also a highlighted solution path.

The generator uses the BFS algorithm (Section 2.1.1) to find the path of the solution and validate that there is a fixed number of solutions. (Appendix A.3)

2.3 Nuitka

In essence, Nuitka compiles Python code to C extensions or standalone executables, requiring no additional syntax or decorators. This is optimal, when rewriting large source code bases is impractical due to its size or economical aspect.

Nuitka aims to maintain complete compatibility with the CPython interpreter, which is the standard interpreter for Python;[17] therefore, ensuring that the code behavior is identical to the execution of the typical Python program.

Optimization steps nuitka performs during the compiliation of the source program include:

- *Constant Folding* - Nuitka predicts certain pythonic expressions and substitutes them with the direct results, e.g. `5 + 6` will be replaced with `11`, and `range(3)` with the evaluated expression.
- *Constant Propagation* - Similar to constant folding, Nuitka can determine the values of variables where possible, thereby removing redundant computations.
- *Builtin Call Prediction* - Nuitka can predict results of built-in python functions like `type()`, `len()`, and, as mentioned, `range()`.
- *Exception Propagation* - Exception occurances, which can be detected at compile time, are optimized in terms of the available code paths, allowing the paths, which raise exceptions to be optimized during runtime.

The documentation shows those features activly being listed until version `2.0.6` (March 2024), where the current release version is `2.6`. This report uses the version `2.5.9` with the assumption that the optimizations above are still in place.[12]

For our purpose, Nuitka appears to be a simple fix to improve native Python code to enhance performance by compiling it down to a functionally identical C executable. With promising performance improvements, we can assume that rather simple approach researchers could improve the run-time performance of a python codebase with little time and extra technical knowledge outside the typical terminal workflow.

2.4 `numba`

Another python compiler that claims to be on a high-performant level is numba. Its main feature is that it uses JIT compilation in conjunction with parts of the LLVM[10] , while running the Python program.

[10]`https://numba.readthedocs.io/`

numba claims that it can run Python programs at runtime speeds that pair with FORTRAN and C.[11] Furthermore, its feature set extends to simple parallelization, compatibility with the CUDA library for GPU programming, and native integrations with scientific libraries such as numpy[12].

The documentation shows that there are several methods to make the Python program leverage the numba library.

The approach we will be following in this report is the recommended method of adding decorators to the Python functions, which we expect to have a significant runtime.

The relevant decorator would be `@njit`, which is an alias of the `@jit(nopython=True)` annotation. The option `nopython=True` which does the following:

> *[`nopython=True` is a] Numba compilation mode that generates code that does not access the Python C API. This compilation mode produces the highest performance code, but requires that the native types of all values in the function can be inferred.*[1]

Taking the previous points into account, a bare Python project would need refactors on its critical functions, which requires knowledge of the runtimes of the given functions.

Moreover, numba is rather picky and states in its "A 5 minute guide to Numba"[13] that it works best with math-heavy code and omits information on other operations.

Also, numba must be able to identify the types of python objects,[14] which can lead to heavy refactors.

2.5 `PyO3`

With PyO3 we are facing an extension module instead of a whole interpreter replacement for python. PyO3 provides Rust bindings and toolings for creating native python extentions.[7] It also supports Rust to python embedding.[7]

PyO3 provides an essential toolkit to complete most of the work. For example, for math-related use cases, it provides build-in packages to interface with numpy[7], asyncio[7], and converting utilites of structured data between those languages[7].

While being well documented, PyO3 requires basic knowledge of rust and a complete implementation of the functions that comsume significant time in it; thereby refactoring between languages to create those native extensions, requirering an in-depth knowledge on how software needs to be developed.

3 Methodology

3.1 A naive implmetation

As the baseline, we implement a pure pythonic version of the A* algorithm. The naive version and all subsequent versions will contain only the methods necessary to solve the problem of the shortest path with A*, and will import all other ones from the maze generator helper package from Section 2.2.

[11] `https://numba.readthedocs.io/`
[12] `https://numba.readthedocs.io/`
[13] `https://numba.readthedocs.io/en/stable/user/5minguide.html`
[14] `https://numba.readthedocs.io/en/stable/`

This variation is used to simulate python source code written by a novice to python, e.g. a researcher who occasionally uses python for computation.

The methods of the implemented search class are at least restricted to:

- `__init__(self, maze)` - the constructor, taking a maze to solve as a parameter
- `manhatten_distance(pos1, pos2)` - the heuristic function, implemented statically if possible following the definition in chapter 2.1
- `get_lowest_f_score_node(f_scores, open_set)` - Get the node with the lowest `f_score` from the open set. The open set are the nodes which are considered, and it yields the position with the lowest `f_score`, or shortest path up until then `find_path(start, goal)` - wraps all the components and the business logic of the A* algorithm, executes it and returns a list with nodes that represent the shortest path, if there is one.

Note that in all cases, the start and end goals are in the same positions, namely $start := (1, 1)$ and $goal := (maze_{width} - 1, maze_{height} - 1)$. Those are chosen that, for every test, we try to reach the furthest point inside the maze.

The implementation of the naive approach is found in `stara_astar_naive/stara_astar/astar.py`.[15]

3.2 Taking it further with the standard library

The Python standard library offers a number of packages to enhance datastructures and caching.

This version is intended to simulate the same algorithm but written by a more seasoned Python developer, which avoids any extra dependencies outside of Python itself.

The focus in this implementation is to leverage Python's focus on the last-recently used (LRU) cache to perform faster operations during the execution of the `find_path()` method.

In addition to helper functions and dataclasses, the logic of the source code is identical to the one provided in the chapter 3.1.

The code can be found in `stara_astar_stdlib/stara_astar_stdlib/a_star_stdlib.py`.[16]

3.3 Nuitka

This variant is identical to the naive version from chapter 3.1, but with the exception that we have added nuitka as a dependency to build an executable.

This is done by running the command in Listing 2.

After successfully compiling the source, we are left with a single executable binary.

[15] `https://github.com/valerius21/stara_astar_naive/blob/75e8a403dbe9351f08c6bc01de3d01887ae09862/stara_astar/astar.py`

[16] `https://github.com/valerius21/stara_astar_stdlib/blob/1a2e425ceb405940e222fbc530b472962adf0b77/stara_astar_stdlib/a_star_stdlib.py`

```
python -m nuitka --follow-imports --standalone \
    --onefile ./stara_astar_nuitka/astar_nuitka.py
```

Listing 2: Compiling Python source code into a binary executable with nuitka.

3.4 Leveraging `numba`

The numba variation was rather tricky to implement, since its support to classes is still experimental.[17] Therefore, refactoring has taken place to leverage the advantages of numba in conjunction with numpy, since the documentation claims first-class support for this library.[18] Otherwise, as discussed in chapter 2.4, the `@njit` decorator has been added to almost every critial function to enforce the use of the LLVM, and the avoidance of the Python interpreter.

Numba was also picky in terms of dependencies, since its compatibility with numpy versions is limited. In the experiment a release candidate (0.61.0-rc2) is used since it supports most of the new features needed of numpy (2.1.3).

This variant varies strongly to the previous variants, due to it requiring a rethinking position for datastructures in the ones provided by numpy, namely numpy NDArrays, which, in some way, simplifies the program code. The numba version still retains its API to have similar calls to the pathfinding algorithm as in the previous variants.

