MazeMastery – A Python Framework for Teaching Maze-Traversal in High School

Raphaël Baur[2], Jens Hartmann[1], and Jacqueline Staub[1]

[1] Faculty 4, Computer Science, University of Trier
Behringstraße 21, 54296 Trier, Germany
[2] Department of Computer Science, ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
rabaur@student.ethz.ch
{s4jehart, staub}@uni-trier.de

Abstract Programming is an activity that is strongly based on abstraction as many solutions can be generalized to cover a wide range of applications. For students who are still in the process of developing their abstraction skills, learning to write code for the general case can be a daunting experience. We present the Python framework MazeMastery which offers a didactic tool for teaching graph exploration strategies through maze-based challenges at high school. Students can verify their algorithms against randomized test cases that currently span six levels of complexity as maze structures continuously increase in their structural complexity. The tool offers an adaptable platform for examining students' learning while challenging their conceptual understanding. MazeMastery is an open-source community project for scientists and educators.

1 Graph Theory and the Long Way to High School

School systems around the globe are slowly adapting to the changing demands in the job market by enforcing algorithmic problem-solving competencies in their school syllabi. The UK, a country once said to be lagging behind Germany in terms of CS education [15], has swiftly reacted by introducing computer science as a compulsory subject across all grades of schooling. Meanwhile, most districts of Germany still teach computer science as an optional subject which is offered at the earliest from lower secondary school [14].

Addressing algorithmic concepts late requires that the focus is all the clearer. The central objective of computer science in school is to foster computational thinking skills. This term summarizes a variety of skills that encompass problem decomposition, pattern recognition, generalization, and abstraction, alongside learning to develop algorithms and formalizing them by programming. A wealth of problems can be tackled using these skills, e.g., from modeling problems using the formal notations of mathematics all the way to advanced programming concepts such as recursion.

One domain that has already been shown to be accessible to students of all ages is graph theory [7,16]. As a fundamental and powerful data structure, graphs

J.-P. Pellet and G. Parriaux (Eds.): ISSEP 2023 Local Proceedings, pp. 75–87, 2023.
https://doi.org/10.5281/zenodo.8431922

can be used for modeling and analyzing multi-entity relationships, such as social networks or transport systems. The corresponding algorithms are universally applicable to all graphs including grid graphs and mazes. Since the 1980s, graph theory has been suggested as a topic for computer science education in Germany [2] and has not disappeared since. In dealing with graph problems, students need to think systematically, experiment, and test their strategies.

Programmers typically explore a range of possible solutions – rarely is there only one correct solution for a given problem. Depending on their skill level, learners may find solutions between two ends of a spectrum: at one end they come up with conceptually simple solutions that are highly specialized to a specific problem instance, while at the other end, students find generalized and complex solutions for larger problem classes. The journey along this spectrum can be modeled as an adapted semantic wave. Maton's *semantic wave theory* [18] distinguishes between semantic density/gravity to describe the different levels of human ingenuity. *Semantic density* conceptualizes the degree of condensation of meaning (perceived as complexity). Basic programming commands, for instance, contain less semantic meaning (i.e., they are semantically less "dense") than the concept of recursion. In contrast, *semantic gravity* conceptualizes the degree of abstraction of meaning (perceived as abstraction). Tailored solutions to a given problem are less abstract (and thus contain less semantic gravity) than fully generalized solutions. Teachers can model a student's learning path via their teaching materials according to the concept of a semantic wave.

Existing materials are available for generating and visualizing mazes [12], for illustrating search algorithms [3,5,17,1], for interacting with mazes in mixed reality [9], using robots [6] or via classical programming [4,13]. While these existing projects feature various maze-related topics in isolation, we present a tool that combines all parts in a single didactic framework for Python programming classes. Moreover, we incorporate techniques from the semantic wave theory into a single open-source tool. The following section describes how this tool works and how it can be used for modeling mazes as graphs and for subsequently traversing a maze in a visual environment.

2 A Framework for Exploring DFS via Maze Traversal

Graph traversal algorithms such as depth-first-search (DFS) are topics in computer science programming curricula at high school. To implement a DFS graph traversal algorithm from scratch, several preconditions must be met: Students require familiarity with both non-linear data structures like graphs for modeling purposes and with programming constructs such as sequences, loops, variables, branching, lists, matrices, stacks, and recursion.

