A-Maze-ing

This is the way

Summary: Create your own maze generator and display its result!

Version: 1.3

Contents

Chapter I

Forewords

Mazes have fascinated humans for thousands of years. From the legendary **Labyrinth of Knossos** in Greek mythology — built by Daedalus to imprison the Minotaur — to modern puzzle books and video games, mazes have always symbolized mystery, challenge, and clever design. In computer science, maze generation is more than just fun: it's a practical application of algorithms, randomness, and graph theory. Some famous algorithms used for maze generation — like *Prim*'s, *Kruskal*'s, or the *recursive backtracker* — are also used in real-world problems like network design or procedural content generation. Interestingly, perfect mazes (with one unique path between any two points) are directly related to *spanning trees* in graph theory. Building a maze, especially one you can visualize and share, is a great way to explore how computers can create structure from chaos — and have a bit of fun while doing it.

"A labyrinth is not a place to be lost, but a path to be found."

— Anonymous

Chapter II

AI Instructions

● Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

● Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

● Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Common Instructions

III.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- Your functions should handle exceptions gracefully to avoid crashes. Use `try-except` blocks to manage potential errors. Prefer context managers for resources like files or connections to ensure automatic cleanup. If your program crashes due to unhandled exceptions during the review, it will be considered non-functional.
- All resources (e.g., file handles, network connections) must be properly managed to prevent leaks. Use context managers where possible for automatic handling.
- Your code must include type hints for function parameters, return types, and variables where applicable (using the `typing` module). Use `mypy` for static type checking. All functions must pass mypy without errors.
- Include docstrings in functions and classes following PEP 257 (e.g., Google or NumPy style) to document purpose, parameters, and returns.

III.2 Makefile

Include a `Makefile` in your project to automate common tasks. It must contain the following rules (mandatory lint implies the specified flags; it is strongly recommended to try `-strict` for enhanced checking):

- **install**: Install project dependencies using `pip`, `uv`, `pipx`, or any other package manager of your choice.
- **run**: Execute the main script of your project (e.g., via Python interpreter).
- **debug**: Run the main script in debug mode using Python's built-in debugger (e.g., pdb).
- **clean**: Remove temporary files or caches (e.g., __pycache__, .mypy_cache) to keep the project environment clean.

- **lint**: Execute the commands `flake8 .` and `mypy . --warn-return-any --warn-unused-ignores --ignore-missing-imports --disallow-untyped-defs --check-untyped-defs`
- **lint-strict** (optional): Execute the commands `flake8 .` and `mypy . --strict`

III.3 Additional Guidelines

- Create test programs to verify project functionality (not submitted or graded). Use frameworks like `pytest` or `unittest` for unit tests, covering edge cases.
- Include a `.gitignore` file to exclude Python artifacts.
- It is recommended to use virtual environments (e.g., venv or conda) for dependency isolation during development.

If any additional project-specific requirements apply, they will be stated immediately below this section.

Chapter IV

Mandatory part

IV.1 Summary

You will implement a maze generator in Python that takes a configuration file, generates a maze, possibly perfect (with a single path between entrance and exit), and writes it to a file using a hexadecimal wall representation. You will also provide a visual representation of the maze and organize your code so that the generation logic can be reused later.

IV.2 Usage

Your program must be run with the following command:

```
python3 a_maze_ing.py config.txt
```

- `a_maze_ing.py` is your main program file. You must use this name.
- `config.txt` is the only argument. It is a plain text file that defines the maze generation options. You can use a different filename.

Your program must handle all errors gracefully: invalid configuration, file not found, bad syntax, impossible maze parameters, etc. It must never crash unexpectedly, and must always provide a clear error message to the user.

IV.3 Configuration file format

The configuration file must contain one ‘KEY=VALUE‘ pair per line.

Lines starting with `#` are comments and must be ignored.
The following keys are **mandatory**:

Key	Description	Example
`WIDTH`	Maze width (number of cells)	`WIDTH=20`
`HEIGHT`	Maze height	`HEIGHT=15`
`ENTRY`	Entry coordinates (x,y)	`ENTRY=0,0`
`EXIT`	Exit coordinates (x,y)	`EXIT=19,14`
`OUTPUT_FILE`	Output filename	`OUTPUT_FILE=maze.txt`
`PERFECT`	Is the maze perfect?	`PERFECT=True`

You may add additional keys (e.g., seed, algorithm, display mode) if useful.

