FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Learning Environment for Teaching Maze Algorithms

MASTERARBEIT

zur Erlangung des akademischen Grades

Magister der Sozial- und Wirtschaftswissenschaften

im Rahmen des Studiums

Informatikmanagement

eingereicht von

Dipl.-Ing. Stefan Melbinger
Matrikelnummer 0225692

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek

Wien, 19. November 2015

Stefan Melbinger	Gerald Futschek

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Learning Environment for Teaching Maze Algorithms

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Magister der Sozial- und Wirtschaftswissenschaften

in

Computer Science Management

by

Dipl.-Ing. Stefan Melbinger
Registration Number 0225692

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek

Vienna, 19th November, 2015

Stefan Melbinger	Gerald Futschek

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Stefan Melbinger
Novaragasse 33/1/24, 1020 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. November 2015

Stefan Melbinger

Acknowledgements

I offer my sincerest gratitude to Professor Gerald Futschek for immediately offering me the chance to graduate under his guidance, for being highly supportive and for sharing my enthusiasm for maze solving algorithms.

This thesis is dedicated to my beloved girlfriend Claudia. Thank you for being with me in good times and bad times.

Kurzfassung

Die Lehre algorithmischen Denkens, einer der grundlegenden Fähigkeiten der Informatik, ist eine herausfordernde Aufgabe. Das Propädeutikum für angehende Studierende der Informatik an der Technischen Universität Wien nutzt dazu ein Computerprogramm namens Theseus das die Ausführung von Algorithmen zur Lösung von Labyrinthen darstellt um so die Diskussion von Algorithmen greifbarer zu machen. Theseus beschreibt die ausgeführten Algorithmen allerdings nicht explizit, sodass die Studenten nicht einsehen können was "in" diesen passiert während am Bildschirm die Suche nach dem Zielpunkt eines Labyrinths veranschaulicht wird.

Diese Diplomarbeit untersucht wie eine interaktive Lernumgebung nicht nur die Ausführung sondern auch die Formulierung von Algorithmen ermöglichen kann um so die Lehre algorithmischen Denkens zu unterstützen und den Lerneffekt für Studierende zu erhöhen.

Nach Analyse des didaktischen Handlungsspielraums und der Auswahl eines geeigneten, an der konstruktionistischen Lerntheorie ausgerichteten Vorgehens beschreibt diese Arbeit die konkrete Implementierung einer neuen, online verfügbaren Lernumgebung namens Ariadne welche eine visuelle Programmiersprache zur Beschreibung von Algorithmen integriert. Studenten und Lehrer können damit die von Ariadne vordefinierten Algorithmen zur Lösungssuche in Labyrinthen verstehen, sie verändern und sogar völlig neue Algorithmen entwerfen.

Abstract

Teaching algorithmic thinking, which is one of the fundamental skills in computer science, is a challenging task. Preparatory courses for budding students of computer science at the Vienna University of Technology use a computer program called Theseus to visualise the execution of maze solving algorithms, thus making the discussion of algorithms more tangible. However, Theseus does not explicitly describe the executed algorithms and students are left to wonder what is happening “inside” them while the search for a maze’s finish is visualised on screen.

This thesis investigates how an interactive learning environment can not only execute but also formulate maze algorithms in order to support the teaching of algorithmic thinking and increase the learning effect for students.

After performing an analysis of various didactic options and selecting a suitable approach that is aligned with the constructionist learning theory, the actual implementation of a new online learning environment called Ariadne is described, which integrates a visual programming language for describing algorithms. This allows students and teachers to understand Ariadne’s pre-defined maze solving algorithms, to modify them and to even create new algorithms from scratch.

Contents

CHAPTER 1

Introduction

This chapter introduces the preparatory course held for budding students of computer science at the Vienna University of Technology and the software used to support teaching algorithmic thinking in these lectures. The research question on which this thesis is based is then presented along with a a description of the methodology used to discuss and answer that question. Finaly, a short overview of the structure of this thesis is given.

1.1 Teaching Algorithmic Thinking at the Vienna University of Technology

It is challenging to teach algorithmic thinking to students of computer science of whom "around 70% are intimidated when they take a basic algorithms course" [Kin12]. While all curricula of bachelor programmes of computer science include lectures about algorithms and data structures, the Faculty of Informatics at the Vienna University of Technology has also introduced optional preparatory courses under the name *PROLOG*[1]. This series of lectures, which not only deals with algorithmic thinking but also with, for example, computer hardware and mathematics, is intended to ease the transition into university life for budding students and to convey to them the basic fundamentals of computer science.

The lectures on algorithmic thinking are held by Professor Gerald Futschek who discusses the thought process of solving problems and introduces the semi-formal formulation of algorithms in pseudocode. To illustrate his points and to make the subject matter more tangible, he explores different ways of finding a given point within a maze and uses a program called Theseus to visualise mazes and the execution of maze solving algorithms.

[1] Propädeutikum für Informatik, see `http://www.informatik.tuwien.ac.at/studium/studierende/prolog`

1.2 Research Question

While Theseus visualises the execution of maze solving algorithms, it does not show the algorithms themselves and students are left to wonder what exactly is happening "inside" the algorithm while the maze cells are being highlighted on screen during the algorithm's search for the finish point.

The thesis at hand wants to analyse such shortcomings of Theseus and improve the situation, increasing the learning effect for students and creating new ways for the PROLOG lectures to be supported by software. Thus, the following chapters try to answer the question how an interactive learning environment can describe and execute maze algorithms in order to support the teaching of algorithmic thinking to budding students of computer science.

The proof of concept implementation resulting from this research is called *Ariadne*.

1.3 Methodology

This thesis presents constructionism as a fundamental learning theory that is well suited for selecting a didactic approach guiding the concrete implementation that is also part of this work. After introducing the theory and closely related fields of research (learning environments, visual block programming languages and algorithm animations), an analysis of the didactic options is performed and an approach is selected and justified. The main chapters then give an in-depth description of the implementation of a web application as a high-quality proof of concept.

1.4 Structure of This Thesis

Chapter 2 gives an overview of existing work and research on which the implementation of Ariadne will set up. Vision and requirements for the program are detailed in Chapter 3. Because Ariadne's vision could be implemented in a number of ways, Chapter 4 presents a didactic analysis of the implementation options and selects the most suitable approach.

Preparing the actual implementation, Chapter 5 introduces the basic concepts of maze structures and maze algorithms. Chapter 6 then explains on which level of abstraction these concepts shall be taught and describes the maze solving algorithms that will be pre-defined by Ariadne.

The concrete implementation of Ariadne, including its basic design decisions and relevant details, is presented in Chapter 7. The result of this implementation and whether it fulfils the requirements stated before is then discussed in Chapter 8. This chapter also mentions potential future development options for Ariadne.

Finally, Chapter 9 gives a summary of this thesis.

CHAPTER 2

Related Work

This thesis builds upon decades of research on learning environments and visual programming environments which have often been closely related to each other. This chapter will highlight some of the major advancements of constructionist learning environments and how visual programming relates to them.

Section 2.1 gives a concise introduction to constructionism. As described in Section 2.2, this learning theory is the foundation of many learning environments. Section 2.3 describes how learning environments, especially in the context of computer science education, often integrate visual block programming languages. Execution of algorithms formulated with these languages is then usually visualised and animated, as explained in Section 2.4. Finally, Section 2.5 shows that educational requirements will often lead to the need for creating a custom learning environment using appropriate frameworks.

2.1 Constructionist Learning Theory

Constructionism is a learning theory which states that “human beings come to learn most effectively [by] building a model, reflecting on it, debugging and sharing” [NC15]. It is based on the constructivist theory which states that “learning is an active process of constructing rather than acquiring knowledge” and that “instruction is a process of supporting that construction rather than communicating knowledge” [CD96]. In an ideal constructionist environment, “teachers [...] are called to function as facilitators who coach learners as they blaze their own paths toward personally meaningful goals” [AL02].

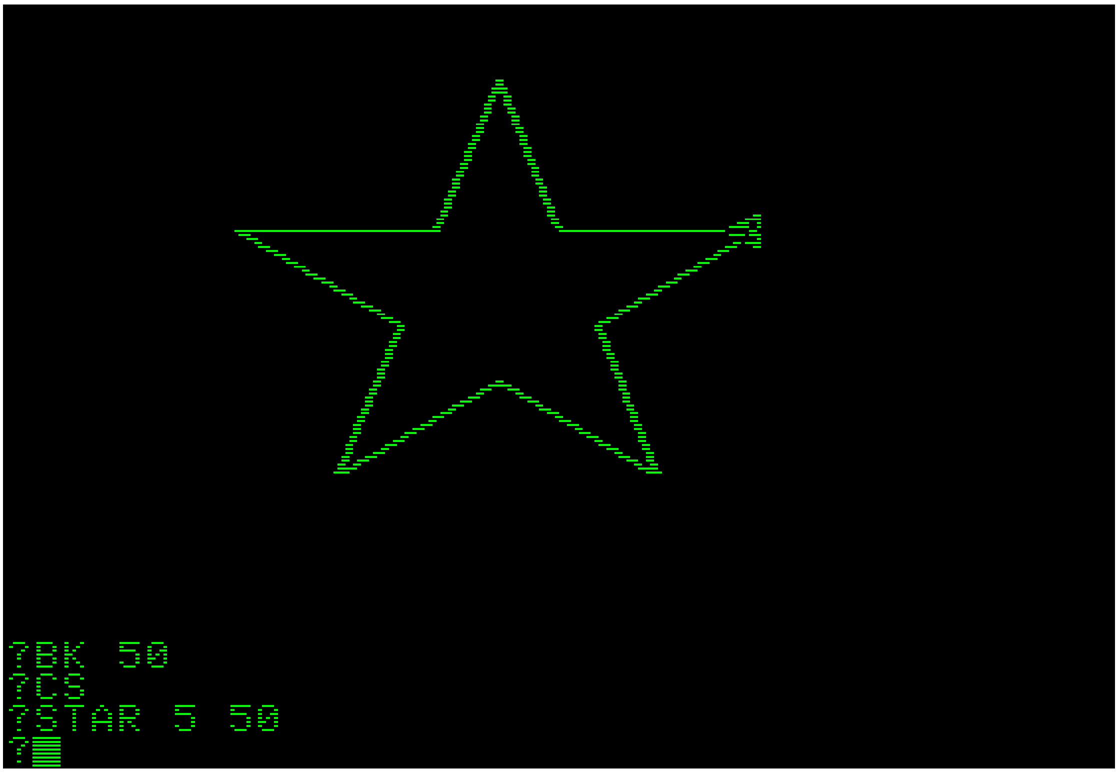
The constructionist learning theory was conceived by Seymour Papert in the 1980’s in the context of teaching children about mathematics and programming. Papert described his vision in his visionary book *Mindstorms: Children, Computers, and Powerful Ideas* [Pap80]:

> “Sesame Street” might offer better and more engaging explanations than a child can get from some parents or nursery school teachers, but the child

is still in the position of listening to explanations. By contrast, when a child learns to program, the process of learning is transformed. It becomes more active and self-directed. In particular, the knowledge is acquired for a recognizable personal purpose. The child does something with it. The new knowledge is a source of power and is experienced as such from the moment it begins to form in the child's mind.

2.2 Learning Environments

Seymour Papert's research in the 1980's led to the development of the first *learning environment*, Logo, which "uses computer[s] to teach some really profound concepts of Math to young children". "Instead of proposing a classical curriculum, [Papert] proposed a micro-world with its rules, in this case basic math rules, that made possible for a child the exploration of well known real world concepts in math with the help of a physical and graphical aid called Turtle, that a child can identify himself with." [Giu12]. A screenshot of Logo is shown in Figure 2.1.



```
?BK 50
?CS
?STAR 5 50
?█
```

Figure 2.1: The *Logo* learning environment

Since then, a number of learning environments have been created based on the constructionist approach to learning. These environments typically combine some sort of algorithm descriptions (e.g. a textual or visual programming language) and algorithm execution visualisation, framed within a “world” whose objects’ and operations’ form and action align with things and behaviours in the real world [SS11]. Modelling familiar environments aids learning because “what an individual can learn, and how he learns it, depends on what models he has available” [Giu12].

These environments promote a step-by-step approach to learning programming by intentionally reducing the amount of impressions simultaneously affecting the student [SS11], thus applying a form of *didactic reduction.* The user has the feeling that he “sees what he is doing” and that “something is happening” [Fru06] – this is because visualisation of algorithm execution allows immediate feedback regarding the correctness of a solution [SK09].

Psychological research shows that learning environments boost the learning effect because they facilitate *anchored instruction* by providing narrative anchors (i.e. problem situations) around which learning and teaching activities can be designed. This allows students’ brains to interconnect episodic (i.e. personal) and semantic (i.e. general, abstract) memories and knowledge, thus increasing the learning effect significantly [Hub13][1].

The most prominent example of learning environments is Scratch[2] which was released as a public online platform in 2007. It was originally “motivated by the needs and interests of young people (ages 8 to 16) at after-school computer centers such as the Intel Computer Clubhouses” [MRR$^+$10] but quickly became popular with a broader audience, resulting in “more than 500,000 projects [that] were shared on the Scratch Web site” in a little more than two years after launch [RMMH$^+$09]. As seen in Figure 2.2, it uses a visual programming language that interacts with a micro-world that appears on screen, thereby “[turning] variables into concrete objects that the user can see and manipulate, making them easier to understand through tinkering and observation” [MRR$^+$10].

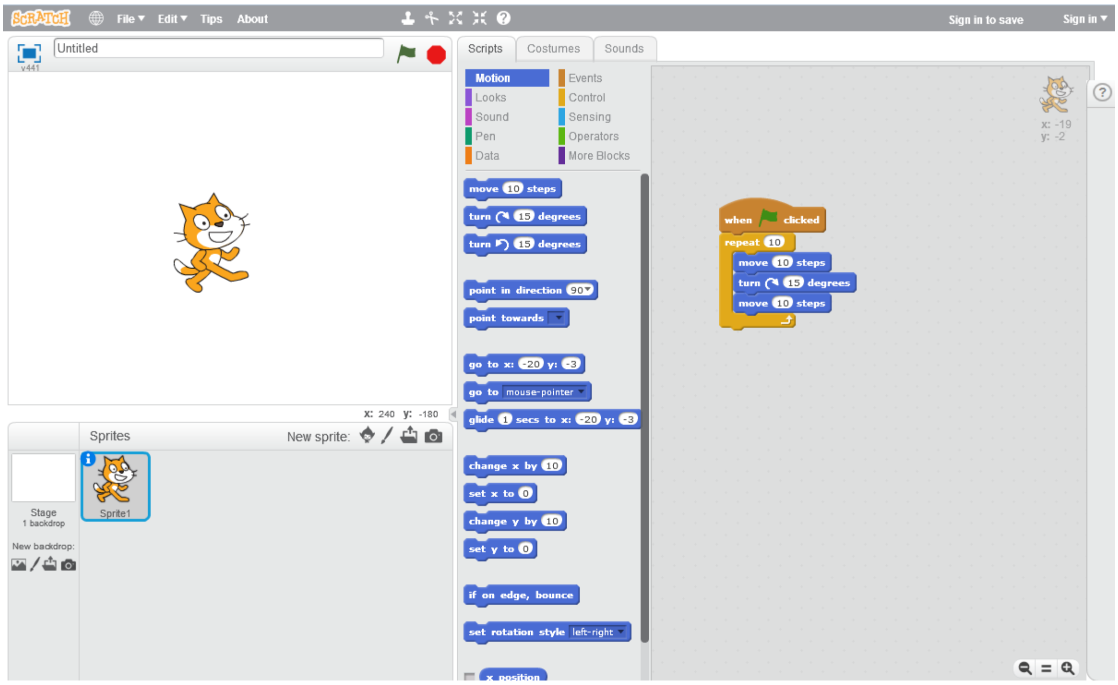
2.3 Block Languages

Learning to program typically requires students to acquire general knowledge about a programming paradigm, its control structures and other fundamentals, as well as specific knowledge about a certain programming language’s syntax [FT03]. [Fra15a] puts it this way: “Novice programmers are fighting two battles at once: the fight to translate their ideas into logical statements, and the fight to keep the syntax legal.”

Clearly, teaching algorithmic thinking should not focus on syntactic details of a concrete implementation but rather convey the idea on which the algorithm is founded. Visual programming languages facilitate this in an elegant and visually appealing way. A subset of those, namely *block languages*, appear to be best suited for beginners. In such languages, “the programmer assembles scripts from a menu of block-shaped command templates that are dragged onto a canvas”. These blocks “contain placeholders for variables and sub-clauses of the commands and can express scope of program-segment

[1] For more information on the organization of human memory, see [TD72]

[2] See `https://scratch.mit.edu`



```
when clicked
repeat 10
    move 10 steps
    turn ↻ 15 degrees
    move 10 steps
```

Figure 2.2: The *Scratch* learning environment

containment, relying on the notation of a block containing physically other blocks, with possible nesting" [MWW12].

The huge advantage for beginners is that advanced block language environments will make it "completely impossible to make a syntax error. There are no unbalanced parentheses, no unescaped strings, no missing semicolons" [Fra15a].

Evidence of the suitability of block languages for teaching and learning programming is the "popularity of the Scratch language [which] suggests that this style of coding is more accessible to children than standard programming languages" [MWW12]. [ML07] elaborates the advantages of block languages:

> [L]anguages like Java challenge students to master programmatic overhead before programming itself: students must become masters of syntax before solvers of problems. [. . .]
>
> [S]o accustomed are students today to graphical interfaces, "hello, world," whether written by student or teacher, cannot help but underwhelm. And, yet, in courses designed to recruit and retain budding computer scientists, it is perhaps just as important to excite as it is to instruct.