In the following, we present a didactic tool for teaching *maze traversal* in Python based on the above-mentioned prerequisites. We first discuss how mazes can be modeled as graphs and then dive into the technical details by presenting a tool that generates custom maze instances to challenge students'

conceptual understanding. Finally, we discuss the generation of mazes with dedicated attributes.

2.1 Modeling Mazes as Graphs

Students encounter mazes in their everyday lives, from architecture to entertainment; it is thus an intuitively known object of study that can be modeled as a graph. Specifically, grid mazes can be represented as planar graphs whose nodes correspond to individual maze cells. A cell permits access to its spatially adjacent cells if and only if there is no wall between them. We call such cells *neighbors*. Three features characterize grid mazes:

1. There is a distinctive *start* and *end point*. These nodes can be marked, e.g., using dedicated markers. In our case, the start point corresponds to the initial position of the agent, whereas the endpoint is the location of the Minotaur.
2. A grid maze consists of individual cells arranged in a *grid structure*. Each cell has at most four neighbors; one in each compass direction.
3. Whether or not two adjacent grid cells are neighbors (which allow passing through in both directions) is determined by the presence or absence of separating walls.

Consider a rectangular maze of m rows and n columns exhibiting the attributes above. A natural approach to model such a maze as a graph consists in representing each cell as a node, arranging them in a grid of the same dimension as the maze, and connecting two adjacent nodes by an edge if their corresponding maze cells have no walls in between. With this representation, we obtain a grid graph of size $m \times n$. Each node has a unique coordinate (i, j) with $i \in \{0, 1, \ldots, m-1\}$ and $j \in \{0, 1, \ldots, n-1\}$ that can be used as a two-tuple to determine the cell. Figure 1 illustrates such a maze of size 4×5 alongside the corresponding grid graph. In this case, maze traversal starts on the cell with coordinates $(0, 0)$, and the target is on coordinates $(2, 3)$.

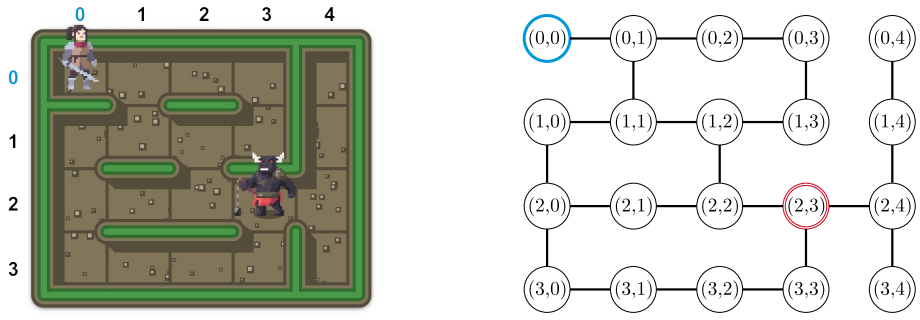

Figure 1: The same 4×5 grid structure once represented as a maze and once as a graph

Note that this naming convention for nodes not only allows for assigning each cell a unique name, but also allows for deriving from a coordinate (i, j) its adjacent cells to the north, south, west, and east ($(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$ respectively). This implicit information is the first objective that students get to explore in our task series.

Although nodes in grid graphs typically have 4 neighbors (ignoring nodes at the maze border), this is generally not the case in grid graphs representing mazes. In grid graphs for mazes, nodes tend to have fewer neighbors with as few as 0 to 4 edges per node. For example, in Figure 1, cell (1,3) is connected only to (0,3) and (1,2). Although (2,3) and (1,4) are also spatially adjacent, they do not allow direct access due to the prevailing topology with walls. It is thus impossible to directly visit the Minotaur on cell (2,3) from (1,3).

2.2 MazeMastery – A Tool for Teaching Maze Traversal

We developed a Python library named *MazeMastery* to facilitate students' explorative, programmatic, and in-depth exposure to the graph model presented in the previous subsection. We provide MazeMastery as a stand-alone Python package, which allows it to be easily integrated into any development environment supporting a Python interpreter. We provide an open-source implementation at GitHub, and the package can be installed using the `pip` package manager using `pip install mazemastery`.

The library provides a student-friendly user interface created with `tkinter` that offers randomized exercises and tests for a diverse learning experience. A minimal vocabulary of six basic commands, shown in Figure 2, is sufficient in combination with classic Python constructs to use the library in high school programming classes.