The code is available in `stara_astar_numba/stara_astar_numba/astar_numba.py`[19].

3.5 Using Rust with Python: `PyO3`

Entering the territory of Python extensions, we described the in and outs of PyO3 in chapter 2.5.

Python extensions are usually written in a lower-level language than C and C++.

Without question, this variant requires the knowledge of another programming language, namely Rust. Thereby, it is a more time intensive and difficult implementation compared to the previous variants for a researcher or user who does not have any knowledge in that regard. Additionally, knowledge of Python packge structuring and *Foreign-Function Interface* (FFI) is helpful when writing these extensions.

The goal with that variant is to write the runtime-expensive functionally in Rust, hoping to improve on runtimes, since the execution instruction do not need to be interpreted.

Starting a project with PyO3 is rather simple. Following the documentation[20], we create a new Python project with an attached virtual environment. Subsequently, a dependency `maturin` is installed. `maturin` helps to set up the python packge structure, as well as the rust project, which aims to compile to the extension.

First, we write the extension part. PyO3 helps to define the functions and objects that are relevant for bindings using Rust macros Listing 3.

For example, Listing 3 defines which classes and methods are visible to Python under which alias. Listing 4 defines a rust struct that will be visible to Python after compilation.

[17] `https://numba.readthedocs.io/en/stable/user/jitclass.html#jitclass`, visited June 4, 2025
[18] `https://numba.readthedocs.io/en/stable/user/vectorize.html`, visited June 4, 2025
[19] `https://github.com/valerius21/stara_astar_nuitka/blob/d803a5b5804f8e30e827da0fd2d2cd36c3b181dd/stara_astar_nuitka/astar_nuitka.py`
[20] `https://pyo3.rs/v0.15.1/`

```rust
#[pymodule]
fn stara_rs(m: &Bound<'_, PyModule>) -> PyResult<()> {
     m.add_class::<MazeSolver>()?;
     Ok(())
}
```

Listing 3: Declaring a module class in `PyO3` (Rust)

Furthermore, `#[pymethods]` will define which methods from the extension are visible to python.

```rust
#[pyclass]
struct MazeSolver {
    width: usize,
    height: usize,
    data: Vec<u8>,             // Flat array: 0 = wall, 1 = path
    visited: Vec<u8>,          // Visit tracking: 0 = unvisited,
                               //     1 = open, 2 = closed
    parent: Vec<u32>,          // Changed to u32 to support larger mazes
    open: [(i32, u32); 1024], // Changed to u32 indices
    open_len: usize,
}
```

Listing 4: Declaring a class with a struct in `PyO3` (Rust)

Implementing the extension was straightforward with basic rust knowledge. The documentation is clear and understandable. A package containing the binaries has been published in PyPi in order to simplify the installation process in other projects, such as the benchmarking project relevant in chapter 4.

The only drawback encountered was that to have a structured interface with Python classes, the extension needed to be wrapped in a simple class in order to be accessed similarly to the other variations.[7]

The extension module source is available on Github[21]

3.6 Classical bindings: A C++ interface

Another library, which helps to write Python extensions on a native level is `pybind11`[22]. The extensions are written in C++, and it helps to expose C++ types to Python and vise versa. This not only has the benefit of writing high-performant modules, but also allows us to integrate existing C++ codebases into Python.

The key features of `pybind11` start out with the integration of STL containers[21], smart pointers like `std::shard_ptr`[21], and numpy arrays. The support of `pybind11` contains the C++ versions C++11 to C++18.[14] Various optimizations are included at

[21]https://github.com/valerius21/stara_astar_rs

[22]https://github.com/pybind/pybind11, accessed June 4, 2025.

compile time, which are function signature precomputation, automatic function vectorization, and buffer protocols for better memory handling.[4]

The drawbacks are similar to those mentioned in Section 3.5, which includes that the binaries need to be compiled multiple times for various architectures and that software developers in contact with this framework need to know the basics of C++.

Implementing a C++ extension for Python with the help of `pybind11` requires additionally the knowledge of C++ build tools and, for an optimized implementation of a critical part of source code, a deeper knowledge of this language and memory management, making it by far the most difficult variation to implement.

The source code for this extension can also be found on GitHub[23].

4 Performance Evaluation

In this chapter, we describe the benchmark environment (Section 4.1) in which the hardware specifications and programs are listed. Section 4.2 elaborates which benchmarking tools and functions that are used to measure the results of the tests. Then Section 4.3 explains how the test data set is generated and set up. The next part, Section 4.4, will show the results found for each implementation, which is followed by Section 4.5 where the results from the latter chapter are compared with each other.

4.1 Benchmark Environment

The benchmarking is performed on a single MacBook Pro 16" (2021) with an M1 Pro processor that includes 32 gigabytes of RAM. Every non-critical process is terminated during the benchmarking, and the device is fully charged running on power-supply consumption. Implicitly, the device is running macOS Sequoia 15.3.1. with the Python version being `3.13.2` and `clang-1700.0.13.3`.

4.2 Benchmark Metrics

The benchmarking will measure the median of 1000 executions in nanoseconds for each implementation listed in the Section 3 per run. The benchmark is designed to measure only the execution time of the `find_path` function, which effectively executes the pathfinding algorithm. To measure each function call, the `perf_counter_ns`[24] in-built python function is used.

Since the `perf_counter_ns` function utilizes the highest-resultion clock available for the current system.

The configuration can vary for different hardware setups.[25]

In order to measure the performance of the various implementations from Section 3, different maze sizes have been generated. The mazes are reproducable by seed, as mentioned in Section 2.2. The generated sizes are 10x10, 100x100, and 1000x1000. For each size, 1000 mazes have been generated, and across all implementations, each pathfinding implementation has been run 1000 times. The average values of each run for each implementation are saved. The goal here is to analyze and compare their performance

[23]`https://github.com/valerius21/stara_cpp`, visited June 4, 2025.

[24]`https://docs.python.org/3/library/time.html#time.perf_counter`, visited June 4, 2025.

[25]`https://docs.python.org/3/library/time.html#time.get_clock_info`, visited June 4, 2025.

characteristics. This also implies that with increasing maze size, the complexity of the paths within the mazes also increases significantly, which should affect overall execution time and efficiency. The possibility of having at least three valid paths forces the path-finding algorithms to evaluate the shortest one with retrospect to its specification.

The complete implementation of the benchmarking programme can be found in the linekd repository in Appendix A.3.

4.3 Test Dataset Design

In addition to the different maze sizes of 10x10 (small), 100x100 (medium), and 1000x1000 (large), we also fix the start and goal coordinates of each maze, with the start coordinate at $(1, 1)$ and the goal at $(maze_{size} - 1, maze_{size} - 1)$.

It is sufficient to retain the seed of the generated maze in order to reconstruct it, allowing one to reduce the benchmarking data size while also retaining the option to repoduce mazes on-demand.

We will utilize a benchmarking generator tool from Appendix A.3, written entirely for this purpose. The capabilities of the benchmarking generator tool allow one to generate a given number of test mazes from seeds, the seeds themselves, and save it to a python pickle binary file[13]. An example call to the benchmark generator tool can be found in Listing 1.