In order to "excite" students, block languages resemble puzzles and pose puzzle-like problems, which, as analysed in [LP02], "usually attract more interest on the part of students" while forcing them to "think about algorithms on a more abstract level, divorced from programming and computer language minutiae". This might even prove to students that "algorithm design strategies can be looked upon as general problem-solving tools that might be useful in areas far removed from computer science".

2.4 Algorithm Animation

A primary requirement for learning environments is to visualise processes that may otherwise be hidden from students.

[HDS02] suggests that researchers should "abandon the passive viewer paradigm in favor of paradigms that view a visualization as a resource for engaging students in an active process of discovery, reflection, and explanation." This is why instructors have long been "looking toward *algorithm animation* as a tool to help their students learn" [KST01], especially within learning environments.

Algorithm animation is a form of program visualisation that uses "the technology of interactive graphics and the crafts of graphic design, typography, animation, and cinematography to enhance the presentation and understanding of computer programs". It is "related to but distinct from the discipline of visual programming which is the use of various two-dimensional or diagrammatic notations in the programming process" [Bae86].

[KST01] illustrates that "[the] dynamic, symbolic images in an algorithm animation help provide a concrete appearance to the abstract notions of algorithm methodologies, thus making them more explicit and clear". Visualisation thereby "serves as a form of external memory, reducing the cognitive complexity of the programming task" [HDS02] and is an essential part of most modern learning environments.

2.5 Creating Custom Learning Environments

The previous sections presented advantages of learning environments that incorporate block languages. However, existing learning environments are usually hard to adapt to specific needs. Scratch, for example, excels at providing an online platform with a restricted command set but is difficult to adapt to custom requirements such as those presented by this thesis: While it "is possible to extend the environment to new languages by using the Scratch Extension Protocol" [FG11], this requires a lot of effort and does not allow hiding unnecessary features from the user.

This realisation has led to the creation of visual programming frameworks that can be used to build custom learning environments. The most popular and flexible one is Blockly[3], "a library for building visual programming editors" [Fra15b] that was first released in 2011 by Google. Its author aims at achieving the following characteristics [Fra12]: Blockly shall be...

- ...inviting:

[3] See `https://developers.google.com/blockly`

“Blockly is free, has no downloads, no plugins, and no installation. It is literally a click away.”

- . . . appealing:

  Blockly features a “highly polished UI; Gaussian blurs, rounded corners, and auditory feedback”.

- . . . unlimited:

  Blockly lets users “seamlessly migrate to a ’real’ language”.

- . . . relevant:

  “Domain-specific programming languages are always more suitable for that domain than general-purpose programming languages – especially when one lacks prior experience.”

Within less than two years after launch, Blockly has been translated to 43 languages. Its use in the online learning environment code.org has even led to videos of Mark Zuckerberg “explaining how to create loops in Blockly” and “Bill Gates explaining how to create conditionals in Blockly” [Fra13].

It is important to note that Blockly itself is targeted at developers. In order to make use of Blockly, it first needs to be adapted to a project’s needs and can then be “integrated into any application”, with “most of the high-level features [residing] in the domain-specific API blocks provided to Blockly by [the] host [application]” [Fra15a].

CHAPTER 3

Requirements for a Learning Environment for Teaching Maze Algorithms

This chapter gives a short overview of the Theseus program in Section 3.1. It has proven a valuable tool in teaching algorithmic thinking at Vienna University of Technology but lacks several aspects that could improve both learning effect and user experience. Analysing these shortcomings motivates the creation of Ariadne, as described in Section 3.2. Finally, a complete list of requirements for Ariadne is presented in Section 3.3.

3.1 Overview of Theseus

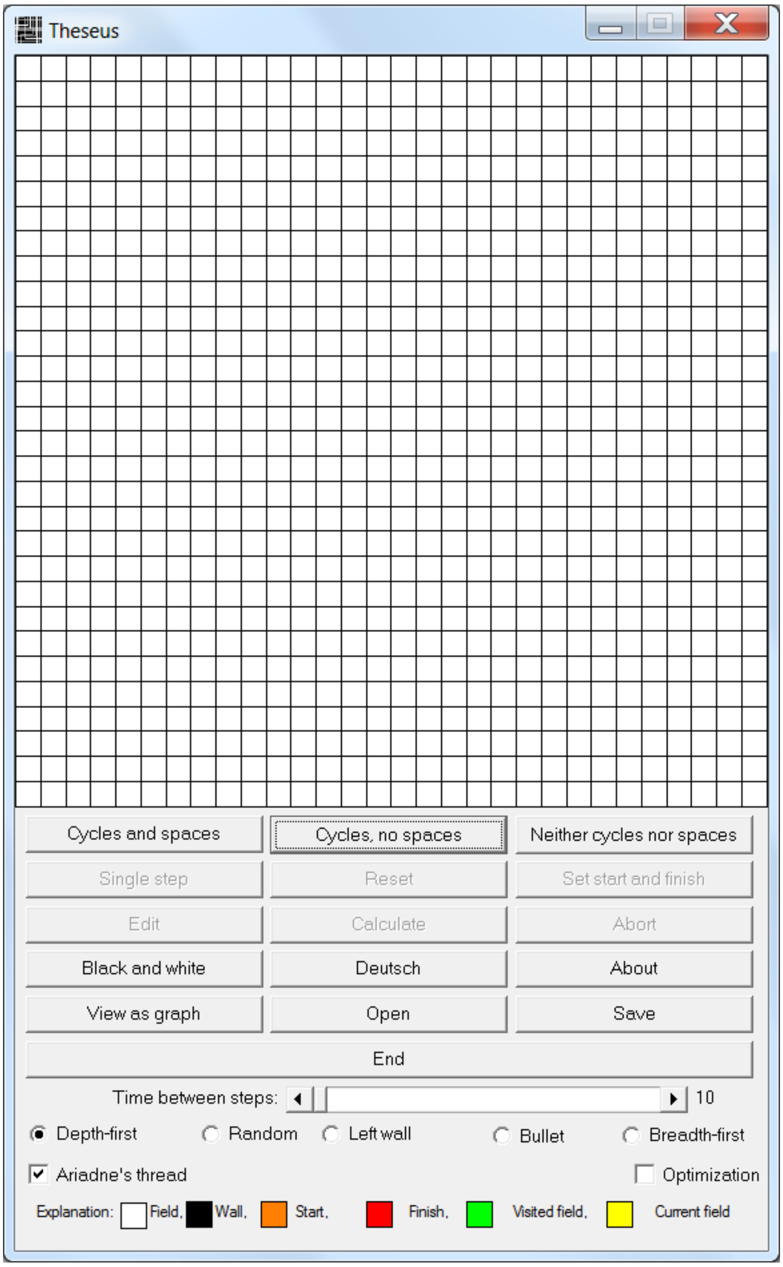
Theseus is a Windows program written by Marian Kogler in 2005. It is used for demonstration purposes in the PROLOG lectures on algorithmic thinking (see Chapter 1) and is available for students to download onto their personal computers[1]. Its start screen is shown in Figure 3.1.

The two main features of Theseus are the generation of mazes and the execution of maze solving algorithms, which will be described in Section 3.1.1 and Section 3.1.2, respectively.

3.1.1 Maze Generation

Theseus is able to generate mazes at any horizontal and vertical extent between 11 cells and 29 cells. Due to the maze visualisation implemented by Theseus, walls between cells always occupy the same space as a whole cell and the left-most, right-most, top and

[1] At the time of writing, the download was available from within the TUWEL e-learning platform under `https://tuwel.tuwien.ac.at/course/view.php?idnumber=188482-2015W`.



Time between steps: 10

● Depth-first ○ Random ○ Left wall ○ Bullet ○ Breadth-first

☑ Ariadne's thread □ Optimization

Explanation: □ Field, ■ Wall, 🟧 Start, 🟥 Finish, 🟩 Visited field, 🟨 Current field

Figure 3.1: Theseus start screen

bottom cells of the maze are always walls. The user can choose to generate a maze with cycles or without cycles. Figure 3.2 shows a 13x13 maze with cycles in Theseus.

Start and finish can be selected automatically as well as manually and are indicated by the orange and red cell, respectively.

3.1.2 Execution and Visualisation of Maze Solving Algorithms

After a maze has been generated, the user can select one of several maze solving algorithms and have Theseus execute it. Algorithm execution is visualised "live" by highlighting the cell under examination and marking visited cells turquoise, as seen in Figure 3.3.

It is during algorithm execution that Theseus' primary shortcoming becomes apparent: Theseus does not describe the algorithms it executes and thus forces users to derive the algorithm's steps by themselves[2]. This makes it difficult for users to understand algorithmic decisions during execution.

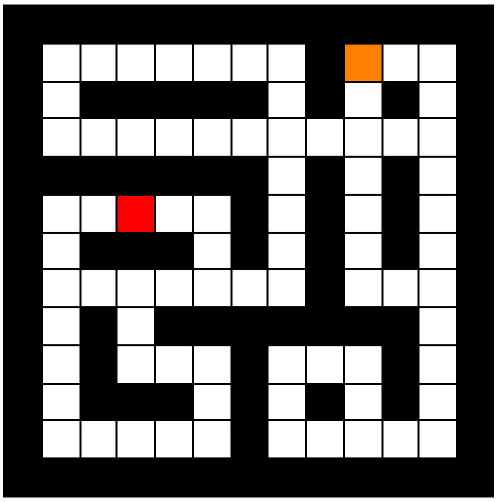


Figure 3.2: A maze generated by Theseus ("cycles, no spaces")

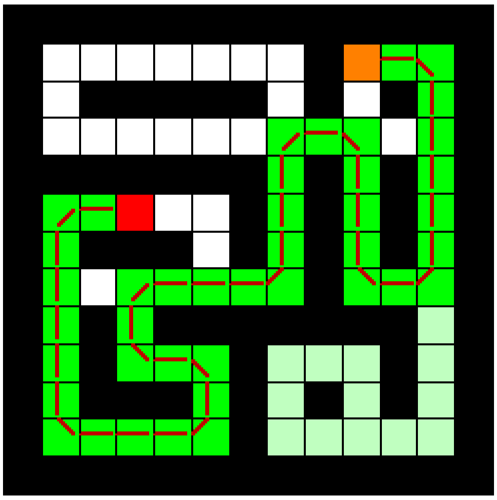


Figure 3.3: Execution of a maze solving algorithm

3.2 Vision for Ariadne

Recognizing the positive experiences made with Theseus, Ariadne will adopt the two main features of Theseus, namely maze generation and maze solving algorithm execution. However, an equally important third aspect will be added: The explicit visual description of the maze solving algorithms themselves, including the possibility to interactively modify these algorithms and to even create them from scratch.

[2] Even if the algorithm is formulated explicitly outside of Theseus, e.g. on slides, the learner is required to correlate the description on the slides with the execution visualisation in Theseus, having a negative impact on the learning experience and effect

Given this general vision for Ariadne, Chapter 4 will discuss what approach the new application will take in regard to the learning and teaching of algorithmic thinking, how the formulation of algorithms shall be implemented and how their execution shall be visualised.

3.3 Detailed List of Requirements

In-depth analysis of Theseus' current features has shown that some features are essential and need to be integrated into Ariadne, while others are "nice to have" or even unnecessary.

At the same time, other features (like the aforementioned ability to describe and modify maze generation algorithms) are highly relevant for the PROLOG lectures on algorithmic thinking and need to be added as new requirements for Ariadne. The following tables present a detailed list of Theseus' current features and requirements for the new Ariadne application.

3.3.1 Maze Generation

Feature	**Theseus**	**Ariadne**	**Comment**
Mazes with cycles	✓	✓	
Mazes without cycles	✓	✓	
Mazes with cycles and spaces	✓	✗	This option makes Theseus generate "rooms" which make it hard to think of the maze and its solution(s) in distinct passages.
Selection of map sizes	✓	✓	Ariadne will provide a number of pre-defined map sizes.
Selection of style of passages	✗	✓	Ariadne will allow choosing between "normal" and "long" passages.
Random selection of start and finish	✓	✓	Ariadne will randomly place start and finish during maze generation.
Manual selection of start and finish	✓	✓	
Manual editing of maze	✓	✗	This feature might be added to Ariadne at a later point in time, see Chapter 8.
Display of line of sight	✗	✓	Theseus does not give any information about the direction into which the algorithm is currently "looking".
Selection of initial line of sight	✗	✓	

3.3.2 Algorithm Formulation

Feature	Theseus	Ariadne	Comment
Description of algorithms	✗	✓	
Creation and modification of algorithms	✗	✓	

3.3.3 Pre-Defined Maze Solving Algorithms

Feature	Theseus	Ariadne	Comment
Random Walk	✓	✓	
Left Wall	✓	✓	
Simple Ariadne Thread	✓	✓	In Theseus: "Depth-first" + "Ariadne's thread"
Optimised Ariadne Thread	✓	✓	In Theseus: "Depth-first" + "Ariadne's thread" + "Optimization"
Bullet	✓	✗	This algorithm is not well documented.
Breadth-First	✓	✓	
Kruskal	✓	✗	In Theseus: Only available in graph view.
Prim	✓	✗	In Theseus: Only available in graph view.

Please note that algorithms which will not be available pre-defined in Ariadne can potentially be built by users themselves, due to the new possibility to "program" algorithms from scratch.

3.3.4 Algorithm Execution

Feature	Theseus	Ariadne	Comment
Start	✓	✓	
Stop	✓	✓	

Single step	✓	✓	
Reset	✓	✓	
Selection of execution speed	✓	✓	
Display of all shortest solutions	∼	✓	In Theseus: Only after execution of Breadth-First Search.
Visualisation of execution in the maze	✓	✓	Both Theseus and Ariadne show an actor moving through the maze during algorithm execution.
Visualisation of execution in algorithm	✗	✓	Ariadne will highlight the individual steps of the algorithm during execution.

3.3.5 Accessibility

Feature	**Theseus**	**Ariadne**	**Comment**
Accessible locally	✓	✓	
Accessible remotely	✗	✓	
Operating system independent	✗	✓	
Ease of use	✗	✓	Controls that can be clicked at any time but require the application to be in a certain state in order to work and the complete lack of documentation make Theseus rather difficult to handle for first time users.
Help dialog	✗	✓	

3.3.6 Display Options

Feature	**Theseus**	**Ariadne**	**Comment**
Colour graphics	✓	✓	
Black & white graphics	✓	✗	
English texts	✓	✓	
German texts	✓	✗	
Maze view	✓	✓	

Graph view	✓	✗	Experience with Theseus has shown that the graph view feature does not improve the learning experience of budding students of computer science but rather confuses them when the familiar maze visualisation is replaced by plain nodes and edges.

3.3.7 File Operations

Feature	**Theseus**	**Ariadne**	**Comment**
Open maze	✓	✓	
Save maze	✓	✓	
Open algorithm	✗	✓	
Save algorithm	✗	✓	

CHAPTER 4

Didactic Analysis

After defining what algorithms are and explaining why they should be taught in Section 4.1, a didactic analysis will be performed to determine the didactic approach that the implementation of Ariadne will be based upon.

According to [Bau96], didactic analysis happens in two steps. First, a range of didactic options is identified – these are alternatives out of which the most suitable one shall be determined. The second step is to define learning targets and content of the curriculum and assess the didactic options in the context of this subject matter. This analysis allows making an informed selection of one of the alternatives.

Didactic options will be presented and selected for three important aspects of Ariadne:

- For the general approach to teaching algorithmic thinking (Section 4.2).
- For the way of formulating algorithms (Section 4.3).
- For the way of visualising algorithm execution (Section 4.4).

4.1 Algorithms

This thesis is about teaching algorithmic thinking. In order to be able to present and evaluate didactic options for this endeavour, this section will define what algorithms are and establish why they should be taught to students of computer science.

4.1.1 Definition

Algorithms can be thought of as "simple extensions of our daily rational thinking process" detailing "step-by-step procedures to achieve [. . .] rational objectives" [Kin12]. Such "descriptions of processes" [BC15] could be precise machine language instructions as well as plain text cooking recipes. In the context of computer science algorithms it is therefore important to state that algorithmic work instructions for solving a problem or a task must be formulated precisely enough in order for a computer to execute them [Bol06].

Algorithms may display certain properties as listed by [Bol06]. These properties are either static (relating to the description of the algorithm) or dynamic (relating to the execution of the algorithm):

- Static properties
  - **Unambiguousness:** Algorithms should unambiguously describe a process for solving a given problem.
  - **Parameterability:** Algorithms should solve a class of problems which follow the same schema, rather than solving one particular problem only.
  - **Finiteness:** Algorithm descriptions should possess a finite length.
- Dynamic properties
  - **Executability:** Algorithms should not contain steps that cannot be executed.
  - **Termination:** Algorithms should terminate after a finite number of steps.
  - **Determination:** Determined algorithms will always produce the same results given the same inputs.
  - **Determinism:** During the execution of deterministic algorithms there are exactly zero or one possibilities to continue execution at any given moment.

The maze solving algorithms discussed in Chapter 6 possess most of these characteristics.