Students take control of an agent and are challenged to guide it from the starting point to the end point (where the Minotaur awaits, drawing inspiration from mythological narratives). Commands like `get_pos()` and `set_pos(c)` are provided to determine the agent's position and facilitate its traversal to neighboring cells. Using `put_[blue|red]_gem()` and `has_[blue|red]_gem(c)`, markers can be positioned on the current cell for state management, and by utilizing `has_minotaur()` and `get_neighbors()`, students gain the means to examine the local surroundings of the agent, even when dealing with randomized and unknown grids. MazeMastery uses a global Cartesian coordinate system where the agent is, however, only able to move within his local neighborhood.

2.3 Level Description

MazeMastery currently provides six progressive levels, each curated to systematically increase the complexity and abstraction of the algorithmic implementation required. Students have the flexibility to approach the levels with whatever methods seem suitable. Each level comprises mazes with a specific characteristic that poses requirements on the generalizability of the students' solutions.

`get_pos()`	
Description	Return current position
Parameters	–
Returns	Integer tuple `(i,j)`, corresponding to the agent's current position

`set_pos(coordinates)`	
Description	Move to `coordinates` if neighboring
Parameters	Integer tuple `(i,j)`
Returns	`None`

`put_[blue|red]_gem()`	
Description	Mark current position with a blue (red) gem, overwriting previous gems
Parameters	–
Returns	`None`

`has_[blue|red]_gem(coordinates)`	
Description	Check if the `coordinates` are marked with a blue (red) gem
Parameters	Integer tuple `(i,j)`
Returns	`True`, if the node is marked with a blue (red) gem, `False` otherwise

`has_minotaur()`	
Description	Check whether the Minotaur is at the agent's current position
Parameters	–
Returns	`True`, if the Minotaur is at current position, `False` otherwise

`get_neighbors()`	
Description	Return list of nodes neighboring to the agent's current position
Parameters	–
Returns	List of integer tuples corresponding to neighboring nodes

Figure 2: Basic Commands used within MazeMastery

An overview of all levels along with an exemplary and modular codebase showcasing key insights, can be found in Figure 10. These insights serve as guiding principles, regardless of the specific implementation chosen by students. We designed levels such that students can build upon their previous code and add new functionality as the complexity of the mazes increases, thus increasing semantic density and decreasing semantic gravity over time.

Level 1 introduces the agent starting on the left of an extended pathway with the Minotaur at the wall to the right (see Figure 3). Navigating through this maze requires students to visit several consecutive nodes. Students can achieve this using a single coordinate instance whose column index is continuously updated and used in the `set_pos(c)` command.

Level 2 shifts the positioning of the Minotaur at the beginning of each level (see Figure 4). Without adapting the solution found for level 1, the agent likely steps past the Minotaur due to this change. Consequently, the search should terminate upon encountering the Minotaur. To this end, our library exposes the `has_minotaur()` function, which checks whether the Minotaur is located at the agent's coordinate.

Level 3 deviates from the linear progression of the previous two levels by presenting an *unicursal* maze structure that does not contain any dead ends or loops, i.e., each node still has only two neighboring nodes but is not arranged in a straight line (see Figure 5). In previous levels, it was sufficient to only adjust the

column index of the agent's position, but now this method will prove ineffective if the path changes direction. Keeping track of where the agent is coming from and where it is going to be is the core challenge of this stage. The users will most likely find that they need to mark previously visited nodes, which can be achieved using the `put_blue_gem()` command, which places a blue jewel on the agent's coordinates, and the `has_blue_gem(c)` command, which enables users to check if node `c` was previously marked.

Level 4 presents a *perfect* maze structure exclusively comprising dead ends, devoid of any cycles (see Figure 6). It introduces nodes with more than two neighbors, thus confronting students with the challenge to effectively address dead ends and navigate despite them to advance further in the maze. A possible strategy involves the utilization of blue gems to mark the nodes that have been visited once and red gems, which can be placed and checked for with the commands `put_red_gem()` and `has_red_gem(c)` respectively, for nodes that have been visited twice (i.e., on the return path from a dead end). When students find themselves at a dead end, they need to employ a mechanism to exclude nodes already marked from their path options.