4.4 Results by Implementation

After running the benchmarks, the results for the average times as described in Chapter 4.2, we get python pickle files containing the execution times. Table 1, Table 2, and Table 3 show the first five results for each run with respect to its size.

Table 1: Rounded execution times (ns) for Maze Size **10x10** (first five rows), `seed`, and `size` columns are omitted.

`naive`	`stdlib`	`pyo3`	`numba`	`cpp`	`nuitka`
1832	3118	134	886551	406	10769
2800	4725	198	1580	653	11517
2819	4714	195	1587	651	11553
2833	4605	194	1607	653	10015
2794	4834	194	1577	650	11986

Table 2: Rounded execution times (ns) for Maze Size **100x100** (first five rows), `seed`, and `size` columns are omitted.

`naive`	`stdlib`	`pyo3`	`numba`	`cpp`	`nuitka`
1558	2860	259	83772	405	11322
2828	4384	434	26094	445	10817
2778	4732	425	26242	482	12534
2802	4425	435	26176	443	10590
2772	4468	433	26089	446	10903

In Figure 5, the performance for the naive implementation is shown (`naive`). Figure 6 shows the standard library implementation (`stdlib`); Figure 9, Figure 8 and Figure 7

Table 3: Rounded execution times (ns) for Maze Size **1000x1000** (first five rows), `seed`, and `size` columns are omitted.

naive	stdlib	pyo3	numba	cpp	nuitka
1608	2923	8990	3238974	426	11098
2798	4620	10624	2982014	427	10476
2841	4803	11061	2934932	427	9463
2872	4531	10798	3150010	428	10481
2849	4722	10768	3207305	427	11071

show the performance of the implementation of nuitka (`nuitka`), numba (`numba`), and PyO3 (`pyo3`) implementation respectively. Finally, we have Figure 10, displaying the performance of the C++ implementation (`cpp`).

4.5 Performance Comparison

When we look at Figure 1, we set up the implementations for comparison. There is a strong indication that the performance of the `numba` implementation strongly deviates from the others. Therefore, we exclude it in some visualizations in this chapter.

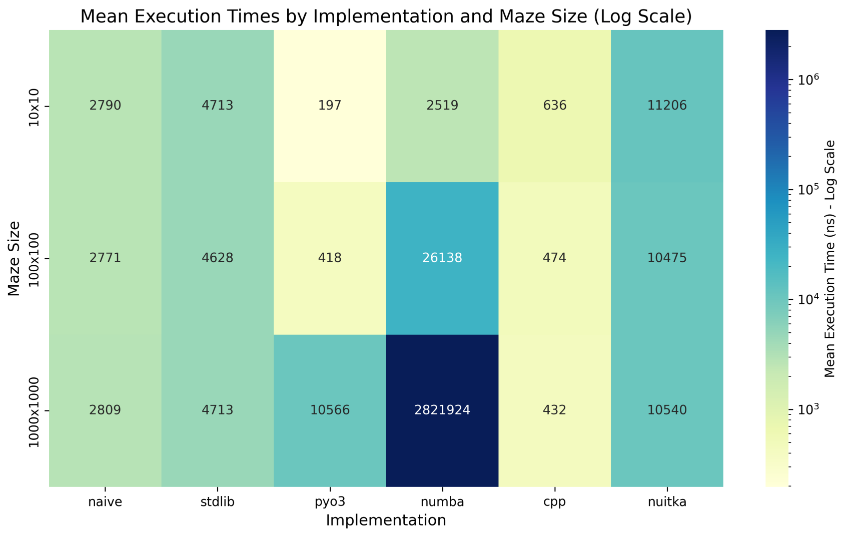

Figure 1: Performance of all implementations across all maze sizes.

Relative to the naive implementation, we get a better comparison in Figure 2, visualizing the performance gains.

In Figure 3 and Figure 4, we can observe how the implementations scale on problem size ran on the same dataset in terms of mean execution times.

Especially when comparing the performance gains on small problem sizes, the native solutions shine with a relative speedup of 14.2x with PyO3 for 10x10 and 6x on 100x100 computations. By far the best overall performance is seen with the C++ extensions, even showing a performance increase with growing problem size from 4.38x at 10x10 to 6.51x for 1000x1000 grids compared to the naive implementation.

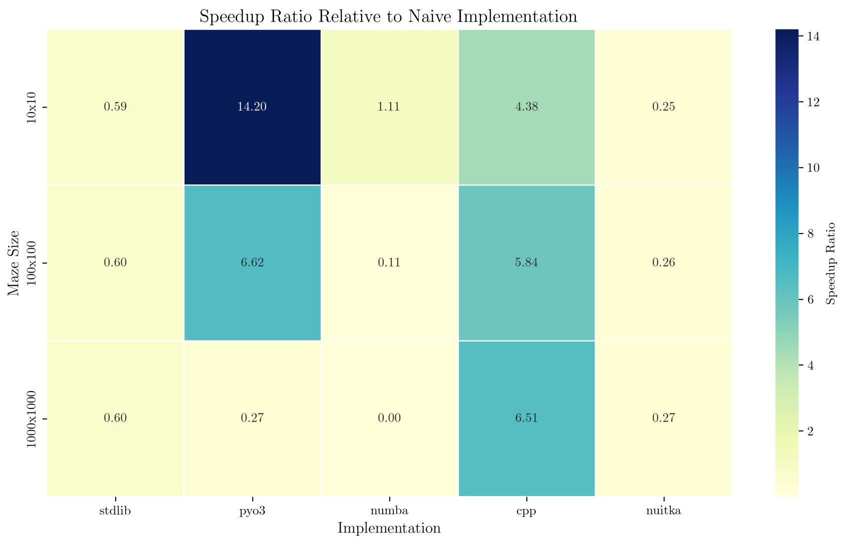

Figure 2: The relative improvement in performance in comparison to the naive implementation. Values ≤ 0 are have a worse performance than the naive implementation.

5 Discussion

In this section, we interpret the results of Section 4. Starting with the implementation trade-offs in Section 5.1, where we look at the different implementations and their possible causes for execution time speed-ups and slow-downs, we will look at the researcher or user experience for adjusting existing code bases in Section 5.2 and the relevance of those results for HPC in Section 5.3

5.1 Implementation Trade-offs

As we can observe, for example in Figure 4, Figure 1, and Figure 3, the naive implementation is not the worst performancewise.

5.1.1 Standard Library

Suprisingly, the standard library refactor has consistently shown no improvement at all for all size problems. As we establish later in Section 5.1.5, calls to C or C++ create an additional overhead for the execution time. The Python Standart library makes use of underlying C-optimized source code,[26], possibly slowing the whole process down with a nonbeneficial call-overhead to computational-benefit ratio.

5.1.2 nuitka

The compilation of an executable binary file with `nuitka`, similar to the implementation of the naive and standard library, was stable in its execution times in all problem sizes.