4.1.2 Motivation for Teaching Algorithms

There is an abundance of programming libraries, cloud services and APIs that can be re-used for the large majority of challenges a programmer might face each day. Thus, it is understandable if one questions the use of teaching algorithms (such as sorting, search or graph algorithms) and their derivation to students. However, similar to calculations in Mathematics, programming (and therefore algorithmic thinking) is a primary experience in computer science [SK09] and remains one of the most important subjects to teach. Understanding the foundations of algorithmic thinking is essential for being able to formulate new algorithms when faced with unfamiliar problem descriptions or tasks.

4.2 Teaching Algorithmic Thinking

4.2.1 Didactic Options

There are a various approaches to learning how to program [SS11][1]. Relevant approaches, i.e. didactic options, are listed below:

Through Programming Languages

In this approach, a programming language is learned by acquiring more and more complex language constructs. This is very similar to learning a foreign language, starting bottom-up [Fru06].

Through System Analysis

Following this approach, a complex system, its modules and their interaction are analysed top-down [Fru06].

Through Learning Environments

Learning environments facilitate easily accessible programming education for beginners because they combine descriptions of algorithms and their execution.

4.2.2 Selection of a Didactic Approach

The constructionist learning theory that was presented in Chapter 2 seems to be especially well suited as a basis for deciding on a didactic approach for teaching algorithmic thinking. It has been developed within the context of mathematics and computer science education and has led to the widespread use of *learning environments.* Comparing learning environments to the other didactic options given in Section 4.2.1 shows that they are better suited for teaching beginners:

- Unlike the approach through programming languages, they lay focus on algorithmic thinking rather than specific programming languages.
- Unlike the approach through system analysis, they are easily understood even without previous knowledge of programming and system design.

Following advice given in [SS11], elementary computer science education should convey a reduced, yet distortion-free, image of computer science. Learning environments clearly focus on this task. Therefore, Ariadne will be designed as a learning environment for students.

[1] It is valid to analyse approaches to learning how to program when it comes to learning how to understand, modify and create algorithms because algorithmic thinking is a subset of the skills required for programming whole systems (which typically contain a number of separate algorithms).

4.3 Formulating Algorithms

4.3.1 Didactic Options

There are endless ways of describing algorithms, differing in vocabulary, preciseness, level of abstraction, way of representation and other dimensions.

To illustrate different ways of describing algorithms, this section follows an example given in [Bol06] and formulates algorithms that calculate the sum of all natural numbers up to a given number n.

Colloquial Notation

Tasks for human beings will usually be described in a colloquial way. While such algorithm descriptions may not be unambiguous enough for programmatic execution, they can be interpreted correctly by humans because they are able to take context and personal experience into consideration. A colloquial description for an algorithm solving the given problem could be:

> Given a natural number n, calculate the sum of all natural numbers from 1 to n. This sum is the result.

Flowchart Notation

Figure 4.1 shows how the algorithm could be described using the flowchart notation. It is a graphical representation which clearly separates inputs and outputs from actionable statements and decision points.

Nassi-Shneiderman Diagrams

Nassi-Shneiderman diagrams, also known as *structograms*, assist in formulating highly structured representations of algorithms [NS73]. They do not allow GOTO statements and therefore encourage structured programming which results in better readability and easier understanding. An example is given in Figure 4.1.

Block Languages

Block languages support a way of visual programming that is especially easy to learn and understand, since students neither need to memorize available commands nor their specific syntax. An example is given in Figure 4.2.

Programming Languages

The most common way of describing algorithms in computer science is the exact specification using a programming language. Listing 4.1 shows the example algorithm in a programming language with syntax similar to that of the C programming language.

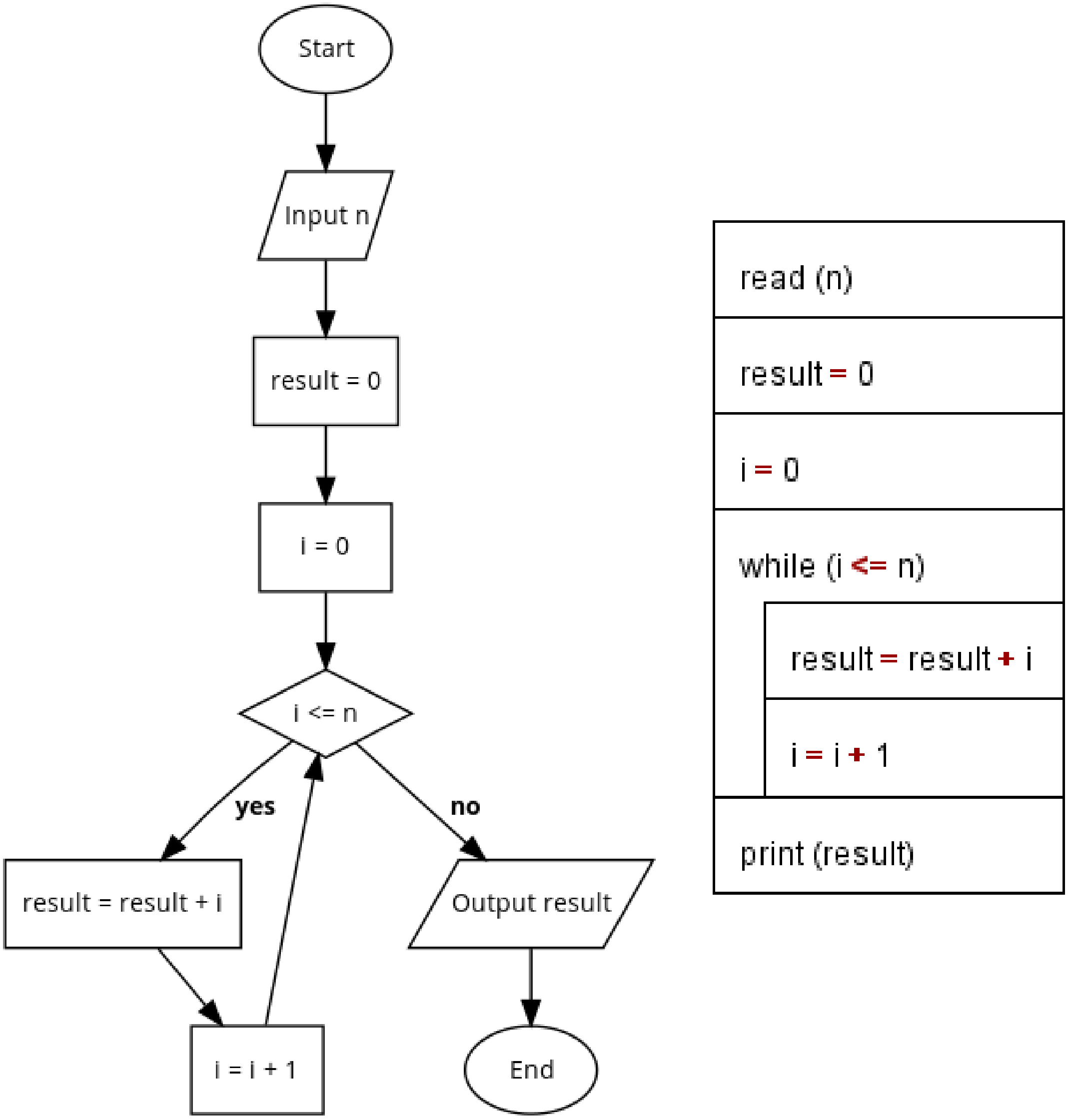


Figure 4.1: Flowchart notation (left) and Nassi-Shneiderman diagram (right)

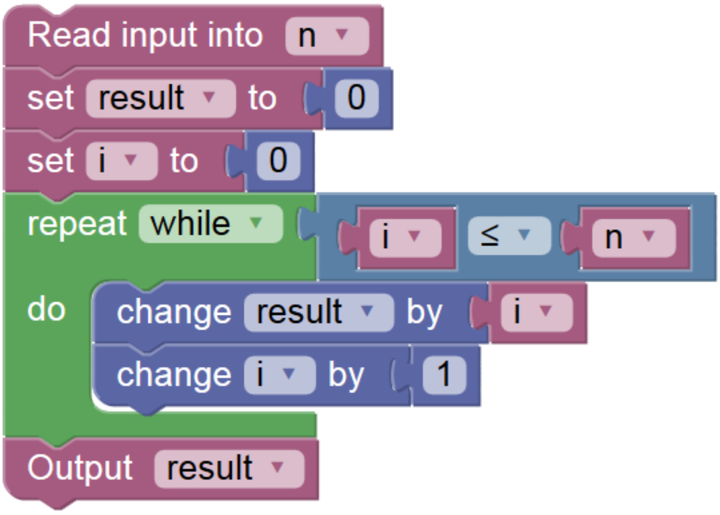


Figure 4.2: Block language

```
int n = readInt();
int result = 0;
int i = 0;

while (i <= n) {
    result = result + i;
    i = i + 1;
}

printInt(result);
```

Listing 4.1: Algorithm formulation resembling the C programming language

4.3.2 Selection of a Didactic Approach

When teaching algorithmic thinking, the focus should not lie on the specific way algorithms are described; any kind of formulation may be used. However, certain kinds of formulation may be better suited for beginners than others.

As presented in Chapter 2, constructionist learning theory advocates immediate graphical feedback in learning situations in order to ease understanding of the problem domain and the solution for the given problem. Continuing this thought, the learning effect could improve even more when not only the problem domain is visualised but also the algorithm operating on this domain. This is one of the reasons why *block languages* seem to be well suited to support the formulation of algorithms. Comparison to the other didactic options given in Section 4.3.1 reinforces this assumption:

- *Colloquial notation* may be easy to understand but does not allow programmatic execution of the described algorithm due to the formulation's ambiguity.

- *Flowchart notation* may result in confusing structures, given its ability to depict GOTO statements, i.e. arrows between any nodes.

- *Nassi-Shneiderman diagrams* are similar to the structured approach to programming used in block languages, but do not allow students to intuitively recognise how the "pieces of the puzzle" fit together.

- *Programming languages* require students to understand syntax expressed by keywords and identifiers specific to a programming language and to distinguish between this syntax and its semantics [FT03]. Also, program code might be formatted in an endless number of ways, while block languages will always present an algorithm in exactly the same way.

Ariadne will therefore integrate a block language into its learning environment.

4.4 Visualising Algorithm Execution

Executing an algorithm means feeding it any required input data and then sequentially executing its specific steps until there are no more steps to execute. Unless any kind of interaction with the user or the environment is required, this process may run by itself, hidden from any insight, and only output the calculation result once the algorithm has finished.

However, especially when it comes to teaching algorithmic thinking and trying to foster understanding for an algorithm's steps and decision points, it may be desirable to give insight into the algorithm by visualising its execution. While there is an arbitrary number of ways to do this, notable examples are given below:

4.4.1 Didactic Options

Log Messages

An algorithm may emit log messages when significant steps have been reached, outputting information about the algorithm execution's internal state at that point in time.

Stepping Through Source Code

Programmers have long relied on debuggers allowing them to step through source code while watching variable values and being able to intervene in order to change the algorithm execution outcome. In IDEs[2], "[s]tatic displays of program code can be animated automatically by highlighting the appropriate parts as the code runs" [Bro88].

Highlighting Blocks in a Block Language Environment

Similarly to stepping through source code, block languages may offer the possibility to highlight the block that is about to be executed.

[2] Integrated Development Environments

Algorithm Animation

Immediately visualising the results of algorithmic calculations during execution may constitute the best way to foster deep understanding of an algorithm's fundamental ideas. This can be seen impressively, for example, in the various ways that sort algorithms can be visualised by depicting the data that is being processed and the way it changes over time during algorithm execution [Bos14]. An example is shown in Figure 4.3.

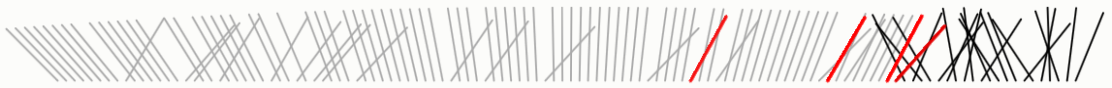


Figure 4.3: Visualisation of a sorting algorithm [Bos14]

A similar method which naturally fits learning environments is the graphical representation of actors within a micro-world displayed on screen who are then animated, following commands given to them from within the algorithm.

4.4.2 Selection of a Didactic Approach

As stated in Chapter 2, the constructionist learning theory advocates immediate graphical feedback in learning situations in order to ease understanding of the problem domain. After all, "[c]omputer programs in execution are complex objects whose properties can be difficult to fathom" [BS84]. Computer science education is "one obvious application" for algorithm animation [BS84] and since "students using animations report that they feel the animations assist them in understanding an algorithm" [KST01], Ariadne will make use of algorithm animation within its maze display and by block execution indication.

Furthermore, Ariadne will allow users to execute algorithms at varying speeds, pause their execution at any time and step through the algorithm block by block, as already suggested by [BS84].

4.5 Conclusion

Constructionist learning theory, as presented in Chapter 2, seems to be especially well suited as a basis for selecting didactic options for the purpose of this thesis. Analysis of the options shows that learning environments are an excellent way of supporting computer science education targeted at beginners (see Section 4.2) and are ideally complemented by block languages as a means of formulating algorithms (see Section 4.3) and algorithm animation as a means of visualising algorithm execution (see Section 4.4). Therefore, Ariadne will implement these approaches.

CHAPTER 5

Maze Structures and Maze Algorithms

[Fle14] gives the following brief introduction to mazes:

> The history of mazes and labyrinths, respectively, and the development of the first escape algorithms are at least as old as Greek mythology itself. For, as the story goes, Theseus used a thread given to him by Ariadne to track down the Minotaur in a maze and – after killing the creature – to find his way out again.

This chapter describes the technical aspects behind the myths and presents maze structures and ways to classify them in Section 5.1, methods for generating mazes in Section 5.2 and methods for solving mazes in Section 5.3.

5.1 Maze Structures

A maze "is a tour puzzle through which a solver must find a solution. It is a network of paths and hedges designed as a puzzle through which one has to find a way" [GS14]. Typically, a starting point and a finish point are given.

5.1.1 "Mazes" vs. "Labyrinths"

While "both maze and labyrinth depict a complex and confusing series of pathways" [Fol11], most sources distinguish between those two terms. This thesis aligns with the following consensus:

- A maze is multicursal (contains branching passages), and thus requires choices of path and direction. It may include dead ends. [med14]

- *Labyrinths* form a subset of mazes. They are unicursal, i.e. they contain "a single passage that never branches but winds in a convoluted path from start to finish" [BC15]. While they "[slow] down the solver from reaching the end point" [Fol11], they are not difficult to solve.

This thesis deals with (multicursal) mazes, since only they require effort in order to be solved.

5.1.2 Classification of Mazes

Mazes can be described along categories given by [Pul15][1]:

- **Dimension:** The number of dimensions in space a maze covers. This thesis focuses on two-dimensional mazes, while higher orders would be possible. For example, one can "imagine a three-dimensional maze as a group of two-dimensional mazes placed 'on each other' with an option to move upwards or downwards between them" [Fol11].
- **Topology:** A maze can either be "normal" (a standard maze in Euclidian space as examined in this thesis, see Figure 5.1) or "planair" (a maze with any kind of abnormal topology, e.g. on a cube or on a Moebius strip, see Figure 5.2).

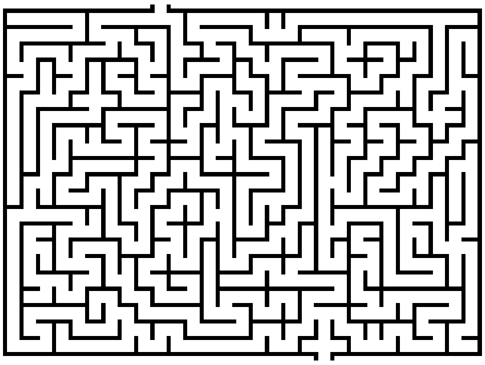


Figure 5.1: Normal topology

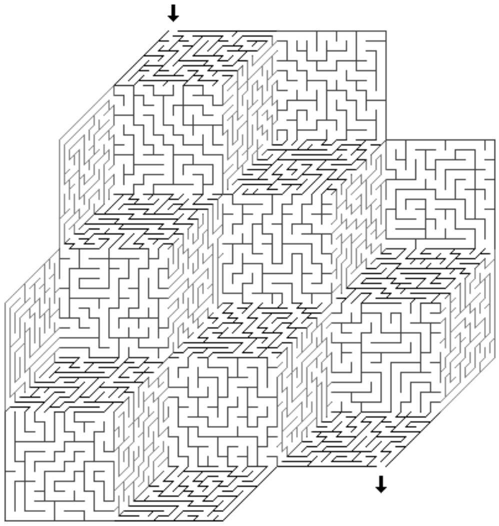


Figure 5.2: Planair topology

- **Tessellation** describes "the geometry of the individual cells that compose the maze". This thesis focuses on orthogonal mazes represented by a "rectangular grid where cells have passages intersecting at right angles" [Pul15]. Other forms of tessellation

[1] The categories *hyperdimension* and *focus* are left out because they are not relevant for this chapter.

are, for example, the “delta” tessellation (interlocking triangles, see Figure 5.3) or the “theta” tessellation (concentric circles of passages, see Figure 5.4).