A default configuration file must be available in your Git repository.

IV.4 Maze Requirements

- The maze must be randomly generated, but reproducibility via a seed is required.
- Each cell of the maze has between 0 and 4 walls, at each cardinal point (North, Est, South, West).
- The maze must be valid, meaning:
  - Entry and exit exist and are different, inside the maze bounds.
  - The structure ensures full connectivity and no isolated cells (except the '42' pattern, see below).
  - As entry and exist are specific cells, there must be walls at the external borders.
  - Your generated data must be coherent: each neighbouring cell must have the same wall if any. E.g., it is forbidden to have a first cell with a wall on the east side, and the second cell behind that wall without a wall on the west side.
- The maze can't have large open areas. Corridors can't be wider than 2 cells.
  For example, you can have 2x3 or 3x2 open area, but never a 3x3 open area.
- When visually represented (see below), the maze must contain a visible "42" drawn by several fully closed cells.
- If the `PERFECT` flag is activated, the maze must contain **exactly one path** between the entry and the exit (i.e., it must be a perfect maze).

```
The "42" pattern may be omitted in case the maze size does not allow
it (i.e.  too small).  Print an error message on the console in that
case.
```

IV.5 Output File Format

The maze must be written in the output file using one hexadecimal digit per cell, where each digit encodes which walls are closed:

Bit	Direction
0 (LSB)	North
1	East
2	South
3	West

- A wall being closed sets the bit to `1`, open means `0`.
  Example: `3` (binary `0011`) means walls are open to the **south** and **west**. Or `A` (binary `1010`) means that **east** and **west** walls are closed.

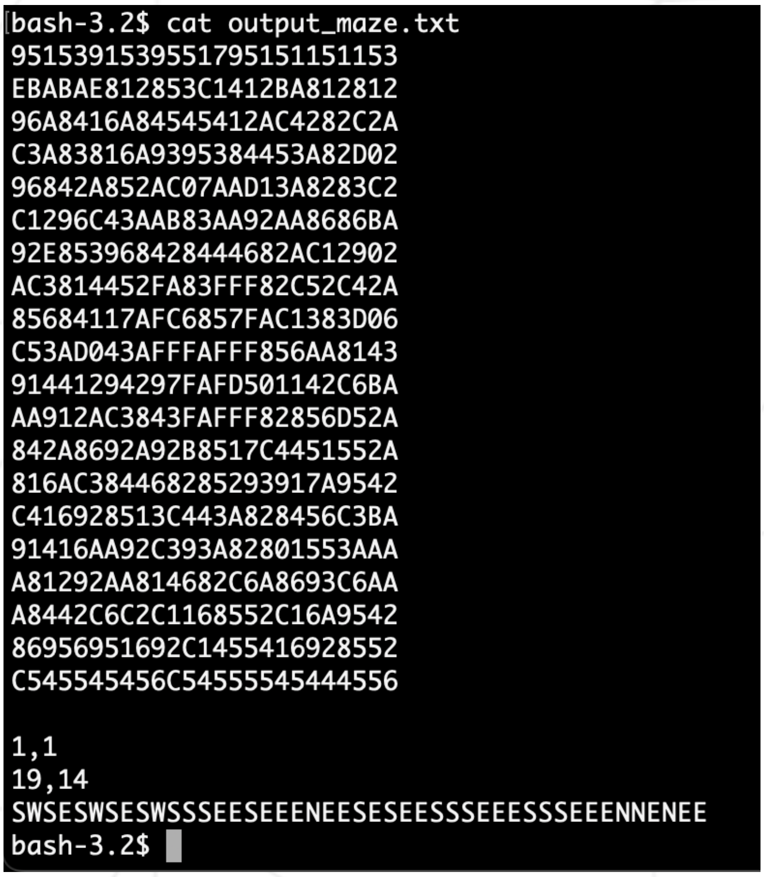
- Cells are stored row by row, one row per line.