This problem can be caused by a number of reasons. Starting with the nuitka startup overhead can significantly slow the program down, [25] However, the Python source code for nuitka accounted for that by saving the results in memory while benchmarking without

[26]`https://github.com/python/cpython/blob/main/Modules/mathmodule.c`, visited June 4, 2025

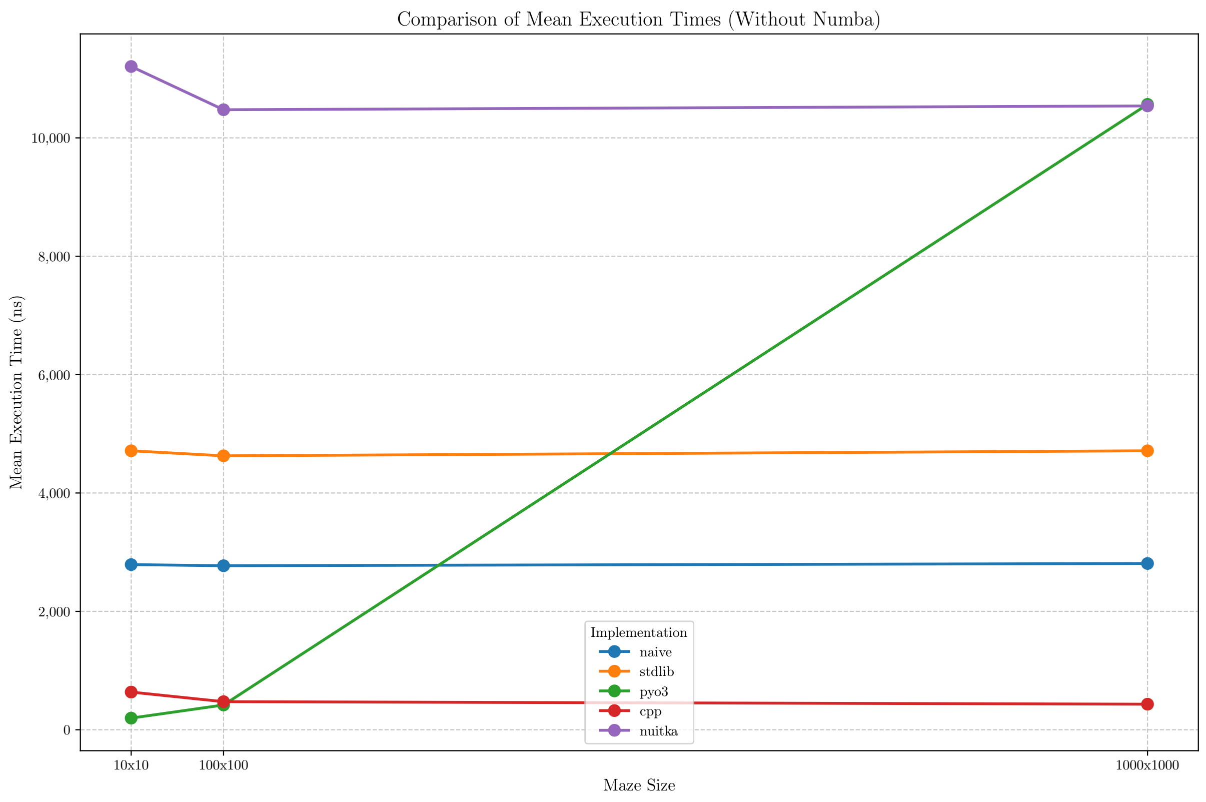

Figure 3: Comparison between mean execution times excluding `numba`

taking the startup into account. (Appendix A.3) Another reason, the nuitka documentation for performance issues lists, is the issue of *dependency creep*, [19] Here, the improper inclusion of Python dependencies can cause a fallback to a slower Python API call. Also, there is also the possibility of static linking issues in relation to `libpython`, a shared object from Python,[11]However, currently there is no obvious way to confirm this. Therefore, the question arises of whether nuitkas performance issues are caused by improper compilation, dependencies, or that this benchmark stands as a particular difficult job for this compiler.

5.1.3 `numba`

`numba` provides a 1.11x increse in speedup when run on small problem sizes (10x10), but falls off for medium sized problems, and is significantly slower than any other solution when it comes to large problem sizes.

Combined with dependency compatibility issues, mentioned in Section 2.4, Section 3.4, a heavy refactor, which numba requires for any nonmathemetics-related use case, makes it an unsuitable choice for consideration.

One performance issue could pose the compilation overhead at the beginning of the first function call, which, however, seems unlikely, since the source code remains identical across all problem sizes and even creates a speedup for small mazes.

Another cause could be parallelization overhead, which could create thread-management and syncronization issues. Since we are executing the path-finding algorithm sequentially, there is no reason to assume that this occurs.

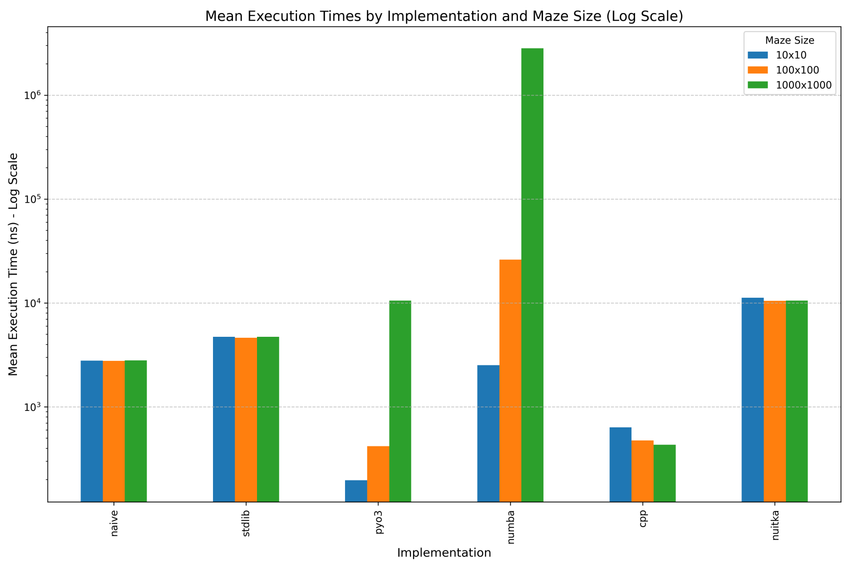

Figure 4: Mean performance across all implementations compared.

Heavily reliant on simple code patterns, but offering first-class support for the NumPy library; on the one hand, it seems unlikely that using vectorized operations including temprary memory allocation could cause a performance issue. On the other hand, the documentation states that exactly this usage could cause such a slowdown [23] [24] Therefore, making numba a difficult decision, and moreover not a recommendation for performance increasing measures, when considering the refactoring drawbacks and unpredictable scaling behavior for recommended practices.

5.1.4 PyO3

PyO3 shines, when used in conjunction with small and medium workloads. It outperforms every other implementation in that sector, increasing the speed by **14.2x for 10x10** mazes and **6.62x for 100x100** mazes, making it an excellent addition if the prerequisitions are met. Those include a basic knowledge of Rust and the availability of resources for refactoring an existing codebase.

An indication why PyO3, or Rust in general can fall off with large array sizes can be found on this GitHub Issue[27]. In summary, large-array transformation and copying can add a significant amount of execution time.

Optimization in that regard (e.g. through partial array splitting) could reduce the execution time of the PyO3 implementation.

The beforementioned benefits could greatly reduce cost for fitting scenarious, possibly justifying resource expanses in that regard.

5.1.5 C++ with `pybind11`

The best overall performance for all size problems has the C++ extension with `pybind11`. Ranging from **4.38x (10x10) for 6.51x (1000x100)**, even showing a slight speed-up for larger problem sizes.