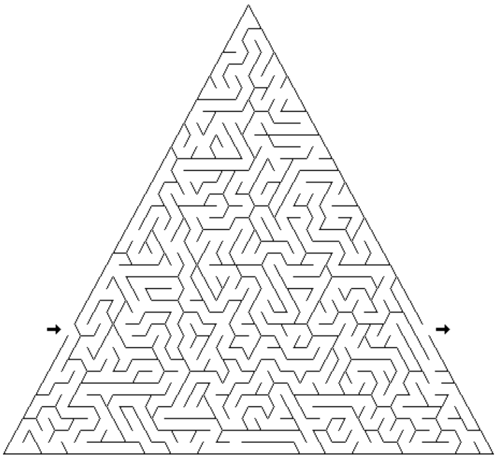


Figure 5.3: Delta tessellation

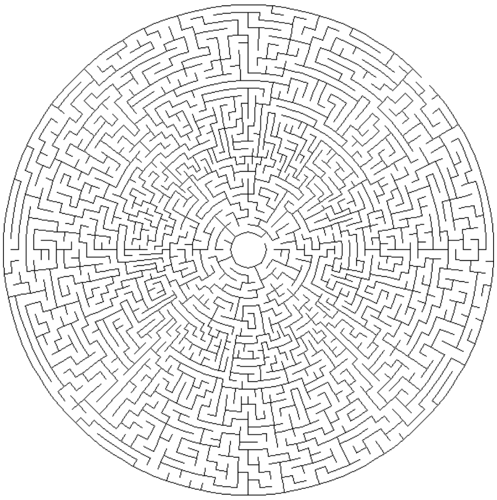


Figure 5.4: Theta tessellation

- **Routing:**
  - A *perfect maze* does not contain any cycles, so that there is exactly one path between any two cells (i.e. exactly one solution). It is “the simplest type of maze for a computer to generate and solve” [Kir15].
  - A *braid maze* does not contain any dead ends and will therefore usually contain cycles.
  - A *partial braid maze* contains both loops and dead ends.
  - A *unicursal maze* (also known as a “labyrinth”, see Section 5.1.1) “consists of just one snake-like passage without any choices for the solver to be taken [. . .]. Solving such a maze is generally easy. All it can cause is delaying the solver from reaching the end point.” [Fol11].

  Ariadne allows generating perfect mazes as well as partial braid mazes.

- **Texture** describes “the style of the passages of a maze, such as how long they tend to be and which direction they tend to go” [BC15], how many dead-ends there are and what the crossroad frequency is. As stated by [Fol11], “identifying a maze’s texture can be done either by just observing the maze or by expressing it using mathematical evaluation”. An algorithm’s “tendency to produce mazes of certain textures” [BC15] is referred to as its *bias*.

  For example, “[a] horizontally biased maze has (relatively) long horizontal passages and (relatively) short vertical passages” [Fol11]. A cut-out of a maze with a bias for horizontal passages is shown in Figure 5.5.

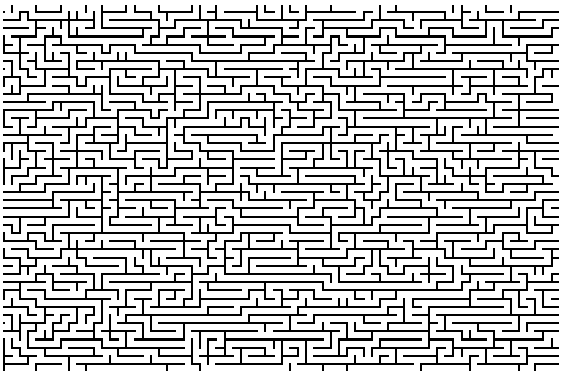


Figure 5.5: Horizontal bias

5.1.3 Mathematical Representation

Mathematically, a maze can be described as a simple, undirected, connected graph. This means that it possesses the following properties:

- It contains neither loops nor multiple edges.
- Its edges can be traversed in either direction.
- Any two vertices are connected by a path.

If an implementation chooses to model each single cell within a maze as a node, no weights need to be associated to the edges. However, if only junctions, dead ends and the start and finish position within the maze are modelled as nodes, the graph needs to be weighted so that the distances between the nodes can be taken into consideration when solving the maze.[2]

The mathematical representation can be used to describe any kind of maze, e.g. 3D mazes. Embedding their graphs on a two-dimensional surface even allows executing the regular plane algorithms on them. However, for reasons of simplicity, Ariadne will implement maze structures as an array of cells with links to neighbouring cells, as described in Chapter 7.

5.2 Maze Generation Algorithms

There is a multitude of maze generation algorithms to choose from. They differ in several dimensions, such as:

- Complexity of algorithm.
- Speed of algorithm execution.

[2] For an in-depth discussion of mazes in graph representation, see [Fle14] and [Fol11].

- Texture of the generated mazes.

Since "[t]here is no universally ideal algorithm for generating mazes" [BC15], an algorithm needs to be chosen depending of the requirements posed by the project, such as memory usage, speed or aesthetics.

5.2.1 Introduction to Maze Generation Algorithms

Maze generation algorithms can be classified by their so-called *focus* [Pul15] into "passage carvers" and "wall adders" [Fol11]. While *passage carvers* work on the precondition that "all the walls of the maze are up" in order to "selectively knock down walls" [Kir15] (also known as *linking cells to each other*), *wall adders* start on a blank grid and then build up walls between cells.

Both types of algorithm need to perform a process of examining and manipulating single cells, called "visiting" these cells [BC15]. Some algorithms can do so in any arbitrary order, while others depend heavily on the order in which they visit cells (e.g. when they are carving a passage from end to end before backtracking and starting to work on a second passage).

Most maze generation algorithms create a perfect maze (see Section 5.1.2) by "'growing' the Maze while ensuring the no loops and no isolations restriction is kept" [Pul15]. While doing so, "[i]n order to create a randomly [. . .] structured maze, at least one or more steps need to be randomized (i.e. the decision making within the step has to use a function returning a random result)“ [Fol11].

Since the resulting maze does not contain any cycles and is easy to solve, many situations will require *braiding* the maze, i.e. adding cycles, usually by removing dead ends. In the end, "[m]azes may be heavily braided (with all or most dead ends removed), or lightly braided (with only a few dead ends removed), or anything in-between" [BC15]. Ariadne will make use of an algorithm called *dead-end culling*, as presented in Chapter 7.

5.2.2 Requirements for Ariadne's Maze Generation Algorithm

In order to comply with the requirements stated in Chapter 3, Ariadne needs to be able to produce mazes without cycles as well as mazes with cycles. Following the definitions given in Section 5.1, this means perfect mazes and partial braid mazes[3].

While beauty is in the eye of the beholder, Ariadne should try to produce mazes that are a good mix of long and short passages without any significant bias towards unusual textures.

Other algorithm properties like speed or efficiency do not need to be taken into consideration since the generated mazes are small enough to be calculated within a matter of milliseconds, regardless of the specific algorithm.

[3] Ariadne will not generate complete braid mazes, that is mazes without any dead-ends, because dead-ends pose their own challenges when designing maze solving algorithms.

5.2.3 Selection of Maze Generation Algorithm

The Growing Tree Algorithm

[BC15] presents the so-called *Growing Tree Algorithm* which belongs to the family of passage carving algorithms and generalises two simple variations of Prim's Algorithm. The resulting algorithm can easily be configured to behave differently and act like a Recursive Backtracker Algorithm, for example. More interestingly even, the algorithm can be "configured to mimic attributes of different algorithms, simultaneously" [BC15]. Ariadne will implement this algorithm because of its flexibility and convincing results which are exempt of any bias. Other algorithms may be easier to implement (e.g. the Binary Tree Algorithm which only needs to iterate over all cells in any order and only keep the current cell in memory) but produce less appealing mazes[4].

Ariadne will make use of the algorithm's flexibility and allow the user to choose between "normal passages" and "long passages", further referred to as the *mode* of maze generation.

Algorithm 5.1 shows the Growing Tree Algorithm in pseudocode. Please note that the maze structure is represented as a rectangular grid of cells, as it will be used by the implementation of Ariadne.

Algorithm 5.1: The Growing Tree Algorithm as implemented in Ariadne

```
Data: a grid of cells with walls between each cell and its neighbours, a mode of
      maze generation (either 'normal passages' or 'long passages')
Result: the same grid with passages carved through the cells in order for them to
        form a perfect maze

active ← a list containing a random cell of the grid as its only element
while active is not empty do
    if mode is 'long passages' or coin toss shows heads then
        cell ← the last cell that was added to active
    else
        cell ← a random cell out of active
    end
    unvisitedNeighbours ← cell's neighbours with intact surrounding walls
    if unvisitedNeighbours is not empty then
        randomUnvisitedNeighbour ← a random cell out of unvisitedNeighbours
        link cell to randomUnvisitedNeighbour
        add randomUnvisitedNeighbour to active
    else
        remove cell from active
    end
end
```

[4] For a detailed explanation and comparison of maze generation algorithm, please refer to [BC15] and [Pul15].

This algorithm builds a set of *active* cells out of which it repeatedly selects one in lines 3–7. This is where the algorithm mixes different styles together. If the *mode* was chosen to produce long passages, line 4 will always be used as a cell selection mechanism. If *mode* was chosen to produce normal passages, however, the coin toss function in line 3 mixes the cell selection criteria in line 4 and those in line 6. While line 4 resembles the depth-first Recursive Backtracker Algorithm, line 6 resembles the Simplified Prim's Algorithm.

Therefore, if long passages are requested, the algorithm will always behave exactly like the Recursive Backtracker Algorithm (thus carving passages until there is nowhere else to go, then backtracking one step and trying to carve new passages from there). On the other hand, if normal passages are requested, the algorithm will sometimes choose to function like the Recursive Backtracker and other times choose to function like a Simplified (unweighted) Prim's Algorithm which leads to earlier branching and more junctions.

Experimentation has shown that this mix of maze generation styles results in well-balanced mazes with passages that are neither too short nor too long.

Line 8 finds all unvisited neighbours of the selected cell. If there are any, lines 10–12 randomly choose one of those neighbours and carve a passage to it, adding it to the list of *active* cells. If there are no unvisited neighbour cells, however, the current cell is removed from the list of *active* cells because it will never be used for carving passages out of again.

Generating Braid Mazes

The Growing Tree Algorithm generates a perfect maze, i.e. one without cycles. However, since Ariadne will allow the user to generate mazes with cycles as well, there needs to be a way to generate (partial) braid mazes. The solution is to run the Growing Tree Algorithm and afterwards manipulate the resulting maze in order to add cycles. There are a number of ways to do this, one is *dead-end culling* as proposed by [BC15] and listed in Algorithm 5.2.

This algorithm iterates over all dead-end cells in the grid. Lines 3–5 check whether the cell is still a dead-end – it could have already been linked to another cell in a previous iteration. Unless that is the case, the algorithm goes on to calculate a random number between 0 (inclusive) and 1 (exclusive) and compares it to the value of the input parameter p. If it is greater or equal, execution continues with the next dead-end cell without manipulating the current one. This lets the user specify the probability of removing dead-ends, thus allowing fine-tuning the algorithm according to personal taste.

Next, all neighbouring cells are determined that are not currently linked to the dead-end cell under inspection. Out of those, the algorithm prefers cells which are dead-ends themselves in order to "kill two birds with one stone" (line 10). If such cells do not exist, all unlinked neighbours remain valid candidates (lines 11–13).

Finally, one cell out of the candidates is chosen randomly and the dead-end cell under inspection is linked to it, i.e. a passage between the two is carved. By doing this, a cycle is generated within the maze (which, therefore, becomes harder to solve).

Algorithm 5.2: Dead-end culling

```
Data: a grid with passages carved through its cells, $p$ = probability of removing
      dead-ends
Result: the same grid with some of its dead-ends removed in favour of cycles
deadendCells ← all dead-end cells in the grid, in random order
foreach cell in deadendCells do
    if cell is not a dead-end then
        continue to next iteration
    end
    if random number in range [0, 1) >= $p$ then
        continue to next iteration
    end
    unlinkedNeighbours ← cells currently not linked to cell
    candidates ← all dead-end cells out of unlinkedNeighbours
    if candidates is empty then
        candidates ← unlinkedNeighbours
    end
    link cell to a random cell out of candidates
end
```

5.3 Maze Solving Algorithms

The question of how to find a path from point A to point B within a maze is well researched but remains fascinating to this day. A formal definition of the underlying “maze search problem” is given by [Fle14]:

> Describe a general algorithm which constructs a closed covering walk W in a connected graph G such that, in the course of constructing W, this algorithm can only handle local information available at any vertex reached by W.

The last part of the quote is significant: The algorithm shall only be able to “see” the immediate surroundings of its current path. It shall not know anything about the finish and its location within the maze, or the current cell’s location within the maze. The situation resembles then a person led into a maze while being blind-folded. After having the blindfold removed, the person is free to walk through the maze on his or her search for the finish.

The person may, however, have certain accessories which might help on the journey:

- The person may attach an infinitely long piece of thread at the starting point and then lay out the thread as he or she walks through the maze. This allows the person to trace back his or her way without having to remember all previous turns by heart. While tracing back, the thread would be rolled up again. After finding the finish, the thread can be used to trace the path from the finish point back to the starting point.

- The person may somehow leave marks on the parts of the maze that are being visited, e.g. by painting a sign onto the floor. Coming across a marked cell reveals that this part has already been visited (and that it may not be worth going that way again because of a cycle within the maze).

As discussed in Chapter 3, Ariadne will provide a number of pre-defined algorithms which will be further discussed in this section. However, as stated before, the user is free to experiment with Ariadne in order to create new or modified algorithms.

Please note that the pseudocode in this chapter will discuss algorithms for use by human beings. The actual implementation of these algorithms within the context of Ariadne and its block language will be presented in Chapter 7. For a detailed discussion of maze solving algorithms, please refer to [Pul15].

5.3.1 The Random Walk Algorithm

The Random Walk Algorithm as listed in Algorithm 5.3 is the simplest one pre-defined by Ariadne in terms of "having to think". The person in the maze simply takes random paths from junction to junction until the finish is found. (Upon reaching a dead-end, the person walks back to the previous junction since this would be the only available path.)

Algorithm 5.3: The Random Walk Algorithm

```
Data: a person on a given starting point within a maze, a finish point within the
      same maze
Result: the person standing on the finish point
start laying out thread
while person has not found finish do
    choose a random path and walk along until the finish or a junction is reached
end
```

The person does not "remember" anything about its previous paths, therefore potentially making "stupid" decision like walking along a cycle several times in succession. Given enough time, the algorithm will find the finish but is obviously not efficient in doing so. Still, the thread could then be used to describe the exact path taken from start to finish.

5.3.2 The Left Wall Algorithm

The Left Wall Algorithm[5], as listed in Algorithm 5.4, is very simple to understand for a human person: "Always walk along the left wall". In a dead-end, this would make the

[5] The Left Wall Algorithm is also known as the Wall Follower Algorithm [BC15].

person go back the way he or she came from.

Algorithm 5.4: The Left Wall Algorithm

```
Data: a $person$ on a given starting point within a maze, a $finish$ point within the
      same maze
Result: the $person$ standing on the $finish$ point
start laying out thread
while $person$ has not found $finish$ do
    follow the left wall until the $finish$ or a junction is reached
end
```

While it is easy to grasp, The Left Wall Algorithm has one big disadvantage: The algorithm is only guaranteed to find the finish within mazes that do not have cycles. For mazes that do have cycles, depending on the start and finish positions, the algorithm may find the target or may get caught up in an endless loop along a cycle.

5.3.3 The Simple Ariadne Thread Algorithm

The Simple Ariadne Thread Algorithm as shown in Algorithm 5.5 is a variant of the depth-first traversal of a graph. The person is told to lay out thread along the way while always following the left wall. In a dead-end the person would need to walk back for a bit and roll up the thread laid out before.

The person is also told to never walk *across* thread that has been rolled out before, because this situation would mean that a cycle has been reached. The thread thus allows making the informed decision that walking along would mean entering a cycle and that it is better to treat the current cell as a dead-end, turn back and roll up thread along the way.

Algorithm 5.5: The Simple Ariadne Thread Algorithm

```
Data: a $person$ on a given starting point within a maze, a $finish$ point within the
      same maze
Result: the $person$ standing on the $finish$ point
start laying out thread
while $person$ has not found $finish$ do
    follow the left wall until the $finish$ or a junction is reached, without crossing
    previously laid out thread
end
```

This algorithm is guaranteed to find the finish in both mazes with and mazes without cycles. However, it is clearly not very efficient because it does not remember which cells have been visited before, therefore potentially visiting cells several times without gaining anything from these extra steps. The Optimised Ariadne Thread Algorithm realises this potential for improvement.

5.3.4 The Optimised Ariadne Thread Algorithm

Improving on the previous algorithm, the Optimised Ariadne Thread Algorithm as shown in Algorithm 5.6 instructs the person in the maze to mark cells that are being visited. This allows recognizing previously visited cells.

Algorithm 5.6: The Optimised Ariadne Thread Algorithm

```
  Data: a person on a given starting point within a maze, a finish point within the
        same maze
  Result: the person standing on the finish point
1 start laying out thread
2 start marking visited cells
3 while person has not found finish do
4     if person sees unvisited paths then
5         follow the left wall until the finish or a junction is reached, without
          walking onto previously visited cells
6     else
7         backtrace to the previous junction
8     end
9 end
```

Line 4 tells the person to evaluate whether any unvisited paths can be seen, i.e. paths that have not been marked as visited yet. If such paths exist, the person is told to follow the left-most one until the *finish* or another junction is reached, or until a previously visited cell lies ahead. Once a dead-end is reached or there is no unvisited path to take, line 6 instructs the person to roll up the thread laid out before until the previous junction.