- After an empty line, the following 3 elements are inserted in the output file on 3 lines:

  - the entry coordinates, the exit coordinates, and the shortest valid path from entry to exit, using the four letters `N` , `E` , `S` , `W` .

- All lines end with a `\n` .

In conjunction with its specific configuration file, this output file could be tested automatically by a Moulinette. Also, a validation script is provided with this subject to control that the output file contains coherent data.



```
bash-3.2$ cat output_maze.txt
9515391539551795151151153
EBABAE812853C1412BA812812
96A8416A84545412AC4282C2A
C3A83816A9395384453A82D02
96842A852AC07AAD13A8283C2
C1296C43AAB83AA92AA8686BA
92E853968428444682AC12902
AC3814452FA83FFF82C52C42A
85684117AFC6857FAC1383D06
C53AD043AFFFAFFF856AA8143
91441294297FAFD501142C6BA
AA912AC3843FAFFF82856D52A
842A8692A92B8517C4451552A
816AC384468285293917A9542
C416928513C443A828456C3BA
91416AA92C393A82801553AAA
A81292AA814682C6A8693C6AA
A8442C6C2C1168552C16A9542
86956951692C1455416928552
C545545456C54555545444556

1,1
19,14
SWSESWSESWSSSEESEEENEESESEESSSEEESSSEEENNENEE
bash-3.2$
```

Output file example

Chapter V

Visual representation

Your program must provide a way to display the maze visually, using either:

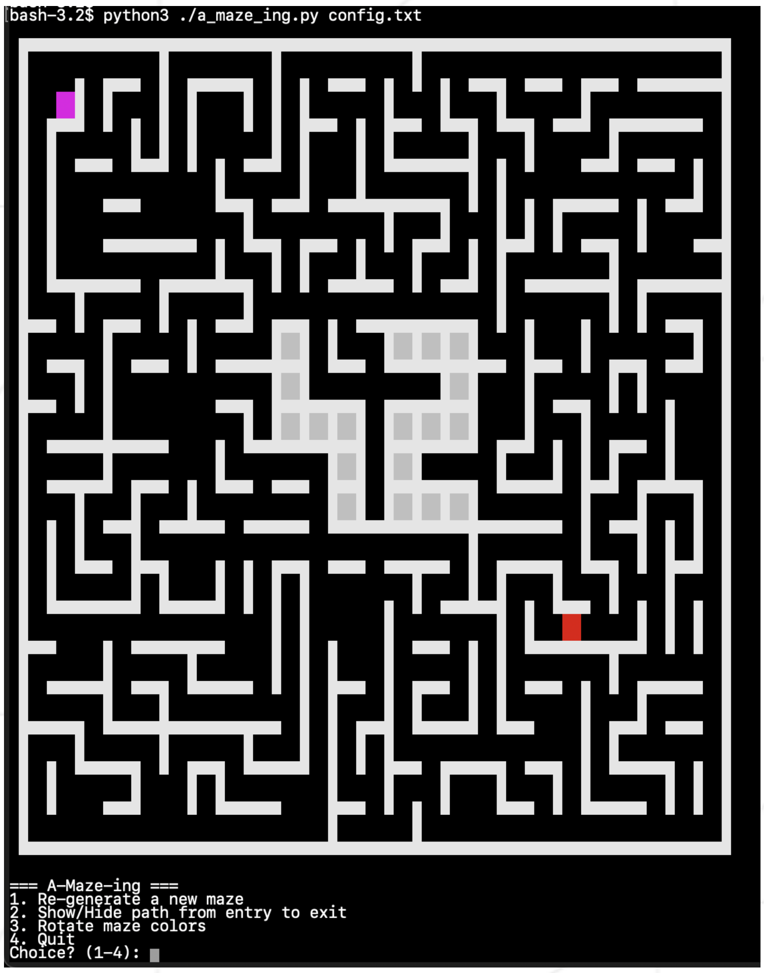
- Terminal ASCII rendering, or
- A graphical display using the MiniLibX (MLX) library.