The benefits of C++ are that it can be highly optimized. Compilers can also optimize the code. The Python-to-C++ interface overhead is significant for smaller function calls,

[27]`https://github.com/PyO3/pyo3/issues/3787`, visited June 4, 2025.

[3] However, when heaver workloads are processed, the benefits outweigh this issue [18] This reports implementation does not require any external system dependencies, and therefore can operate completely on its own; making it with the beforementioned reasons the best recommendation for performance improvements, if the resources are available.

5.2 Researcher Experience Analysis

In retrospect with the researching user that wants to optimize the execution time of Section 1, the best course of action would be to remain with the naively implemented program code. If such a user has any knowledge, even on a novice level, he could attempt to optimize the performance-ciritcal section with PyO3 or C++, which could result in significant performance benefits. For long running programs, the saved execution time must outweigh the time spent refactoring in order to reap the benefits of optimizing those code passages. Furthermore, debugging a multi-language project adds additional complexity for maintenance and should be carefully considered.

5.3 HPC Relevance

When dealing with slow Python codebases for long running experiments, a few optimizations in performance critical areas could speed up the execution time of those programs significantly, saving computational power, and therefore provide a HPC cluster with more capacities for other computations. In conjunction with heavy workloads per node, the C++ extension could pose as a possible option for improvement. However, if smaller workloads are of the essence, Rust extensions could create significant performance benefits over Python.

6 Conclusion

6.1 Summary

In this report, we evaluated a number of approaches to obtain a speedup of execution time for Python code. For that, we focused on different implementations of the A* pathfinding algorithm. In the benchmark, detailed in Section 4.5, we saw that there are significant performance differences between implementations.

The performance across all impellers, which are:

- naive, for the baseline implementation
- stdlib, for the standard library implementation
- nuitka, the binary executable variant
- numba, for JIT variation
- Pyo3, as a Rust extension
- C++, as a C++ extension with `pybind11`

showed, that not all approaches yield better results than our baseline measurement. The data shown in Figure 4, and Figure 2, indicate that, suprisingly, event the standard library variation has no improvement over the naive. This is possibly due to the overhead of calling C-optimized code from the standard library, which outweighs the computational advantages Section 5.1.

Furthermore, the PyO3 implementation showed remarkable performance for small and medium workloads, creating a 14.2x speed increase for 10x10 mazes and 6.62x for mediums, compared to the baseline approach. This makes extension with PyO3 a suitable choice when working within these problem sizes (Section 5.1).

The best overall performance gain was achieved with the C++ extensions across all problem sizes. The speed ups range from 4.38x to 6.51x for 10x10 and 1000x1000 mazes, respectively, even gaining relative performance for increasing problem sizes, which is elaborated in Section 5.1.5.

Against initial expectations, the numba variant was only notibly faster on smaller problemsets, with a 1.11x increase in speed for 10x10 mazes, but degraded significantly on others. That makes it the worst performer for larger problem sets. The issues, as discussed in Section 5.1, indicate that numba is limited in regard to complex non-math-heavy applications.

We also examine the experience of researchers or users in Section 5.2. A critical trade-off between development effort and performance gain is discussed. For non-critical Python code, optimizations should not take place, but on the flipside circular passages, which get called frequently, could pose as a potential refactoring point into either C++ or Rust. This could create substantial time savings in the long run, which is especially beneficial for HPC environments, as noted in Section 5.3.

6.2 Future Work

Expanding the benchmark not only to pathfinding algorithms, but other computations could reveal the benefits of the other considered frameworks, which did perform worse than our baseline implementation.

Evaluation on different hardware in comparison to the one described in Section 4.1, e.g., CPU architectures and operational systems, could reveal more insight in bigger performance deltas.

A paralell implementation of the A* Pathfinding algorithm could pose as a more accurate benchmarking model for cluster environments like HPC-Clusters, furthermore evaluating the benefits of Python code optimization with the used frameworks in this report.

Finally, exploring hybrid solutions, consisting of PyO3 optimized extensions for small workload parts of a program in conjunction with C++ extensions for high workload parts, could result in the best possible performance across all workload sizes, leveraging the strength of both technologies as identified in Section 4.5.

References

[1] *5. Glossary — Numba 0.17.0-Py2.7-Linux-X86_64.Egg Documentation.* URL: `https://numba.pydata.org/numba-doc/0.17.0/glossary.html` (visited on 04/06/2025).

[2] *Buckblog: Maze Generation: Prim's Algorithm.* URL: `https://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm` (visited on 04/06/2025).

[3] *Calling a C++ Function Is Roughly 2 Times Slower Then Calling a Native Python Function · Issue #2005 · Pybind/Pybind11.* GitHub. URL: `https://github.com/pybind/pybind11/issues/2005` (visited on 04/06/2025).

[4] *Frequently Asked Questions - Pybind11 Documentation.* URL: `https://pybind11.readthedocs.io/en/stable/faq.html` (visited on 04/06/2025).

[5] Francesco Garavaglia et al. "Moody5: Personality-biased Agents to Enhance Interactive Storytelling in Video Games". In: *2022 IEEE Conference on Games (CoG)* (Aug. 21, 2022). [TLDR] Moody5 is proposed, a preliminary solution designed to help game designers create "personality-biased" agents able to interact in sensible ways in the framework of interactive storytelling and could improve the gameplay experience and replay value while providing a helpful Unity plugin for game developers., pp. 175–182. DOI: `10.1109/CoG51982.2022.9893689`. URL: `https://ieeexplore.ieee.org/document/9893689/` (visited on 04/06/2025).

[6] *Heuristics.* URL: `https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html` (visited on 04/06/2025).

[7] *Introduction - PyO3 User Guide.* URL: `https://pyo3.rs/v0.15.1/` (visited on 04/06/2025).

[8] Mandara G S and Prashant Ankalkoti. "Car Damage Assessment for Insurance Companies". In: *International Journal of Advanced Research in Science, Communication and Technology* (June 23, 2022). [TLDR] In this proposed project the insurance company can machine-driven the car damage analysis process without the need for humans to analyse the damage done to the car., pp. 431–436. ISSN: 2581-9429. DOI: `10.48175/IJARSCT-5048`. URL: `http://ijarsct.co.in/june6i.html` (visited on 04/06/2025).

[9] Mochammad Darip et al. "Comparison of BFS and DFS Algorithm for Routes to Historical-Cultural Tourism Locations in Banten Province". In: *Journal of Advances in Information and Industrial Technology* 6.2 (Oct. 11, 2024). [TLDR] This research aims to maximize tourists' experience in visiting historical-cultural tourist attraction locations in Banten Province by choosing optimal travel routes, thereby increasing visit efficiency, minimizing travel time and distance, and enabling them to visit more locations in a limited time., pp. 113–122. ISSN: 2716-1927, 2716-1935. DOI: `10.52435/jaiit.v6i2.560`. URL: `https://journal.ittelkom-sby.ac.id/jaiit/article/view/560` (visited on 04/06/2025).

[10] Quantum News. *Jupyter Notebooks.* Sept. 23, 2024. URL: `https://quantumzeitgeist.com/jupyter-notebooks-2/` (visited on 04/06/2025).

[11] *Nuitka: An Extremely Compatible Python Compiler | Hacker News.* URL: `https://news.ycombinator.com/item?id=28377541` (visited on 04/06/2025).