This algorithm is guaranteed to find a path from start to finish and does so in an efficient manner – no cell is visited more often than twice. However, just like the algorithms before, this algorithm is not guaranteed to find the *shortest* path. This is what the next algorithm is designed for.

5.3.5 The Breadth-First Search Algorithm

The Breadth-First Search Algorithm as implemented by Ariadne and listed in Algorithm 5.7 allows finding a shortest path from the start to the finish. Note that there may be other equally short paths which are not found, however, because the algorithm terminates after finding one shortest path.

The algorithm resembles a breadth-first search on a graph. Each iteration extends the already found paths by one single step. When a junction is encountered, the algorithm clones the person at the junction into several persons who then each follow one of the paths on their own. As soon as one of the persons in the maze has found the finish, this path is known to be a shortest path.

One person alone could not reasonably execute this algorithm, which is why the metaphor of "splitting up into clones" has been introduced in the pseudocode. The reason for not using more common programming terms (e.g. by using an array of paths) is that

Algorithm 5.7: The Breadth-First Search Algorithm

```
Data: a person on a given starting point within a maze, a $finish$ point within the
      same maze
Result: one clone of the person standing on the $finish$ point
start laying out thread
start marking visited cells
$finished \leftarrow$ false
while not $finished$ do
    foreach person in maze do
        pathCount ← number of unvisited paths seen by person
        switch pathCount do
            case 0 tell person to give up
            case 2 or more tell person to split up into pathCount clones
        endsw
    end
    foreach person in maze do
        tell person to step onto a random unvisited neighbour cell
        if person found finish then
            $finished \leftarrow$ true
            break;
        end
    end
end
```

the block implementation shown in Chapter 7 will be using the very similar notion of “cloning avatars”.

CHAPTER 6

Preparation of Teaching Material

While Chapter 5 has specified the algorithms that will be pre-defined as block language constructs in Ariadne, the *level of abstraction* that these algorithms will be formulated on still needs to be defined. Section 6.1 discusses why this decision is important. Individual blocks for describing maze solving algorithms are then presented in Section 6.2.

6.1 Selection of a Level of Abstraction

[Win08] states that "[t]he essence of computational thinking is abstraction" and goes on to define the underlying process as follows:

> The abstraction process introduces layers. In computing, we work simultaneously with at least two, usually more, layers of abstraction: the layer of interest and the layer below; or the layer of interest and the layer above. Well-defined interfaces between layers enable us to build large, complex systems. Given the application programming interface (API) of a software component, a user need not know the details of the component's implementation to know how to interact with it, and an implementer need not know who all the component's potential users might be in order to implement it correctly.

When it comes to the way that algorithms are formulated, this means that Ariadne could present its maze solving algorithms on any level of abstraction that can be thought of, providing coarse, generic block constructs that are useful for solving a certain task but can hardly be re-used in another context, or very fine-grained blocks that can be used in a number of contexts but would require its users to have deeper understanding of the "lower layers", i.e. the way that mazes and the objects acting on it are implemented.

As one simple example, consider the Random Walk Algorithm which was presented in Chapter 5, re-printed here as Algorithm 6.1:

Algorithm 6.1: The Random Walk Algorithm

```
Data: a person on a given starting point within a maze, a finish point within the
      same maze
Result: the person standing on the finish point

start laying out thread
while person has not found finish do
    choose a random path and walk along until the finish or a junction is reached
end
```

This pseudocode could be rewritten on a higher level of abstraction as listed in Algorithm 6.2.

Algorithm 6.2: The Random Walk Algorithm – higher level of abstraction

```
Data: a person on a given starting point within a maze, a finish point within the
      same maze
Result: the person standing on the finish point

start laying out thread
find finish by following random paths
```

Taking the other direction, the algorithm could of course also be rewritten on a lower level of abstraction as in Algorithm 6.3, revealing more implementation details than the versions before.

Algorithm 6.3: The Random Walk Algorithm – lower level of abstraction

```
Data: a person on a given starting point within a maze, a finish point within the
      same maze
Result: the person standing on the finish point

start laying out thread
while person has not found finish do
    neighbours $\leftarrow$ the cells linked to the cell on which person is standing
    if count of neighbours $> 1$ then
        neighbours $\leftarrow$ neighbours without the last cell that thread was laid out on
    end
    nextCell $\leftarrow$ a random element out of neighbours
    move person to nextCell
end
```

The initial design of block language constructs, their iterative re-design and the diligent observation of how they work together in Ariadne's maze solving algorithms have led to a level of abstraction which aims at finding the balance between *didactic reduction* (i.e. the simplified depiction of complex issues, see Chapter 4) and the requirement for learning environments to convey a distortion-free image of computer science [SS11].

The outcome aligns with the opinion of the developer of Blockly [Fra15c], a framework for visual programming that will be used for implementing Ariadne (see Chapter 7):

> Wherever possible a higher-level approach should be taken, even if it reduces execution performance or flexibility. Consider this Apps Script expression:
>
> `SpreadsheetApp.getActiveSheet().getDataRange().getValues()`
>
> Under a 1:1 mapping which preserves all potential capabilities, the above expression would be built using four blocks. But Blockly aims for a higher-level and would provide one block that encapsulates the entire expression. The goal is to optimize for the 95% case, even if it makes the remaining 5% more difficult. Blockly is not intended to be a replacement for text-based languages, it is intended to help users get over the initial learning curve so that they can use text-based languages.

The individual blocks which have been designed based on the principles laid out above are presented in Section 6.2.

6.2 Visual Programming Blocks

This section presents the individual blocks that are necessary to implement the pre-defined maze solving algorithms presented in Chapter 5[1].

The Breadth-First Search Algorithm needs to be able to deal with multiple avatars, hence requiring more sophisticated blocks. In order to keep the other algorithms as simple as possible, Ariadne will include two sets of blocks – one set that deals with "single avatar" situations and one set that deals with "multiple avatar" situations.

The internal structure of each block shows the text that the user will see on the block. Items within square brackets ("[" and "]"), separated from each other by the "|" sign, denote drop-down fields out of which the user needs to select one choice. Items within angled brackets ("<" and ">") denote places where other blocks need to be attached.

6.2.1 "Single Avatar" Blocks

These blocks are used by the algorithms which feature one single avatar moving through the maze, i.e. the Random Walk Algorithm, the Left Wall Algorithm, the Simple Ariadne Thread Algorithm and the Optimised Ariadne Thread Algorithm.

[1] Note that additionally to the custom maze-related blocks shown in this chapter, a number of blocks are required that are not related to mazes at all, for example variable assignment, if/then/else and loop structures. These blocks will not be presented in this chapter because the reader is most likely familiar with their semantics. The implementation, as described in Chapter 7, will set up on the Blockly library which has already built in the mentioned blocks for immediate use. For more information, please see the Blockly Homepage [Fra15b] and the Blockly Wiki [Fra14a].

- **`backtrace`**

  This block instructs the avatar to roll up thread. If the avatar has not even started laying out thread before, nothing happens. Otherwise, the thread is traced back until...

  - ...the finish is reached.
  - ...a junction is reached.
  - ...there is no more thread to roll up.

-
  ```
  follow  [random path |
           1st path from left |
           1st path from right]
  without [any condition |
           crossing previously laid out thread |
           walking onto previously visited cells]
  ```

  This blocks tells the avatar to follow either a random path, the first path from the left or the first path from the right (depending on the avatar's place in the maze and its line of sight) out of a set of eligible paths.

  The second drop-down allows for constraining which paths are seen as eligible:

  - *Without any condition* means that all paths leading away from the avatar's current position are eligible.
  - *Without crossing previously laid out thread* means that the avatar must not step onto a cell on which thread has already been laid out before – except for the single cell on which the last piece of thread was laid out.
  - *Without walking onto previously visited cells* means that the avatar must not step onto a cell which has already been marked as visited.

  If there is no eligible path, nothing happens. Otherwise, the avatar chooses one of the eligible paths:

  - To choose a *random path*, the avatar uses a randomising function to select one. Note, however, that unless it has not moved yet at all, the avatar chooses the way "backwards" only if there are no other cells available. Without this logic, the avatar might walk from junction A to junction B and then right back from junction B to junction A – this would seem unnatural to the observer.
  - To choose the first path from the left, the avatar determines whether the path to its left is eligible. If not, the paths straight ahead, to the right and backwards are evaluated in this order.
  - To choose the first path from the right, the avatar determines whether the path to its right is eligible. If not, the paths straight ahead, to the left and backwards are evaluated in this order.

Having chosen a path, the avatar follows it until…

- …the finish is reached.
- …a junction is reached.

If walking along the chosen path means that the avatar is walking backwards along the most recently laid out thread, the avatar rolls up the thread while walking.

- **`found finish?`**

  This block returns *true* if the avatar is standing on the finish cell and *false* otherwise.

- **`number of unvisited paths`**

  This block returns the number of paths leading away from the avatar's current position that have not been marked as visited yet.

- **`say`** *<MESSAGE>*

  This block instructs the avatar to print a message to the screen. The message is shown for 3 seconds.

- **`start laying out thread`**

  This block tells the avatar to start laying out thread. One can imagine the avatar ramming a plug into the floor, attaching an infinitely long thread to it and subsequently laying out the thread along the chosen paths.

- **`start marking visited cells`**

  This block tells the avatar to mark cells as visited once it walks onto them. One can imagine the avatar painting a sign onto the floor of the cells it is visiting.

- **`thread length`**

  This block returns the length of the thread that the avatar has laid out. Having walked from the first cell to a neighbouring cell, for example, 1 would be returned. Without having laid out any thread, 0 is returned.

6.2.2 "Multiple Avatar" Blocks

These blocks are used by the Breadth-First Search Algorithm which requires the initial avatar (that is already placed into the maze before the algorithm is executed) to split up into multiple avatars in order to find the shortest path to the finish. Most blocks require the user to specify which exact avatar instance shall receive a command.

- **all avatars**

  This block returns a list containing all active avatars (i.e. avatars which have not given up their search) that are currently in the maze.

- **avatar** *<AVATAR>* **found finish?**

  This block behaves like its single avatar version but allows the user to specify a specific avatar instance.

- **avatar** *<AVATAR>***'s thread length**

  This block behaves like its single avatar version but allows the user to specify a specific avatar instance.

- **initial avatar**

  This block returns the initial avatar instance, i.e. the avatar that has been placed into the maze already before algorithm execution.

- **make** *<COUNT>* **avatars out of avatar** *<AVATAR>*

  This block is used for cloning avatars. Results differ depending on *count*:

  - If *count* is 0, the avatar gives up its search for the finish and becomes inactive. (See the description of the equivalent "tell avatar *<AVATAR>* to give up" block below.)
  - If *count* is 1, nothing happens.
  - If *count* is 2 or higher, the avatar splits up into *count* avatars. Each new avatar inherits the original avatar's position and line of sight, as well as its preference to lay out thread and to mark visited cells. Additionally, the original avatar's whole thread is cloned from beginning to end.

- **number of unvisited paths seen by avatar** *<AVATAR>*

  This block behaves like its single avatar version but allows the user to specify a specific avatar instance.

- **tell avatar** *<AVATAR>* **to give up**

  This block instructs the specified avatar to give up its search for the finish and become inactive. Afterwards, the avatar will not react to commands anymore and the list returned by `all avatars` will not include this avatar.

- **`tell avatar`** *<AVATAR>* **`to say`** *<MESSAGE>*

  This block behaves like its single avatar version but allows the user to specify a specific avatar instance.

- **`tell avatar`** *<AVATAR>* **`to start laying out thread`**

  This block behaves like its single avatar version but allows the user to specify a specific avatar instance.

- **`tell avatar`** *<AVATAR>* **`to start marking visited cells`**

  This block behaves like its single avatar version but allows the user to specify a specific avatar instance.

- **`tell avatar`** *<AVATAR>* **`to step onto a random unvisited cell`**

  This block tells the specified avatar to step onto a neighbouring cell that has not been visited yet. Note that the avatar does not follow the path for a number of steps but takes exactly one single step. If there is no unvisited cell next to the avatar, nothing happens.

CHAPTER 7

Implementation

After the previous chapters have defined what requirements Ariadne needs to fulfil as a learning environment and what contents it shall pre-define in order to allow students an easy start, this chapter will give details about how its features have actually been implemented in a "proof of concept" available at `http://ariadne.melbinger.org`.

Section 7.1 explains the main design decisions behind Ariadne and what the underlying rationales have been. Section 7.2 shows what graphical user interface is presented to the user and what its main components are. Going into details, Section 7.3 describes the implementation of mazes and avatars, Section 7.4 explains how the required visual programming capability has been implemented and Section 7.5 illustrates how algorithm execution is performed and visualised.

7.1 Design Decisions

Ariadne is intended to be easily accessible to students in order to minimise the required effort and keep learning motivation up. This can be translated into the following functional requirements (see also Chapter 3):

- Ariadne needs to be accessible locally.
- Ariadne needs to be accessible remotely.
- Ariadne needs to be operating system independent.
- Ariadne needs to run on modern computers without requiring additional installation of libraries or drivers.

7.1.1 Technical Approach

Ariadne is designed as a Web application, i.e. a set of scripts executed within a Web browser. It can be accessed remotely whenever an Internet connection is available or locally after an archive of source files has been downloaded and extracted. Since all required technology is included within Ariadne, no additional installations are needed as long as a modern Web browser is used which supports HTML5 and JavaScript.

This approach follows today's trend to provide online environments. As [FG11] notes, they are "lightweight, just building upon the standard features of regular internet browsers" and "ready to be used". This is beneficial especially for online programming environments which "offer an unprecedented opportunity to make available to students powerful environments that do not require going through complex installation procedures that may go well beyond their skills".

7.1.2 High-Level Architecture

The application uses the React framework[1] to organise its user interface into potentially re-usable components and relies on JavaScript ECMAScript 6 language features[2] to modularise the source code. Ariadne's code is "transpiled" to standard JavaScript code using Babel[3].

Figure 7.1 illustrates the high-level architecture of Ariadne's source code and the main interactions between its components.

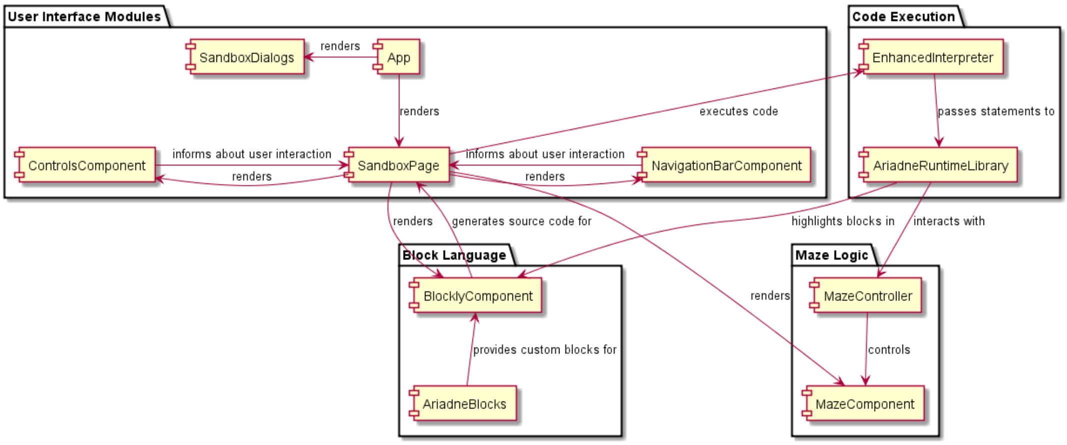


Figure 7.1: High-level architecture of Ariadne

The main entry point for loading Ariadne is the `App` React component which renders the primary (and currently only) page, namely `SandboxPage`.

[1] See `http://facebook.github.io/react`

[2] See `http://es6-features.org`.

[3] See `https://babeljs.io`.

`SandboxPage` is a high-level component that encapsulates the following user interface components:

- `BlocklyComponent` shows the block language editor and generates JavaScript code for live execution of the user-provided block language algorithm.
- `ControlsComponent` shows the code execution control panel.
- `MazeComponent` shows the maze and the avatars that are placed within.
- `NavigationBarComponent` shows the navigation bar menu on top of the screen.
- `SandboxDialogsComponent` is used for displaying various dialogs to the user (e.g. "New maze", "Load maze" and "Save maze").