Level 5 presents *multiply connected* mazes with both dead ends and cycles (see Figure 7). To address this challenge, students need to recognize the importance of making previously visited nodes retrievable, a requirement not present in previous levels. In this context, students are encouraged to contemplate adopting either an iterative or recursive approach. In the iterative method, students employ a stack data structure, which allows them to store the visited nodes in a last-in-first-out manner. As the agent naturally progresses through the maze, visited nodes can be pushed onto the stack, facilitating easy retrieval when backtracking is necessary.

Level 6 serves as the most advanced stage in the current structure. After locating the Minotaur, the students' objective changes to guiding the agent back to the entrance (Figure 8). This can be achieved via recursion; the recursive descent enables traversal into the maze, while the recursive ascent allows for an effective return path. Level 6 serves as a culmination of the student's learning journey, consolidating their understanding of recursive algorithms and providing a platform to showcase algorithmic skills in the context of maze traversal.

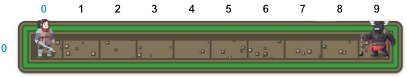

Figure 3: Sample Maze for Level 1

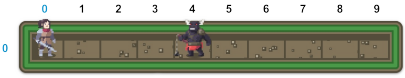

Figure 4: Sample Maze for Level 2

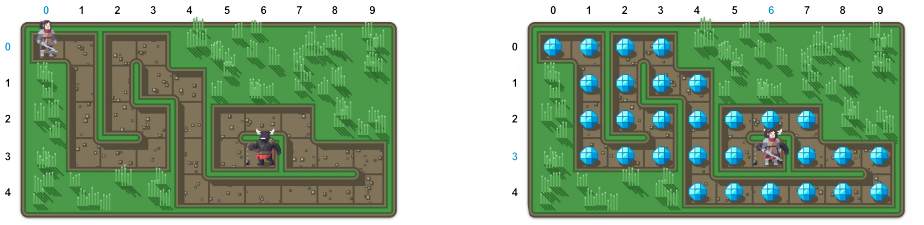

Figure 5: Unicursal Maze with One Single Path From Start to End (Level 3)

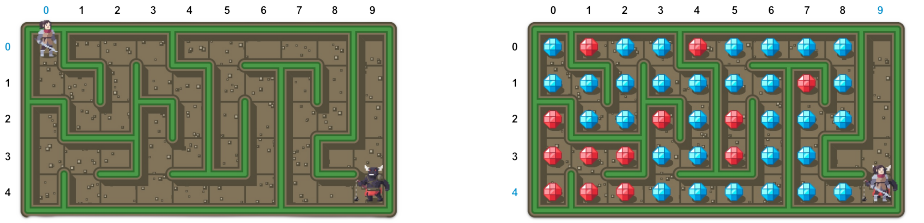

Figure 6: Perfect Maze without Loops but with Dead Ends (Level 4)

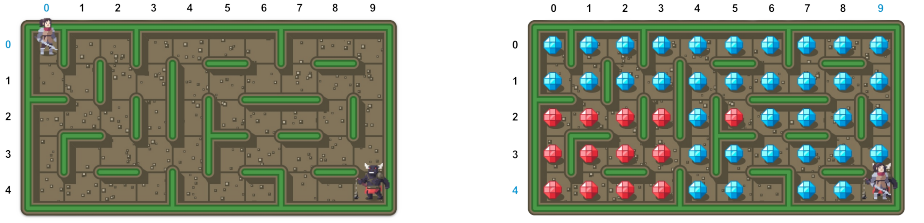

Figure 7: Multiply connected Maze with Loops and Dead Ends (Level 5)

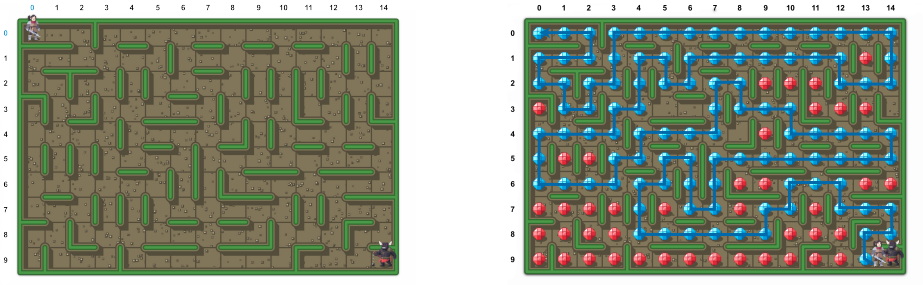

Figure 8: Multiply connected Maze with Backtracking (Level 6)