[12] *Nuitka: Python Compiler with Full Language Support and CPython Compatibility.* Version 0.5.14. URL: http://nuitka.net (visited on 04/06/2025).

[13] *Pickle — Python Object Serialization.* Python documentation. URL: https://docs.python.org/3.13/library/pickle.html (visited on 04/06/2025).

[14] *Pybind/Pybind11.* pybind, Apr. 6, 2025. URL: https://github.com/pybind/pybind11 (visited on 04/06/2025).

[15] *Python Has a Reputation for Being Slow, Is That Reputation Warranted Even You're Using Numpy Effectively? - Python Help.* Discussions on Python.org. Nov. 22, 2024. URL: https://discuss.python.org/t/python-has-a-reputation-for-being-slow-is-that-reputation-warranted-even-youre-using-numpy-effectively/72044 (visited on 04/06/2025).

[16] *Python Stretches Lead in Language Popularity Index.* InfoWorld. URL: https://www.infoworld.com/article/2336419/python-stretches-lead-in-language-popularity-index.html (visited on 04/06/2025).

[17] *Python/Cpython: The Python Programming Language.* URL: https://github.com/python/cpython/tree/main (visited on 04/06/2025).

[18] Arjun Sahlot. *Speeding up Python 100x Using C/C++ Integration.* Arjun's Blog. Apr. 14, 2023. URL: https://arjunsahlot.hashnode.dev/speeding-up-python-100x-using-c-integration (visited on 04/06/2025).

[19] *Solutions to the Common Issues — Nuitka the Python Compiler.* URL: https://nuitka.net/user-documentation/common-issue-solutions.html (visited on 04/06/2025).

[20] Laszlo Sragner. *How Can a Data Scientist Refactor Jupyter Notebooks towards Production-Quality Code?* Deliberate Machine Learning. Oct. 24, 2021. URL: https://laszlo.substack.com/p/how-can-a-data-scientist-refactor (visited on 04/06/2025).

[21] *STL Containers - Pybind11 Documentation.* URL: https://pybind11.readthedocs.io/en/stable/advanced/cast/stl.html (visited on 04/06/2025).

[22] *The A* Algorithm: A Complete Guide.* URL: https://www.datacamp.com/tutorial/a-star-algorithm (visited on 04/06/2025).

[23] *Troubleshooting and Tips — Numba 0.52.0.Dev0+274.G626b40e-Py3.7-Linux-X86_64.Egg Documentation.* URL: https://numba.pydata.org/numba-doc/dev/user/troubleshoot.html (visited on 04/06/2025).

[24] Itamar Turner-Trauring. *The Wrong Way to Speed up Your Code with Numba.* PythonSpeed. Mar. 21, 2024. URL: https://pythonspeed.com/articles/slow-numba/ (visited on 04/06/2025).

[25] user258532. *PyPy vs. Nuitka.* Stack Overflow. Dec. 27, 2017. URL: https://stackoverflow.com/q/47992232 (visited on 04/06/2025).

[26] Wirarama Wedashwara et al. "Data Storage and Modeling System for GPS, Gyro and Camera Data Using Apache Flume and Hadoop Map Reduce". In: (2024), p. 050027. DOI: 10.1063/5.0200515. URL: https://pubs.aip.org/aip/acp/article-lookup/doi/10.1063/5.0200515 (visited on 04/06/2025).

A Appendix

A.1 Means by Implemenation

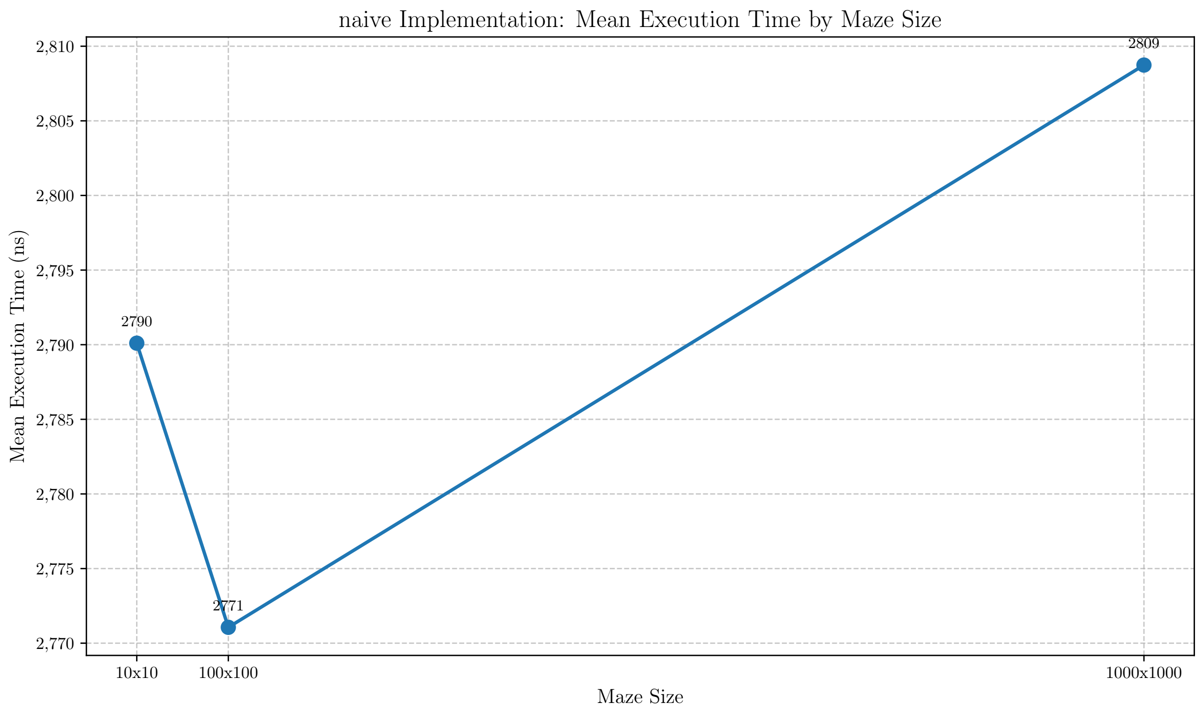

Figure 5: Mean execution Time of the Naive Implementation Across Maze Sizes

A.2 Code Snippets

A.3 Contribution List

- `https://github.com/valerius21/stara_cpp`
- `https://github.com/valerius21/stara_astar_nuitka`
- `https://github.com/valerius21/stara_astar_rs`
- `https://github.com/valerius21/stara_astar_naive`
- `https://github.com/valerius21/stara_astar_numba`
- `https://github.com/valerius21/stara_astar_stdlib`
- `https://github.com/valerius21/Stara-Maze-Generator`
- `https://github.com/valerius21/stara-batch-benchmark`