Furthermore, `SandboxPage` instantiates several classes containing Ariadne's business logic:

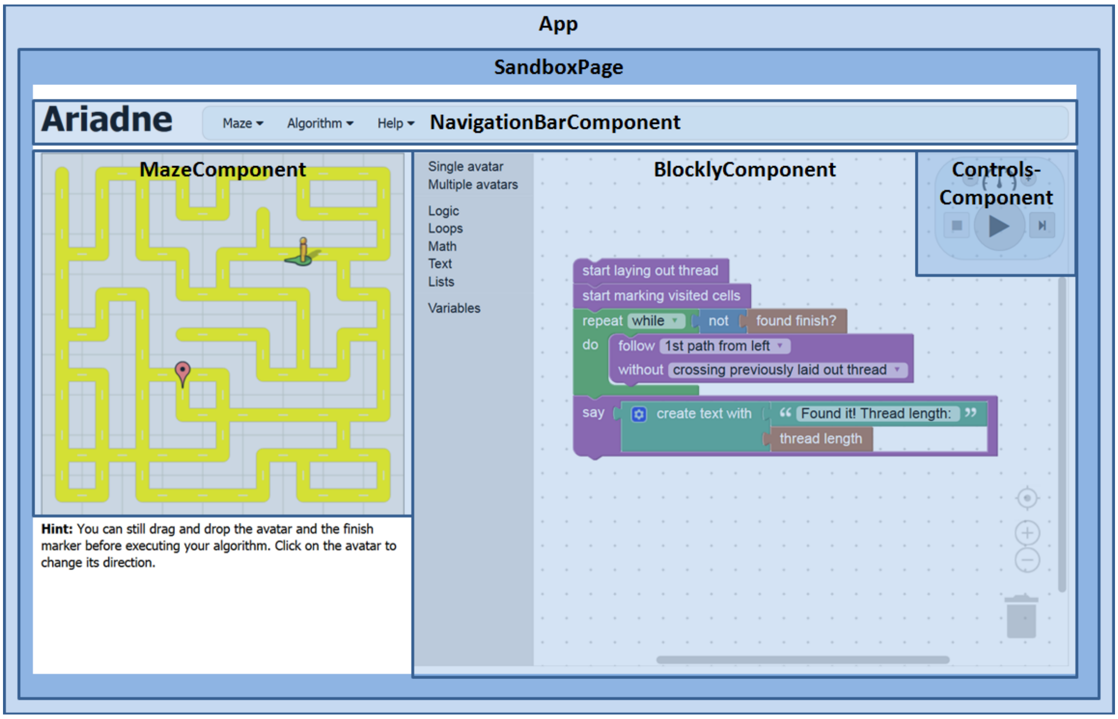
- `AriadneBlocks` contains descriptions of the custom maze programming blocks which are available in `BlocklyComponent`.
- `AriadneRuntimeLibrary` provides the actual implementation of the custom maze programming blocks described by `AriadneBlocks`.
- `EnhancedInterpreter` provides a sandbox for safely executing JavaScript code.
- `MazeController` controls `MazeComponent` and holds meta information about the maze, e.g. position of the avatars placed within and positions of visited cells.

7.2 Graphical User Interface Design

Ariadne's graphical user interface follows the well-known pattern to have a menu bar on top and the main content below it, in this case in two main panels that are aligned side-by-side as shown in Figure 7.2.

A guiding principle for this very simple design was the insight that the presentation of educational content should be reduced to the essential parts and that the less information is presented, the better it can be processed by the learner [FT03]. This helps an environment to prevent crossing the so-called "Deutsch limit" which roughly states "that you can't have more than 50 visual primitives on the screen at the same time" [Beg96] – a practical barrier for programming highly complex systems in visual programming languages.

The following two subsections describe `NavigationBarComponent` and `Sandbox-Dialogs` which are embedded within the `SandboxPage` wrapper.



```
start laying out thread
start marking visited cells
repeat while not found finish?
do  follow 1st path from left
    without crossing previously laid out thread
say create text with " Found it! Thread length: "
                     thread length
```

Hint: You can still drag and drop the avatar and the finish marker before executing your algorithm. Click on the avatar to change its direction.

Figure 7.2: Ariadne's graphical user interface and its main components

7.2.1 Navigation Bar

The navigation bar on top of the screen is rendered by `NavigationBarComponent` and designed to appear familiar to virtually all students. It can be used to execute commands from various categories.

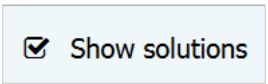
The first navigation bar menu is called "Maze" (see Figure 7.3) and offers the following commands:

- **New** opens the "New maze" dialog.

- **Load** opens the "Load maze" dialog.

- **Save** opens the "Save maze" dialog.

- **Show solutions** toggles the graphical display of all shortest solutions within the maze. See Section 7.3 for information on the implementation of this feature. Note that the toggle state is displayed graphically. If the solutions are currently shown, a checkmark is displayed within the navigation bar menu:

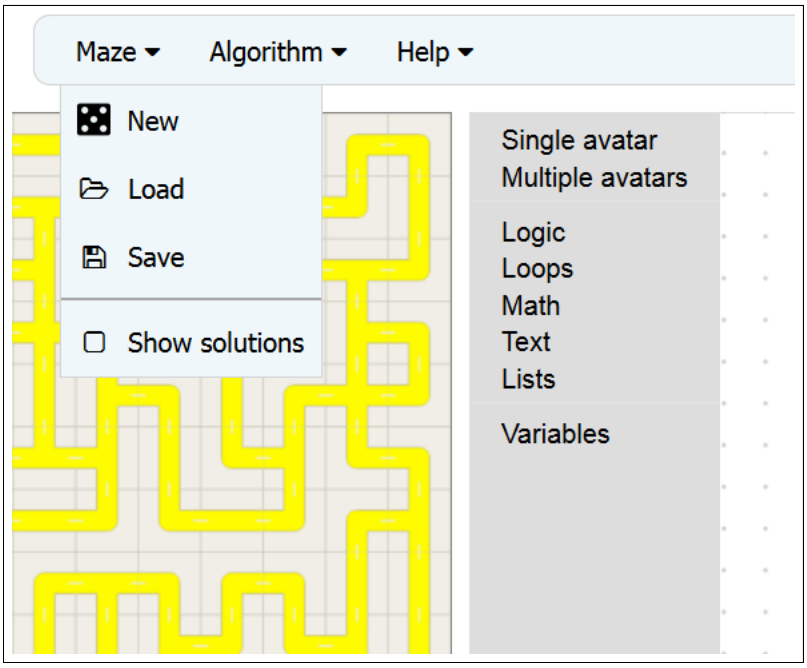


Figure 7.3: The “Maze” navigation bar menu

The second navigation bar menu is called “Algorithm” (see Figure 7.4) and offers the following commands:

- **Clear** removes all block language code from the visual programming editor.
- **Load** opens the “Load algorithm” dialog.
- **Save** opens the “Save algorithm” dialog.
- **Load Random Walk Algorithm**, **Load Left Wall Algorithm**, **Load Simple Ariadne Thread Algorithm**, **Load Optimised Ariadne Thread Algorithm** and **Load Breadth-First Algorithm** remove all current block language code and replace it with the requested pre-defined algorithm. The block language implementations of Ariadne’s maze solving algorithms are specified in Section 7.4.3.

The last navigation bar menu is called “Help” (see Figure 7.5) and offers the following commands:

- **Help** opens the “Help” dialog.
- **About** opens the “About“ dialog.

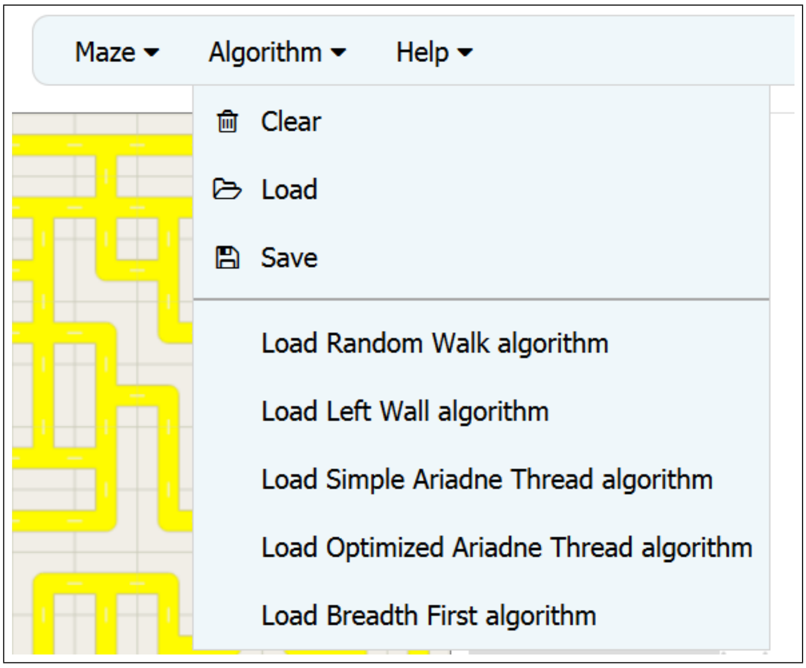

Maze
Algorithm
Help
Clear
Load
Save
Load Random Walk algorithm
Load Left Wall algorithm
Load Simple Ariadne Thread algorithm
Load Optimized Ariadne Thread algorithm
Load Breadth First algorithm

Figure 7.4: The “Algorithm” navigation bar menu

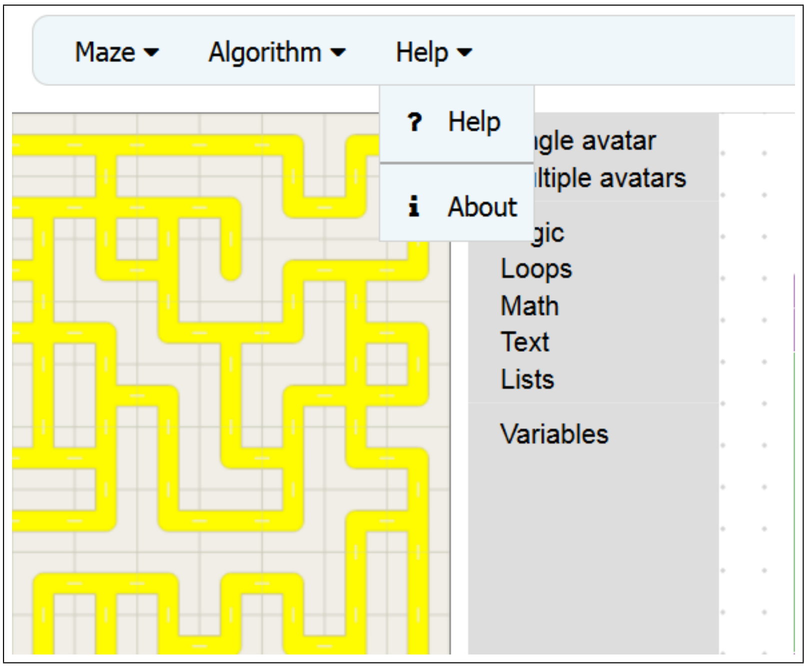
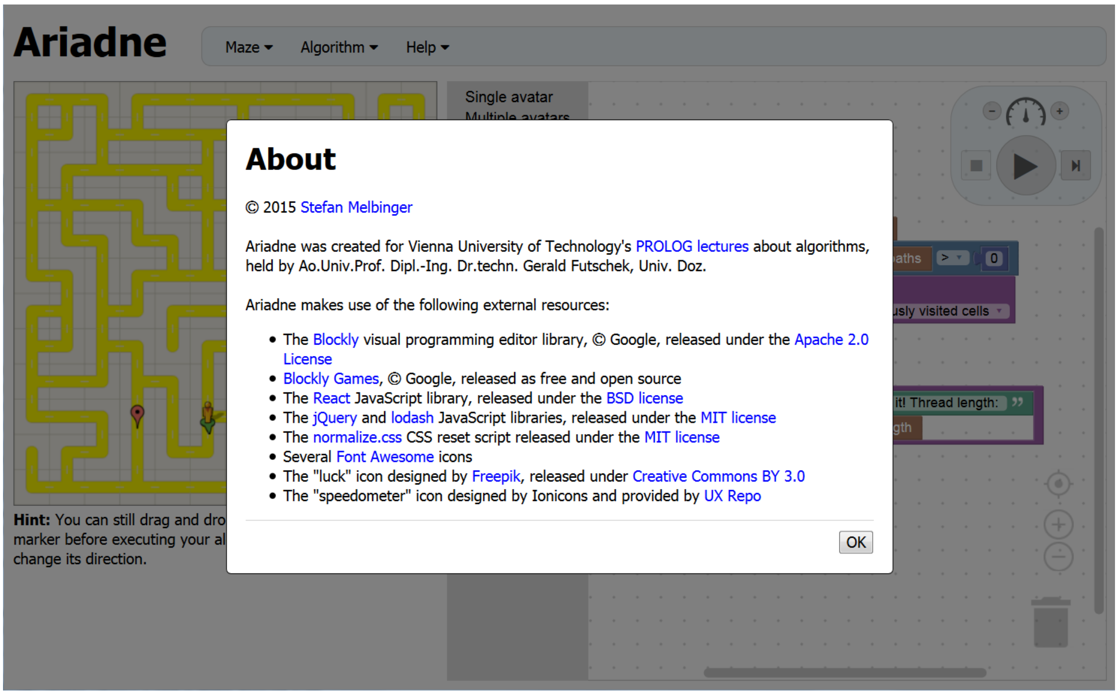


Figure 7.5: The “Help” navigation bar menu

7.2.2 Dialogs

The component `SandboxDialogs` was not shown in Figure 7.2 because it spans the whole screen and is invisible by default. When the user has selected an entry from `NavigationBarComponent` that triggers a dialog, `SandboxDialogs` becomes active, greying out the whole screen and displaying the requested dialog in the centre. As an example, see Figure 7.6 which shows the *About* dialog.



About

© 2015 Stefan Melbinger

Ariadne was created for Vienna University of Technology's PROLOG lectures about algorithms, held by Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek, Univ. Doz.

Ariadne makes use of the following external resources:

- The Blockly visual programming editor library, © Google, released under the Apache 2.0 License
- Blockly Games, © Google, released as free and open source
- The React JavaScript library, released under the BSD license
- The jQuery and lodash JavaScript libraries, released under the MIT license
- The normalize.css CSS reset script released under the MIT license
- Several Font Awesome icons
- The "luck" icon designed by Freepik, released under Creative Commons BY 3.0
- The "speedometer" icon designed by Ionicons and provided by UX Repo

OK

Hint: You can still drag and dro marker before executing your al change its direction.

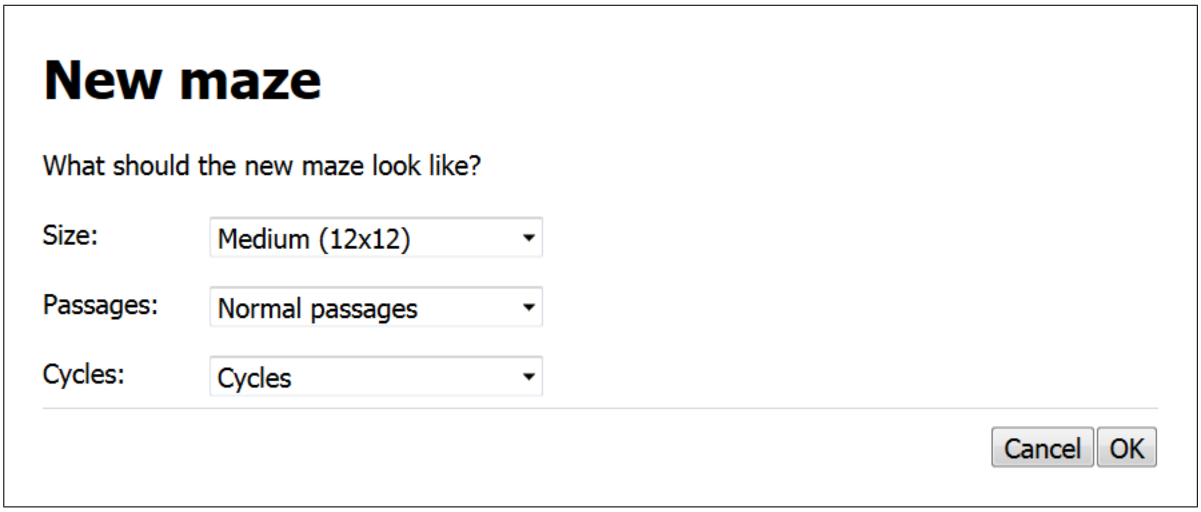
Figure 7.6: The *About* dialog

The following dialogs can be shown:

- The **New maze** dialog as seen in Figure 7.7 lets the user create a new maze according to the specified settings:
  - The *size* of the maze may be chosen from the following values:
    - X-Small (6 × 6)
    - Small (9 × 9)
    - Medium (12 × 12)
    - Large (15 × 15)
    - X-Large (21 × 21)
  - The style of the *passages* determines whether the generated maze will have “normal passages” or “long passages”.

  - The *cycles* field determines whether the generated maze will be a perfect maze (option “no cycles”) or a partial braid maze (option “cycles”).

The initial avatar, its direction and the finish position are chosen randomly. For more information about the maze generation and braiding algorithm used by Ariadne please refer to Chapter 5.



New maze

What should the new maze look like?

Size: Medium (12x12)

Passages: Normal passages

Cycles: Cycles

Figure 7.7: The *New maze* dialog

- The **Load maze** dialog as seen in Figure 7.8 allows the user to restore a previously saved maze (including its avatar and finish) by pasting an encoded maze into the textbox. The maze must be encoded using JSON[4] (as returned by the *Save maze* dialog).

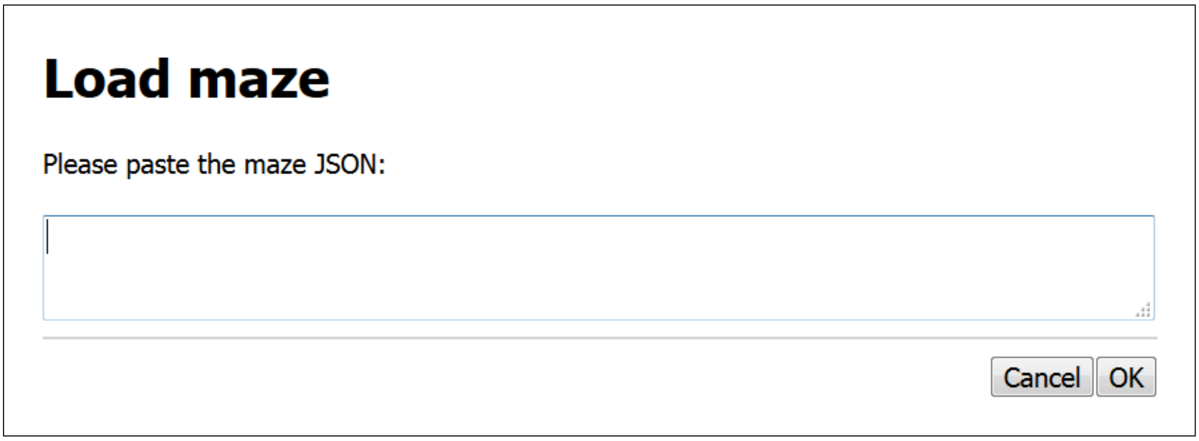
- The **Save maze** dialog as seen in Figure 7.9 encodes the maze, its avatar and finish into the JSON format. The user may save this description to a local file and later restore the maze using the *Load maze* dialog.