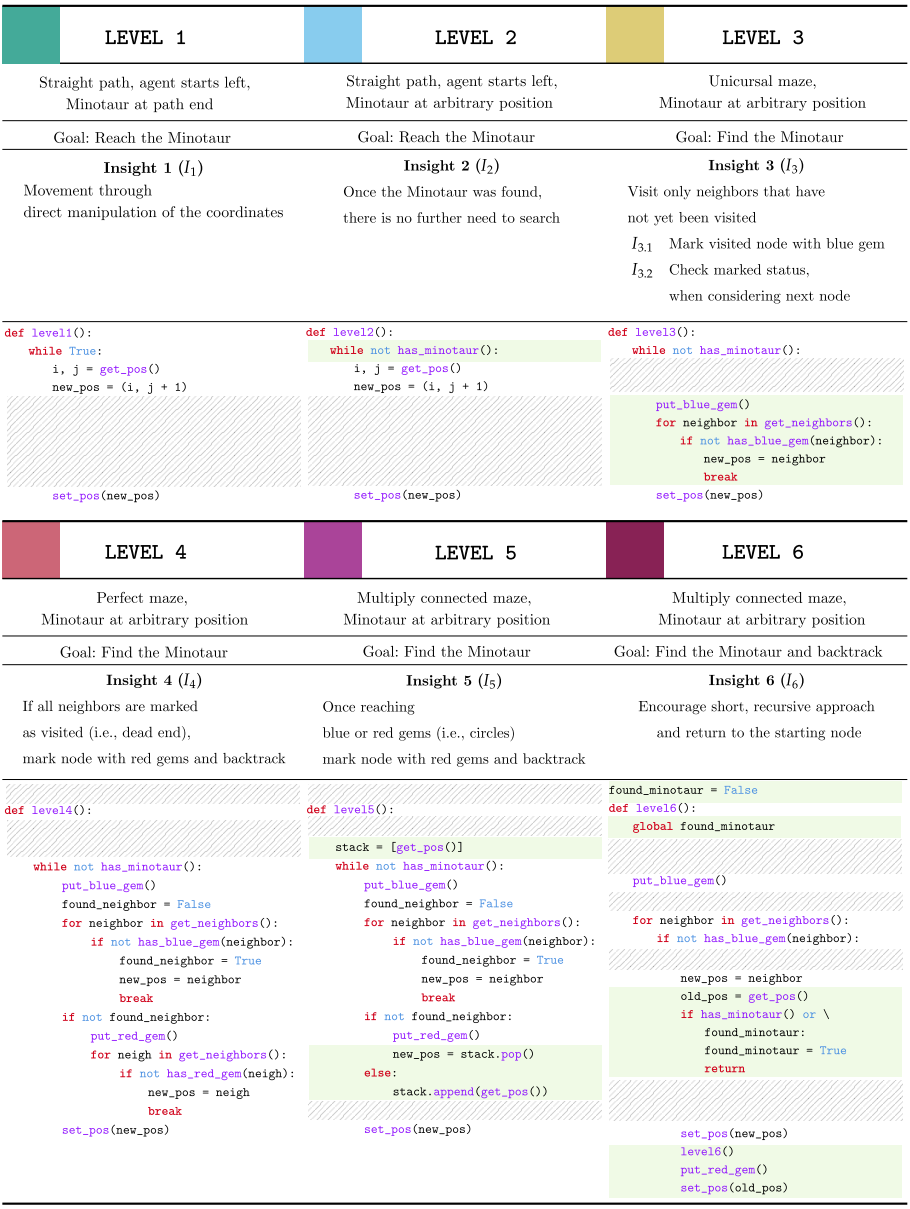

Figure 9: Level structure with possible implementation

2.4 Our Contribution to Teaching Recursion

The outlined level hierarchy is designed to provide students with a structured and progressive introduction to algorithmic concepts related to maze traversal. The challenges emphasize the importance of iterative and recursive methods, conditional statements, and data structures for graph traversal. While some of these concepts can be challenging to teach, recursion is known as a concept that is especially hard to grasp [10,19] and for which several incorrect mental models have been found [8,11]. We address two such flawed mental models:

1. The *looping model* manifests as an erroneous identification of recursion and loops, where the recursive process is perceived as a unified entity rather than a sequence of successive instantiations. This misinterpretation blurs the distinction between recursion and traditional looping constructs. It is crucial to note that until level 4, the maze challenges can indeed be effectively solved using loops. However, although loops can be replaced with tail recursion, this substitution is not universally applicable to all forms of recursion. This limitation becomes evident in levels 5 and 6, where more complex tasks such as identifying cycles and dead ends demand a more profound understanding of the backtracking process. As students grapple with the intricacies of backtracking in these scenarios, they might recognize the constraints inherent in relying solely on an iterative approach, especially without an explicit stack.
2. The *recursive descent model* manifests as an initial assumption that recursion terminates solely upon reaching the base case. This model is challenged in levels 5 and 6 as students realize the necessity of the recursive ascent to continue the traversal process by backtracking and overcoming impediments. They realize that instructions following the recursive call are required to return to the correct path using different-colored node markings.

Students who develop mental models that do not fully capture the essence of recursion might potentially encounter challenges in their conceptual understanding, which could affect their problem-solving abilities. Addressing these misconceptions is, therefore, a pertinent aspect of teaching programming. Our library addresses this point using tailor-made maze instances for each of the six levels.

2.5 Maze Generation

To accommodate different levels of complexity, MazeMastery generates maze instances with specific topological and geometrical properties. We discuss how mazes for each level are generated:

- Levels 1 and 2 present mazes that represent straight corridors that only differ in the positioning of the Minotaur. The corresponding graphs consist of a single chain of nodes that differ only by their column coordinates.
- Level 3 involves *unicursal* mazes constituted by a single corridor with randomized turns. This type of maze is generated by sampling self-avoiding walks, i.e., from an initial node, neighbors are added by sampling randomly

until the path self-intersects. To prevent early intersections, we use a heuristic by increasing the probability of choosing neighbors in less-traversed directions. For each direction, we count the number of nodes that lie opposite the direction at hand, yielding values z_d where d indicates the direction *up*, *down*, *left*, and *right*. To compute the probability of selecting the neighbor in a specific direction, we use a *tempered softmax* function, defined as $\exp(-\beta z_d)/\sum_{d'} \exp(-\beta z_{d'})$. The tempered softmax serves two purposes. First, it compresses the counts we obtain into a range between 0 and 1. This compression allows us to interpret the results as probabilities. Second, it ensures that the sum of these probabilities across all directions equals 1, creating a valid probability distribution. The parameter β influences the shape of the resulting distribution. When β is small, the distribution approaches uniformity, meaning each direction is chosen with roughly equal probability. Larger values of β emphasize the differences between the counts, increasing the chances of exploring less frequently chosen directions. However, deterministically extending the path towards a less-explored direction will not yield trajectories that cover the whole grid, but will tend to the center of the grid and then end due to self-collisions. Choosing $\beta = 0.01$ seems to strike a good balance between uniform distribution, which is prone to early self-intersections, and a strategy that strives towards unexplored areas too greedily, leading to paths that converge to the middle of the maze too quickly.
- Level 4 involves mazes with junctions and dead-ends, but no cycles, so-called *perfect mazes* that correspond to trees. To generate these mazes, we create a spanning tree using randomized depth-first search. In that process, potential neighbors are chosen uniformly at random. To create the final maze, we start with a fully disconnected grid graph and connect two nodes if and only if they are connected in the previously generated spanning tree. With that, we ensure that each cell is reachable and no loops exist, following the properties of a spanning tree.
- Levels 5 and 6 involve mazes with dead ends and loops. We first generate the maze analogously to level 4. Then, we remove walls with probability p if their removal does not create 2×2 grids within the maze that do not contain a wall, retaining the maze structure while avoiding large areas with no walls.

3 Evaluation

We conducted a qualitative think-aloud study with five male participants aged 19 to 23 years. They had all programming experience of at least 3 years but varied knowledge of Python syntax.

3.1 Study Setup

The study aimed to assess (i) whether the tool allows heterogeneous learning, and (ii) whether there are specific problems all participants encounter. After a brief introduction to the six commands (see Figure 2), participants started progressing from level 1 to 6 on their own. All actions were recorded for subsequent analysis.