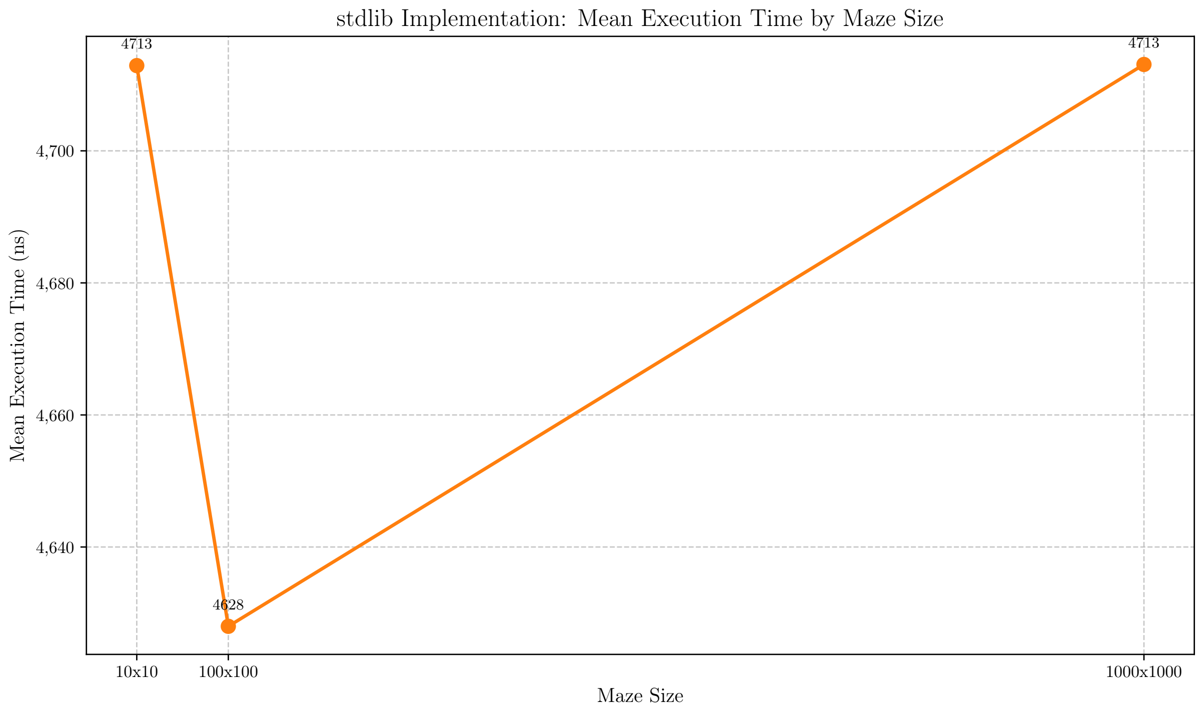

Figure 6: Mean execution Time of the Standard Library Implementation Across Maze Sizes