- The **Load algorithm** dialog as seen in Figure 7.10 lets the student restore a previously saved algorithm (i.e. its visual programming blocks) by pasting an XML-encoded algorithm (as generated by the *Save Algorithm* dialog) into the textbox.

- The **Save algorithm** dialog as seen in Figure 7.11 returns an XML description of the algorithm including the used blocks and their exact placement within the visual programming component `BlocklyComponent`.

- The **Help** dialog as seen in Figure 7.12 hints students at the very first steps that need to be taken in order to get an example algorithm loaded and running.

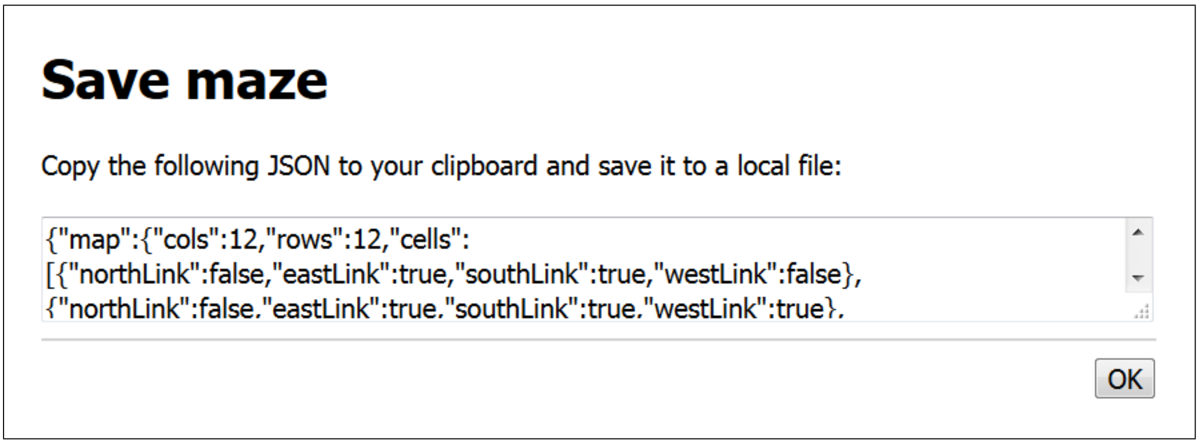
[4] JavaScript Object Notation, see `http://www.json.org`



Load maze

Please paste the maze JSON:

Figure 7.8: The *Load maze* dialog

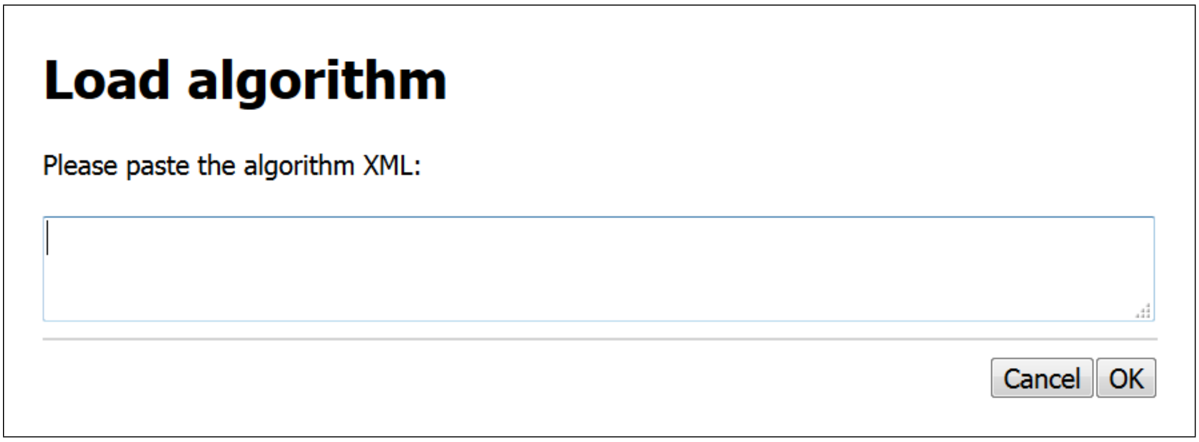


Save maze

Copy the following JSON to your clipboard and save it to a local file:

```
{"map":{"cols":12,"rows":12,"cells":
[{"northLink":false,"eastLink":true,"southLink":true,"westLink":false},
{"northLink":false,"eastLink":true,"southLink":true,"westLink":true},
```

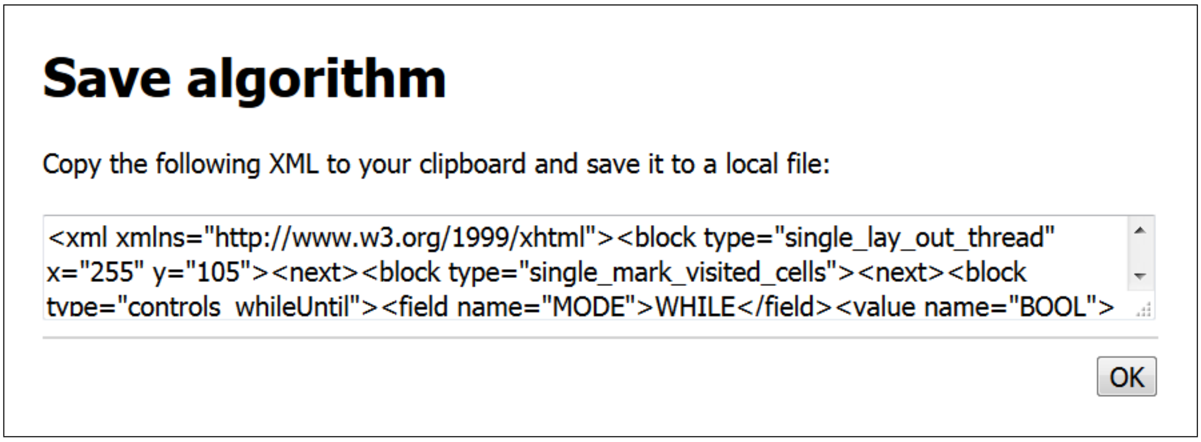
Figure 7.9: The *Save maze* dialog



Load algorithm

Please paste the algorithm XML:

Figure 7.10: The *Load algorithm* dialog

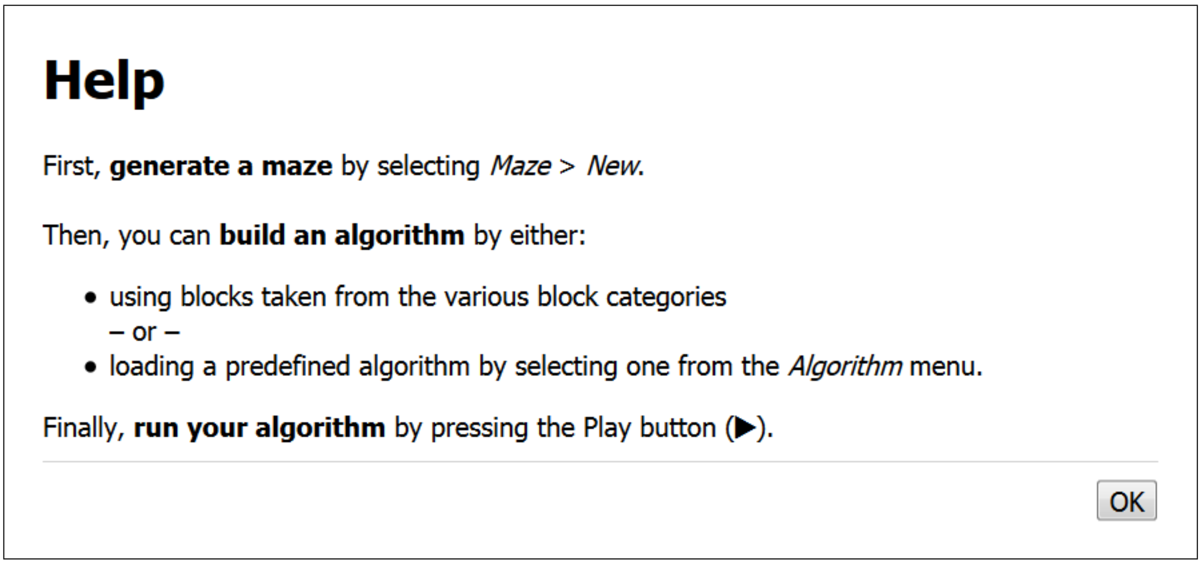


Save algorithm

Copy the following XML to your clipboard and save it to a local file:

```
<xml xmlns="http://www.w3.org/1999/xhtml"><block type="single_lay_out_thread"
x="255" y="105"><next><block type="single_mark_visited_cells"><next><block
type="controls_whileUntil"><field name="MODE">WHILE</field><value name="BOOL">
```

Figure 7.11: The *Save algorithm* dialog



Help

First, **generate a maze** by selecting *Maze* > *New*.

Then, you can **build an algorithm** by either:

- using blocks taken from the various block categories
  – or –
- loading a predefined algorithm by selecting one from the *Algorithm* menu.

Finally, **run your algorithm** by pressing the Play button (▶).

Figure 7.12: The *Help* dialog

- The **About** dialog has already been show in Figure 7.6 at the beginning of this section. It displays information about the Ariadne project including links to the license agreements of various re-used external libraries and resources.

7.3 Mazes and Avatars

Ariadne implements mazes and avatars as a set of JavaScript classes which include business logic (e.g. on how to generate mazes) as well as information on how they need to be rendered by `MazeComponent`.

7.3.1 Data Structures

Mazes can be defined in many ways:

- As "a system of catacombs (in 'real life')" [Fle14].
- As "a set of unit squares in the Euclidean plane" [Fle14], i.e. a grid.
- As a graph, especially because "a grid can be viewed as a special case of a graph" [Pat13]. Graphs may be used to define perfect as well as (partial) braid mazes (see Chapter 5). A perfect maze could even be defined as a specialisation of graphs, namely as a tree, because "it is connected and contains no cycles" [Jár09] which implicates that "there is exactly one path between each pair of vertices" [Fol11].

Ariadne's implementation follows the "[i]ntuitive and wide spread understanding and graphical representation of a maze grid" as a "rectangular shape with orthogonal tessellation", as formulated by [Fol11]. (As the author notes, however, "it is important to keep in mind that this is just one of the potential representations.").

Grid

Ariadne models a maze as cells (instances of class `Cell`) of a grid (instance of class `Grid`). [Kir15] and [BC15] suggest alternative representations, e.g. bitfields and sets of edges and nodes, but Ariadne's implementation appears to be far easier to understand and delivers a performance that is more than sufficient for the project's needs. Listing 7.1 shows the most relevant parts of the implementation of `Grid`.

Please note this listing, just like the following ones, has been slightly simplified in order to increase readability and that the external library *lodash*[5] (referred to in the source code as the underscore symbol '_') is used when native JavaScript does not offer the required functionality.

```
import _ from 'lodash';
import Cell from './Cell';

```

[5] See `https://lodash.com`

```
export default class Grid {

    // [...]

    /**
     * Instantiates a new Grid.
     *
     * @param {number} cols
     * @param {number} rows
     */
    constructor(cols, rows) {
        this._cols = cols;
        this._rows = rows;
        this.clear();
    }

    /**
     * Clears a grid by setting up a new array of unlinked cells.
     */
    clear() {
        this._cells = [];

        let x, y;
        for (y = 0; y < this._rows; y++) {
            for (x = 0; x < this._cols; x++) {
                this._cells.push(new Cell(x, y));
            }
        }

        for (y = 0; y < this._rows; y++) {
            for (x = 0; x < this._cols; x++) {
                let cell = this.getCell(x, y);

                if (y > 0)
                        cell.setNeighbour(’N’, this.getCell(x    , y - 1));

                if (x < this._cols - 1)
                        cell.setNeighbour(’E’, this.getCell(x + 1, y    ));

                if (y < this._rows - 1)
                        cell.setNeighbour(’S’, this.getCell(x    , y + 1));

                if (x > 0)
                        cell.setNeighbour(’W’, this.getCell(x - 1, y    ));
            }
        }
    }

    /**
     * Returns the cell on the given location.
     *
     * @param {number} x
     * @param {number} y
     * @returns {Cell}
     */
    getCell(x, y) {
        return this._cells[y * this._cols + x];
    }

    /**
     * Returns a cell’s array index.
     *
     * @param {Cell} cell
     * @returns {number}
     */
    getCellIndex(cell) {
        return cell.getY() * this._cols + cell.getX();
    }

    /**
     * Returns all cells.
     *
     * @returns {Cell[]}
     */
    getCells() {
        return this._cells;
    }

    /**
     * Returns the width of the grid.
     *
     * @returns {number}
     */
    getColumnCount() {
        return this._cols;
    }

    /**
     * Returns the height of the grid.
     *
     * @returns {number}
     */
    getRowCount() {
        return this._rows;
    }

    /**
     * Returns a random cell out of the grid.
     *
     * @returns {Grid}
     */
    getRandomCell() {
        return _.sample(this._cells);
    }

    // [...]
```

```

}
```

Listing 7.1: `Grid.js`

Note how `constructor()` and `clear()` work together to create an empty grid of cells which are stored in a flat array. Each cell knows its immediate neighbours to the north, east, south and west. A grid, its cells and their cell indices are illustrated in Figure 7.13.

	x = 0	x = 1	x = 2	x = 3	x = 4	x = 5
y = 0	Cell Index 0	Cell Index 1	Cell Index 2	Cell Index 3	Cell Index 4	Cell Index 5
y = 1	Cell Index 6	Cell Index 7	Cell Index 8	Cell Index 9	Cell Index 10	Cell Index 11
y = 2	Cell Index 12	Cell Index 13	Cell Index 14	Cell Index 15	Cell Index 16	Cell Index 17
y = 3	Cell Index 18	Cell Index 19	Cell Index 20	Cell Index 21	Cell Index 22	Cell Index 23
y = 4	Cell Index 24	Cell Index 25	Cell Index 26	Cell Index 27	Cell Index 28	Cell Index 29
y = 5	Cell Index 30	Cell Index 31	Cell Index 32	Cell Index 33	Cell Index 34	Cell Index 35