3.2 Findings

A short summary of experiences:

(i) Personalized learning seems possible. Solutions for level 1 sometimes foreshadowed later concepts. Participants progressed at their individual pace, some reaching level 3 in just five, others taking more than 20 minutes.
(i) Six levels seem too few. Intermediate levels between 2 and 3, and between 3 and 4 are advisable for a smoother learning experience. Within 60 minutes, no participant reached beyond level 4, one only reached level 3.
(ii) Confusion arose from associating commands with the agent's actions. Initial parameterized commands like `put_red_gem()` allowing off-agent gem placement caused confusion, leading to the command set presented before.

More detailed information including solutions is provided here.

4 Conclusion

Programming is an activity that requires abstraction as many solutions can be generalized to cover a wide range of applications. For students who are still in the process of developing their abstraction skills, this can be a daunting experience. This work proposes an approach to address this hurdle in the context of graph traversal using two-dimensional mazes. MazeMastery, our Python framework, currently provides six levels of increasing semantic complexity: students first infer global attributes of a given problem class by analyzing concrete instances, they then develop a generalized algorithm and finally verify their algorithm against concrete but unknown test cases. The intended learning path varies in semantic complexity both throughout a specific level but also across all levels.

Ongoing research evaluates the framework's effectiveness in teaching algorithmic problem-solving and analyzing learning paths. The modular and adaptable nature of the framework allows for customization and integration with other teaching resources in the context of programming. Limitations include the frameworks' early development stage which did not yet involve tutorials and a quantitative evaluation. However, we hope that this work inspires the scientific community to become an active partner in researching the topic of graph theory in education and promote future research on or with our open-source Python framework.

Acknowledgment We gratefully acknowledge the generous support of the Carl-Zeiss-Foundation for funding our research. Moreover, we extend our heartfelt appreciation to Dr. Martin Löhnertz for his assistance in proofreading and providing input to this work.

References

1. Alan Blair, David Collien, Dwayne Ripley, and Selena Griffith. *Constructivist Simulations for Path Search Algorithms*, pages 990–998. Australasian Association for Engineering Education, Sydney, 2017.

2. R. Bodendiek, H. Schumacher, K. Wagner, and G. Walther. *Graphen in Forschung und Unterricht: Festschrift K. Wagner*. B. Franzbecker, 1985.
3. Alberte Emilie Christensen, Cecilie Jegind Christensen, Johannes Louis Geishauser Hald, and Nicolai Otto. Generating and solving mazes, 2019.
4. Xiaozhou Deng, Danli Wang, Qiao Jin, and Fang Sun. Arcat: A tangible programming tool for dfs algorithm teaching. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children*, IDC '19, page 533–537, New York, NY, USA, 2019. Association for Computing Machinery.
5. Clemens Bandrock et al. Projekt 4d labyrinth. https://www.mintgruen.tu-berlin.de/mathesisWiki/doku.php?id=ss2021:project6:4d_labyrinth retrieved 23-4-10, 2021.
6. Sergey Filippov, Natalia Ten, Ilya Shirokolobov, and Alexander Fradkov. Teaching robotics in secondary school. *IFAC-PapersOnLine*, 50(1):12155–12160, 2017.
7. J. Paul Gibson. Teaching graph algorithms to children of all ages. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, pages 34–39, 2012.
8. Tina Götschi, Ian Sanders, and Vashti Galpin. Mental models of recursion. *ACM SIGCSE Bulletin*, 35(1):346–350, 2003.
9. Lorenz Klopfenstein, Saverio Delpriori, Brendan Paolini, and Alessandro Bogliolo. Codymaze: The hour of code in a mixed-reality maze. In *INTED2018 proceedings*, pages 4878–4884. IATED, 2018.
10. Dalit Levy and Tami Lapidot. Recursively speaking: analyzing students' discourse of recursive phenomena. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 315–319, Austin Texas USA, 2000. ACM.
11. Claudio Mirolo. Is iteration really easier to learn than recursion for CS1 students? In *Proceedings of the ninth annual international conference on International computing education research*, ICER '12, pages 99–104, New York, 2012. Association for Computing Machinery.
12. Muhammad Ahsan Naeem. pyamaze. https://github.com/MAN1986/pyamaze retrieved 2023-4-10, 2021.
13. Richard Rasala, Jeff Raab, and Viera K. Proulx. The sigcse 2001 maze demonstration program. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 287–291, 2002.
14. Richard Schwarz, Lutz Hellmig, and Steffen Friedrich. Informatikunterricht in Deutschland – eine Übersicht. *Informatik Spektrum*, 44:95–103, 2021.
15. Sue Sentance and Neena Thota. A comparison of current trends within computer science teaching in school in germany and the uk. In *Informatics in schools: local proceedings of the 6th International Conference ISSEP 2013; selected papers; Oldenburg, Germany, February 26–March 2, 2013*, volume 6, pages 63–75. Universitätsverlag Potsdam, 2013.
16. Robert R. Snapp. Teaching graph algorithms in a corn maze. *ACM SIGCSE Bulletin*, 38(3):347–347, 2006.
17. MakeSchool Team. Solving the maze: Trees and mazes. https://makeschool.org/mediabook/oa/tutorials/trees-and-mazes/solving-the-maze/ retrieved 23-4-10.
18. Jane Waite, Karl Maton, Paul Curzon, and Lucinda Tuttiett. Unplugged computing and semantic waves: Analysing crazy characters. In *Proceedings of the 2019 Conference on United Kingdom & Ireland Computing Education Research*, pages 1–7, 2019.
19. Susan Wiedenbeck. Learning recursion as a concept and as a programming technique. *ACM SIGCSE Bulletin*, 20(1):275–278, 1988.