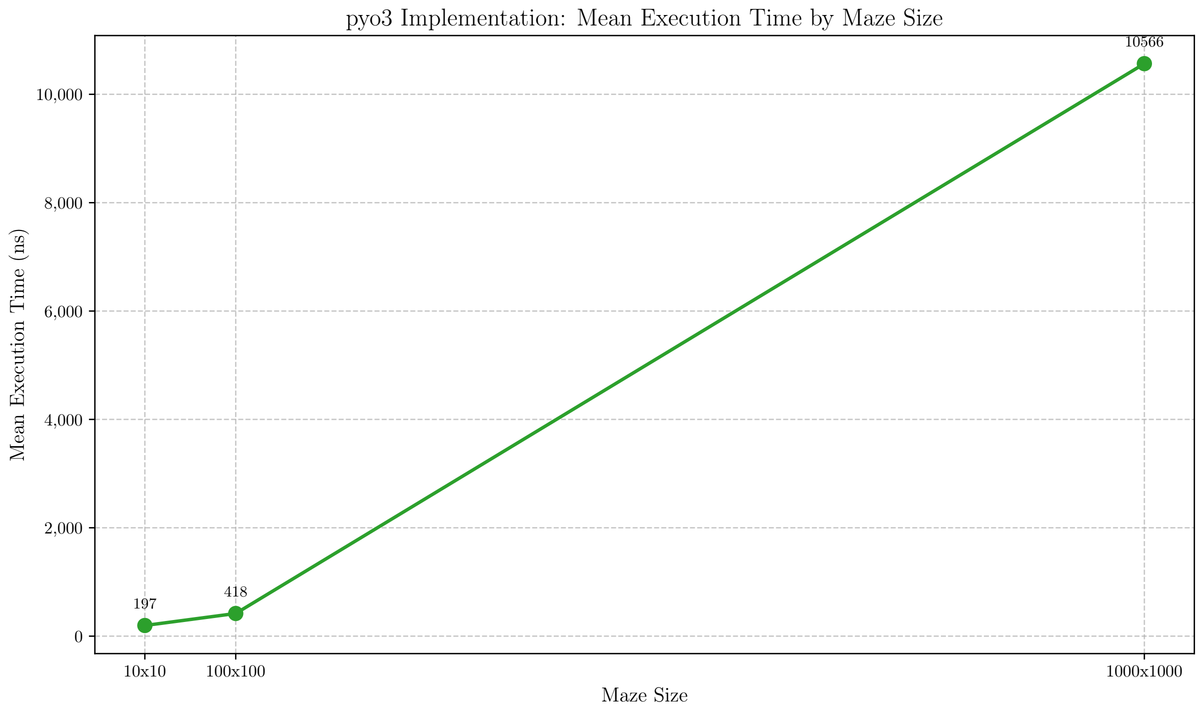

Figure 7: Mean execution Time of the PyO3 Implementation Across Maze Sizes

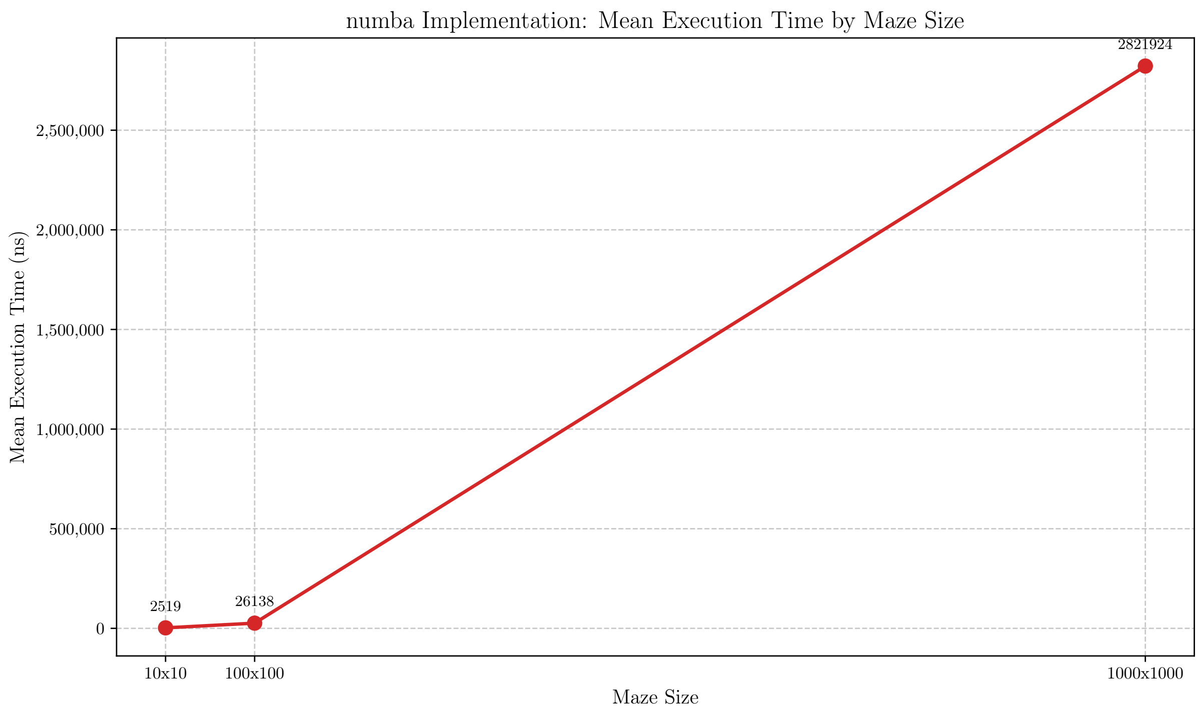

Figure 8: Mean execution Time of the `numba` Implementation Across Maze Sizes

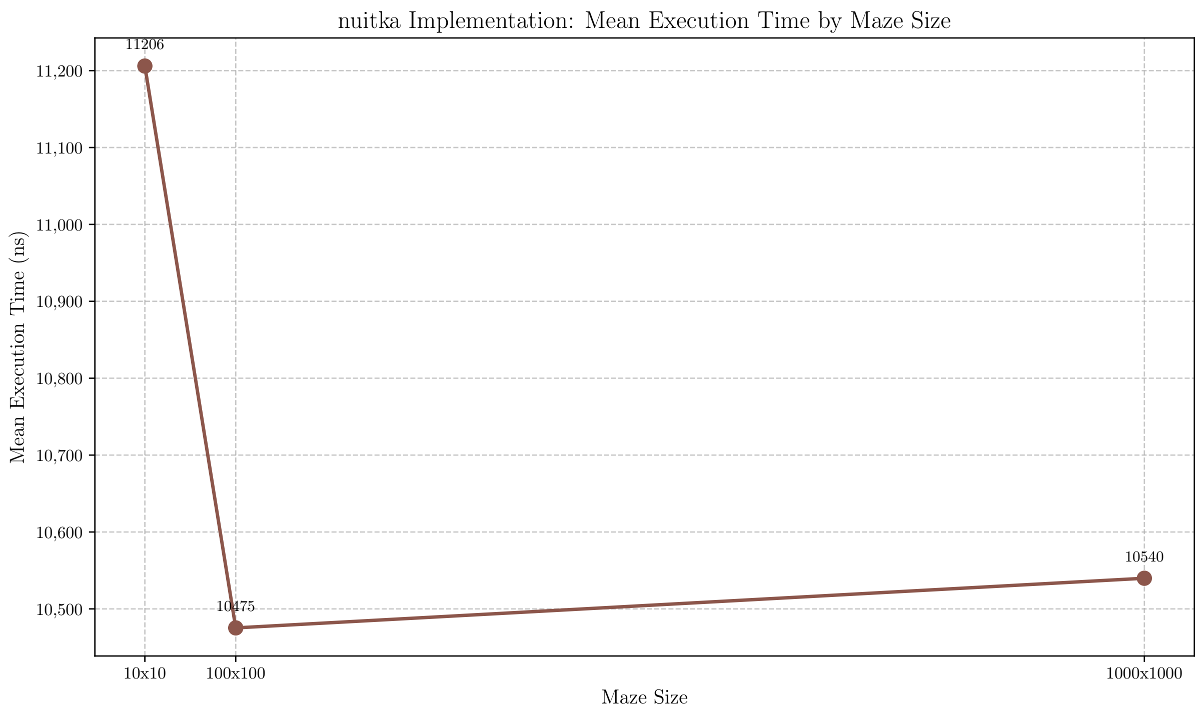

Figure 9: Mean execution Time of the `nuitka` implemntation Across Maze Sizes

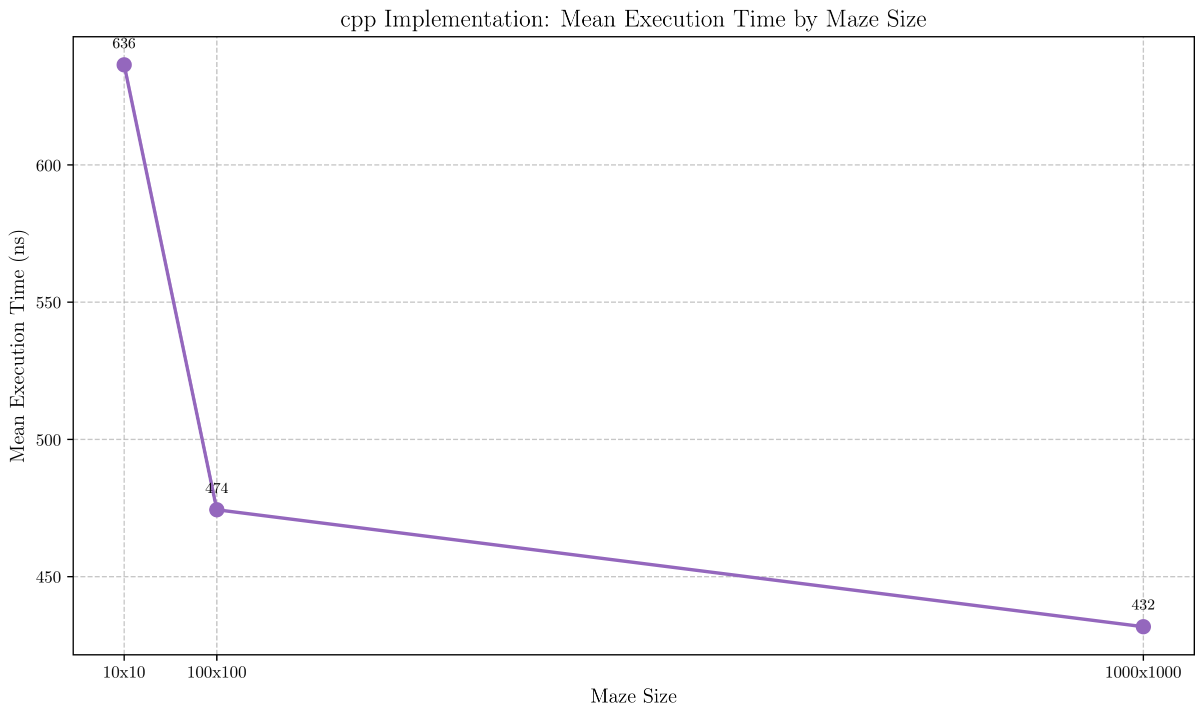

Figure 10: Mean execution Time of the C++ Implementation Across Maze Sizes

```python
from pathlib import Path
import numpy as np
from stara_maze_generator.vmaze import VMaze
from stara_maze_generator.pathfinder import Pathfinder

# Create a 20x20 maze
maze = VMaze(
    seed=42,                # Random seed for reproducibility
    size=20,                # Creates a 20x20 grid
    start=(1, 1),           # Starting position
    goal=(18, 18),          # Goal position
    min_valid_paths=3       # Minimum number of valid paths
)

# Generate the maze structure
maze.generate_maze(pathfinding_algorithm=Pathfinder.BFS)

# Find a path from start to goal
path = maze.find_path()

# Export as HTML visualization
maze.export_html(Path("maze.html"), draw_solution=True)
```

Listing 5: Maze Objects and generator in Python.[28]