The visual should clearly show walls, entry, exit, and the solution path.

User interactions must be available, at least for the following tasks:

- Re-generate a new maze and display it.
- Show/Hide a valid shortest path from the entrance to the exit.
- Change maze wall colours.
- Optional: set specific colours to display the “42” pattern.

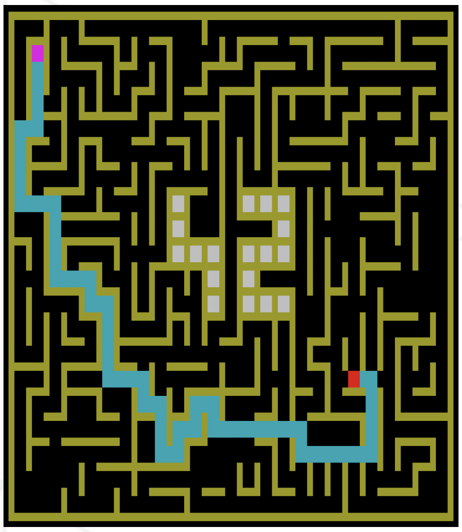
You can add extra user interactions.



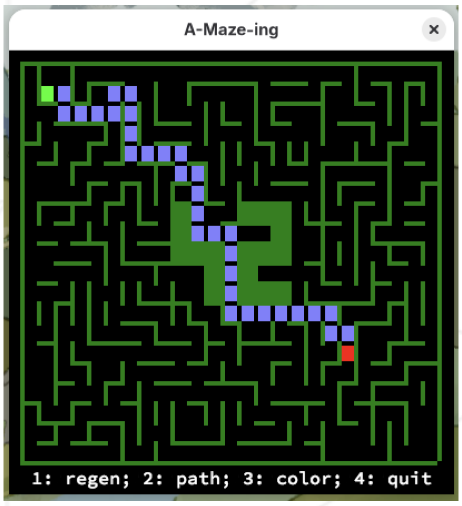
```
bash-3.2$ python3 ./a_maze_ing.py config.txt
```

```
=== A-Maze-ing ===
1. Re-generate a new maze
2. Show/Hide path from entry to exit
3. Rotate maze colors
4. Quit
Choice? (1-4):
```

Terminal default rendering of the maze



Different maze, shortest path and wall colours



Maze rendering using Mlx

Chapter VI

Code reusability requirements

You must implement the maze generation as a unique class (e.g., ‘MazeGenerator‘) inside a standalone module that can be imported in a future project.

You must provide a short documentation describing how to:

- Instantiate and use your generator, with at least a basic example.
- Pass custom parameters (e.g., size, seed).
- Access the generated structure, and access at least a solution.

`The maze generator module grants access to the maze structure, but it is not necessarily the same format as the output file.`

This entire reusable module (code and documentation) must be available in a single file suitable for a later installation by `pip`.
This package must be called `mazegen-*` and the file must be located at the root of your git repository.
Both `.tar.gz` and `.whl` extensions are allowed, as generated by the standard build of a Python package.
Example of a full filename: `mazegen-1.0.0-py3-none-any.whl` .

You must provide in you Git repository all needed elements to build the package. This will be asked during the evaluation: in a virtualenv or equivalent, install the needed tools and build your package again from your sources.

The main `README.md` file (not part of the reusable module) must also contain this short documentation.

Chapter VII

Readme Requirements

A README.md file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The README.md must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the project, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

➠ **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

- The complete structure and format of your config file.
- The maze generation algorithm you chose.
- Why you chose this algorithm.
- What part of your code is reusable, and how.
- Your team and project management with:
  - The roles of each team member.

- Your anticipated planning and how it evolved until the end
- What worked well and what could be improved
- Have you used any specific tools? Which ones?

If you implement advanced features (multiple algorithms, display options), describe them in this `README.md` file.

English is recommended; alternatively, you may use the main language of your campus.

Chapter VIII

Bonuses

You may add various bonuses to your project. Here are possible examples:

- Support multiple maze generation algorithms.
- Add animation during maze generation.

Chapter IX

Submission and peer-evaluation

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behaviour change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific time frame is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.