A Appendix

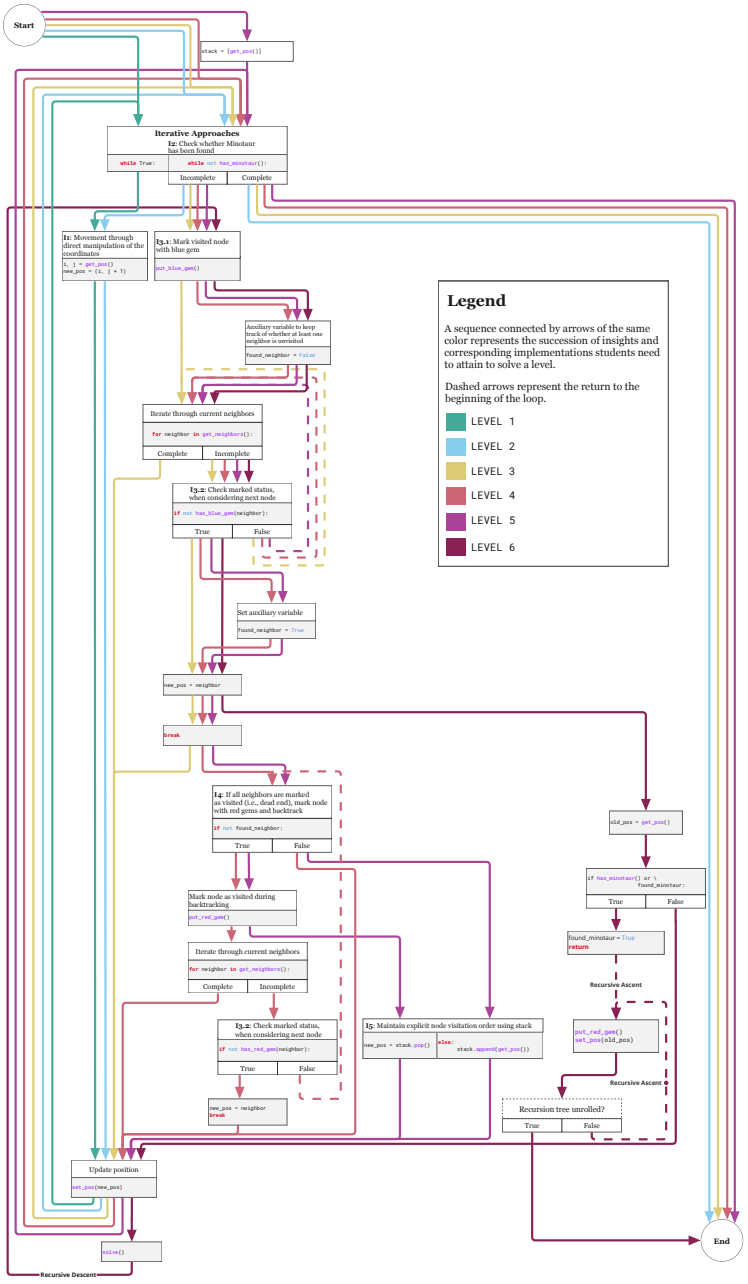

Figure 10: Flowchart illustrating the insights gained in the course of programming