Figure 7.13: A 6 × 6 `Grid` and its array of `Cell` instances

Cell

As explained above, a grid is made out of many cells. When there is a path between two neighbouring cells, they are said to be “linked”. Listing 7.2 shows the most relevant parts of the implementation of `Cell`.

```
import _ from 'lodash';

export default class Cell {

    // [...]

    /**
     * Instantiates a new Cell.
     *
     * @param {number} x
     * @param {number} y
     */
    constructor(x, y) {
        this._x = x;
        this._y = y;

        this._north = null;
        this._east = null;
        this._south = null;
        this._west = null;

        this._links = [];
    }

    /**
     * Returns the direction to a neighbouring cell.
     *
     * @param {Cell} cell
     * @returns {number} The direction in degrees.
     */
    getDirectionTo(cell) {
        if (cell == this._north) return 0;
        if (cell == this._east)  return 90;
        if (cell == this._south) return 180;
        if (cell == this._west)  return 270;
    }

    /**
     * Returns the neighbouring cell in the specified direction,
     * or null if there is no neighbour there.
     *
     * @param {number} direction The direction in degrees.
     * @returns {Cell|null}
     */
    getLinkedNeighbourInDirection(direction) {
        switch (direction) {
            case 0:
                if (this._north && this.isLinkedTo(this._north))
                    return this._north;
                break;

            case 90:
                if (this._east  && this.isLinkedTo(this._east))
                    return this._east;
                break;

            case 180:
                if (this._south && this.isLinkedTo(this._south))
                    return this._south;
                break;

            case 270:
                if (this._west  && this.isLinkedTo(this._west))
                    return this._west;
                break;
        }

        return null;
    }

    /**
     * Returns all linked neighbours.
     *
     * @returns {Cell[]}
     */
    getLinkedNeighbours() {
        let linkedNeighbours = [];

        if (this._north && this.isLinkedTo(this._north))
            linkedNeighbours.push(this._north);

        if (this._east  && this.isLinkedTo(this._east))
            linkedNeighbours.push(this._east);

        if (this._south && this.isLinkedTo(this._south))
            linkedNeighbours.push(this._south);

        if (this._west  && this.isLinkedTo(this._west))
            linkedNeighbours.push(this._west);

        return linkedNeighbours;
    }

    /**
     * Returns all neighbours.
     *
     * @returns {Cell[]}
     */
    getNeighbours() {
        let neighbours = [];
        if (this._north) neighbours.push(this._north);
        if (this._east)  neighbours.push(this._east);
        if (this._south) neighbours.push(this._south);
        if (this._west)  neighbours.push(this._west);
        return neighbours;
    }

    /**
     * Returns the orientation towards a neighbouring cell.
     *
     * @param {Cell} cell
     * @returns {string}
     */
    getOrientationTo(cell) {
        if (cell == this._north) return ’N’;
        if (cell == this._east)  return ’E’;
        if (cell == this._south) return ’S’;
        if (cell == this._west)  return ’W’;
    }

    /**
     * Encodes the cell’s passages to its neighbour cells.
     *
     * @returns {string}
     */
    getTileIndex() {
        return (
            (this._north && this.isLinkedTo(this._north) ? ’1’ : ’0’) +
            (this._east  && this.isLinkedTo(this._east)  ? ’1’ : ’0’) +
            (this._south && this.isLinkedTo(this._south) ? ’1’ : ’0’) +
            (this._west  && this.isLinkedTo(this._west)  ? ’1’ : ’0’)
        );
    }

    /**
     * Returns all neighbours that are not linked to this cell.
     *
     * @returns {Cell[]}
     */
    getUnlinkedNeighbours() {
        let unlinkedNeighbours = [];

        if (this._north && !this.isLinkedTo(this._north))
            unlinkedNeighbours.push(this._north);

        if (this._east  && !this.isLinkedTo(this._east))
            unlinkedNeighbours.push(this._east);

        if (this._south && !this.isLinkedTo(this._south))
            unlinkedNeighbours.push(this._south);

        if (this._west  && !this.isLinkedTo(this._west))
            unlinkedNeighbours.push(this._west);

        return unlinkedNeighbours;
    }

    /**
     * Returns the x coordinate within the grid.
     *
     * @returns {number}
     */
    getX() {
        return this._x;
    }

    /**
     * Returns the y coordinate within the grid.
     *
     * @returns {number}
     */
    getY() {
        return this._y;
    }

    /**
     * Returns true if the cell has only one linked neighbour.
     *
     * @returns {boolean}
     */
    isDeadend() {
        return this._links.length == 1;
    }

    /**
     * Returns true if the cell has any linked neighbours.
     *
     * @returns {boolean}
     */
    isLinked() {
        return this._links.length > 0;
    }

    /**
     * Returns true if the cell is linked to the specified other cell.
     *
     * @param {Cell} cell
     * @returns {boolean}
     */
    isLinkedTo(cell) {
        return _.contains(this._links, cell);
    }

    /**
     * Links this cell to the specified other cell and vice versa.
     *
     * @param {Cell} cell
     */
    linkTo(cell) {
        if (!_.contains(this._links, cell)) this._links.push(cell);
        if (!_.contains(cell._links, this)) cell._links.push(this);
    }

    /**
     * Informs this cell about its neighbour in the specified orientation.
     *
     * @param {string} orientation
     * @param {Cell} cell
     */
    setNeighbour(orientation, cell) {
        switch (orientation) {
            case ’N’:
                this._north = cell;
                break;

            case ’E’:
                this._east = cell;
                break;

            case ’S’:
                this._south = cell;
                break;

            case ’W’:
                this._west = cell;
                break;
        }
    }

    // [...]

}
```

Listing 7.2: `Cell.js`

The list below presents a few insights on the `Cell` source code:

- The `constructor()` instantiates a cell which is not yet aware of its neighbours and has no links to other cells. `Grid::clear()` uses `setNeighbour()` to correctly set up a grid of cells.

- `getDirectionTo()` and `getOrientationTo()` determine a cell’s position in relation to a given neighbouring cell’s position, returning either an angle of 0°, 90°, 180° or 270° or one of the values ‘N’, ‘E’, ‘S’ or ‘W’, respectively. Each method is best suited for a specific context – working with angles allows easily calculating rotations, while the orientations north, east, south and west ensure unambiguousness and improved understandability of the source code.

- `getLinkedNeighbours()` returns an array of all of a cell’s linked neighbours, while `getUnlinkedNeighbours()` returns an array of all of a cell’s neighbours that cannot be reached directly. `getNeighbours()` returns all neighbouring cells without any further condition.

- `getTileIndex()` encodes information about a cell and the passages to its neighbours. The result is used to determine which kind of passage or junction needs to be displayed when rendering the cell.

- `isDeadend()` determines whether the cell only has one linked neighbour, which would make it a dead-end. `isLinked()` checks whether the cell is linked to any other cell at all. isLinkedTo() determines whether the cell is linked to another specific cell.

Overlay

Avatars, the starting and the finish point are all modelled as classes that all derive from the base class Overlay which contains fields like x and y position, as well as shape width and shape height. These attributes are required for rendering the entities on top of the maze.

7.3.2 Maze Generation

Listing 7.3 shows the source code responsible for generating mazes.

```
import _ from ’lodash’;

export default class Grid {

    // [...]

    /**
     * Generates a maze.
     *
     * @param {number} cols
     * @param {number} rows
     * @param {boolean} braid
     * @returns {Grid}
     */
    static generate(cols, rows, mode, braid) {
        let grid = new Grid(cols, rows);

        let active = [grid.getRandomCell()];

        while (active.length > 0) {
            let cell;

            switch (mode) {
                case ’LAST’:
                    // Always select the most recently added member.
                    cell = _.last(active);
                    break;

                case ’MIX’:
                    // Select the most recently added cell half the time
                    // and a random member the other times.
                    cell = Math.floor(Math.random() * 2) == 0 ?
                        _.sample(active) :
                        _.last(active);
                    break;
            }

            // Select all neighbours that have no links yet,
            // i.e. that have not been visited yet.
            let unvisitedNeighbours = _.filter(
                    cell.getNeighbours(),
                    (neighbourCell) => !neighbourCell.isLinked()
            );

            if (unvisitedNeighbours.length > 0) {
                // Select a random neighbour to link to
                // and add it to the list of active cells.
                let randomUnvisitedNeighbour = _.sample(unvisitedNeighbours);
                cell.linkTo(randomUnvisitedNeighbour);
                active.push(randomUnvisitedNeighbour);
            } else {
                // Nothing more to do with this cell.
                _.pull(active, cell);
            }
        }

        if (braid) grid.braid(0.8);
        return grid;
    }

    /**
     * Performs dead-end culling on a grid.
     *
     * @param {number} p The braiding intensity between 0.0 and 1.0.
     * @returns {Grid}
     */
    braid(p = 1.0) {
        let deadendCells = _.shuffle(
            _.filter(
                this._cells,
                (cell) => cell.isDeadend()
            )
        );

        for (let cell of deadendCells) {
            // Might have been linked in a previous iteration.
            if (!cell.isDeadend()) continue;

            // Take the specified braiding intensity into consideration.
            if (Math.random() > p) continue;

            // Select all neighbours that cell is not already linked to.
            let unlinkedNeighbours = cell.getUnlinkedNeighbours();

            // Prefer to link two dead-end cells together.
            let candidates = _.filter(
                    unlinkedNeighbours,
                    (cell) => cell.isDeadend()
            );

            // If there is no chance to link two dead-end cells together,
            // fall back to all possible neighbour cells again.
            if (candidates.length == 0) candidates = unlinkedNeighbours;

            // Randomly choose one of the potential neighbours to link to.
            cell.linkTo(_.sample(candidates));
        }

        return this;
    }

    // [...]

}
```

Listing 7.3: Maze generation logic in `Grid.js`

While Ariadne's maze generation algorithm has already been introduced formally in Chapter 5, the following notes add some insight to the concrete JavaScript implementation:

- `generate()` is a static method that generates a maze and returns an instance of `Grid`. It accepts the following parameters:
  - `cols`: The requested maze width.
  - `rows`: The requested maze height.
  - `mode`: Determines the order in which the cells are visited. 'LAST' means that the most recently added member of the *active* cells will be selected for inspection while 'MIX' only selects the most recent cell half the time and a random member of the *active* set the rest of the time.

    Different selection criteria result in different maze textures: 'LAST' is used when the user requests "long passages" and 'MIX' is used when the user requests "normal passages".
  - `braid`: Determines whether the resulting maze will be a perfect maze ("no cycles") or a partial braid maze ("cycles"). If a partial braid maze is requested, the algorithm calls `braid()` on the generated (perfect) maze with a braiding intensity value of 0.8 that has proven to produce good looking mazes.
- `braid()` can be called on a grid to perform *dead-end culling* as described in Section 5.2.3. The parameter `p` allows specifying the braiding intensity with a value of 0.0 resulting in no changes to the maze and a value of 1.0 resulting in a maze without any dead-ends.

7.3.3 Visualisation

Canvas Layers

The `MazeComponent` module is responsible for rendering the maze and everything on top of it onto the screen. The following details need to be considered, starting with the bottom layer to the top layer:

- The grid's individual cells need to be rendered with paths in between them.
- The solution paths need to be rendered.
- The avatars' threads need to be rendered.
- The starting point and finish point need to be rendered.
- The avatars need to be rendered.
- The avatar speech bubbles need to be rendered.

The "most widely supported standard for 2D immediate mode graphics on the web" [Smu11] is the `canvas` element that was introduced with HTML5. It allows rendering graphics fast enough to support animations which are "critical to a fun experience" [Lon13].

[Lon13] suggests putting all required images into one image called a "sprite map". This is how Ariadne stores, for example, avatar animation graphics as seen in Figure 7.14. Note that Ariadne's maze graphics have been largely been taken from Blockly Games, "a series of games that teach programming concepts and provides playgrounds to do open-ended programming" [Fra14b].

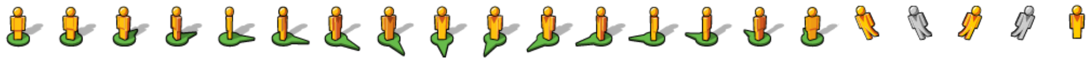


Figure 7.14: The avatar sprite map

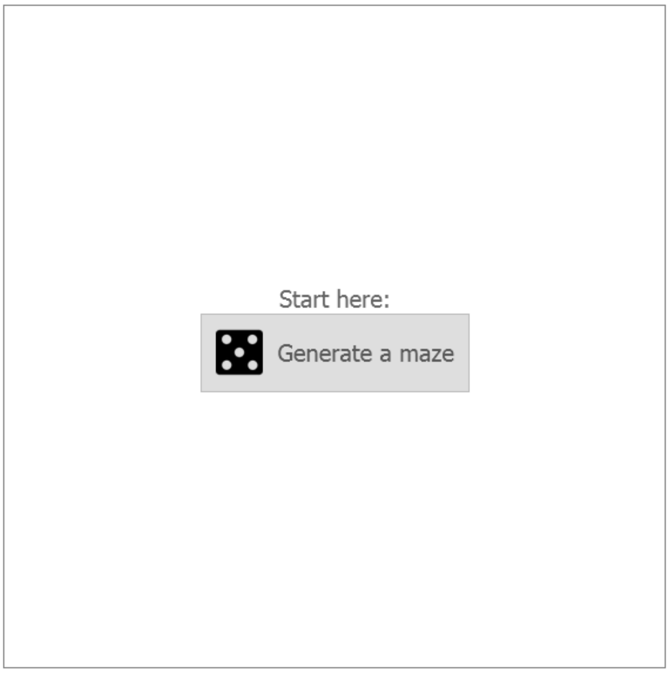
Since some parts of the maze are static and only need to be drawn once while other parts require constant updating (e.g. moving avatars), the implementation of Ariadne follows the advice given by [Smu11] to layer canvases on top of one another: "By using transparency in the foreground canvas, we can rely on the GPU to composite the [alpha transparency values] together at render time." Sprites are therefore drawn onto either the "grid canvas" (used for the underlying grid), the "overlay canvas" (used for avatars, starting and finish point, threads and solution paths) or the "speech canvas" (used for avatar speech bubbles).

Maze Display

On start-up, Ariadne does not automatically generate a maze but presents the user with a quick way to get started as shown in Figure 7.15. The button labelled "Generate a maze" lets the user quickly generate a maze using the default options for a 12 × 12 partial braid maze with normal passages.

Once a maze is loaded, the navigation bar allows the user to toggle the display of all shortest paths (see Section 7.2.1). Figure 7.16 shows how the solution paths are rendered.

In order to find the shortest solution paths, Ariadne performs a breadth-first traversal of the maze as shown in Listing 7.4.



Start here:

Generate a maze

Figure 7.15: Ariadne's maze panel after starting the application

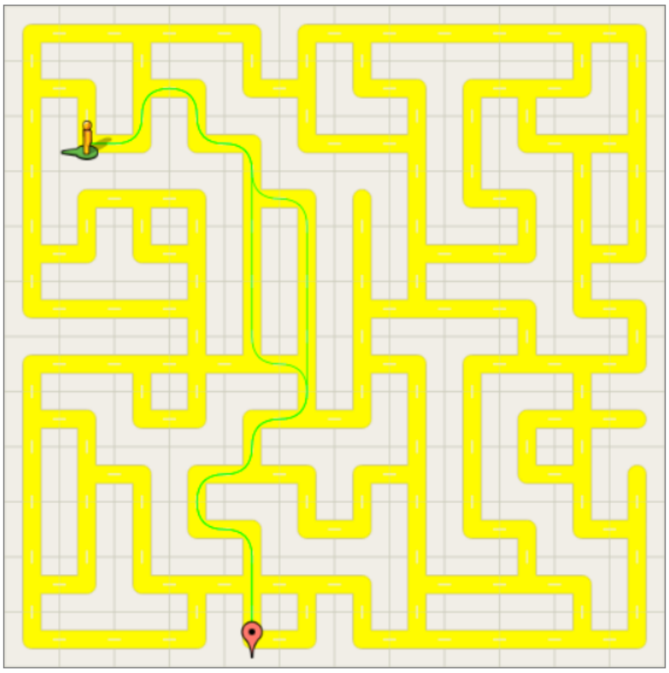


Figure 7.16: A maze with its solutions rendered as green paths

```
import _ from ’lodash’;

export default class Grid {

    // [...]

    /**
     * Finds all shortest path from a starting point to a finish point.
     *
     * @param {Cell} start
     * @param {Cell} finish
     * @returns {[[Cell]]} an array of paths
     */
    findShortestPaths(start, finish) {
        if (!start || !finish) return [];

        let distance = 0;
        let distances = _.fill(new Array(this._cells.length), null);
        let visitedCells = [start];
        let frontier = [start];

        // Keep "flooding" the maze, starting from one point and increasing
        // the radius one by one, until there are no more unvisited cells to
        // examine. The result is an array with each cell’s distance to the
        // starting point.
        while (frontier.length > 0) {
            let newFrontier = [];

            for (let cell of frontier) {
                distances[this.getCellIndex(cell)] = distance;

                let neighbourCells = cell.getLinkedNeighbours();
                for (let unvisitedNeighbourCell of _.difference(
                    neighbourCells,
                    visitedCells
                )) {
                    visitedCells.push(unvisitedNeighbourCell);
                    newFrontier.push(unvisitedNeighbourCell);
                }
            }

            distance++;
            frontier = newFrontier;
        }

        let paths = [[finish]];
        let pathsGrew;

        // Start at the finish and walk backwards onto cells of lower
            distance
        // towards the starting point. At junctions, clone the current path
            and
        // follow each path individually.
        do {
            let newPaths = [];

            pathsGrew = false;
            for (let path of paths) {
                let cell = _.last(path);
                let neighbourCells = cell.getLinkedNeighbours();

                let lowerNeighboursCells = [];
                for (let neighbourCell of neighbourCells) {
                    if (distances[this.getCellIndex(neighbourCell)] <
                        distances[this.getCellIndex(cell)]) {
                        lowerNeighboursCells.push(neighbourCell);
                    }
                }

                if (lowerNeighboursCells.length > 0) {
                    pathsGrew = true;

                    // Take the first one.
                    let firstLowerNeighbourCell =
                        lowerNeighboursCells.shift();

                    // Make new paths for the other lower neighbours cells,
                    // if there are any.
                    for (let lowerNeighbourCell of lowerNeighboursCells) {
                        let newPath = _.clone(path);
                        newPath.push(lowerNeighbourCell);
                        newPaths.push(newPath);
                    }

                    // Append the first lower neighbour cell to the current
                    // path array.
                    path.push(firstLowerNeighbourCell);
                }
            }

            for (let newPath of newPaths) {
                paths.push(newPath);
            }
        } while (pathsGrew);

        return paths;
    }

}
```

Listing 7.4: Determining all shortest solutions in `Grid.js`

Animations

Lots of care has been put into making the avatar animations as fluently as possible. [Smu11] explains the state of the art when it comes to animations on the Web:

> The relatively new `requestAnimationFrame` API is the recommended way of implementing interactive applications in the browser. Rather than command the browser to render at a particular fixed tick rate, you politely ask the browser to call your rendering routine and get called when the browser is available.

Please note that since Ariadne relies on this modern feature to move and rotate the overlays pixel by pixel, it is not compliant with older Internet browsers.

7.4 Algorithm Descriptions

This section describes what framework Ariadne uses to provide a visual programming editor for the user, how the custom blocks are designed and how the maze solving algorithms pre-defined by Ariadne are implemented using these blocks.

7.4.1 Selection and Setup of a Visual Programming Framework

The didactic analysis presented in Chapter 4 suggests that teaching algorithmic thinking profits massively from employing a visual programming language for students to "tinker with". Since Ariadne as a whole shall act as a learning environment, one reasonable approach could be to implement maze algorithms on top of an existing learning environment that includes a visual programming language, such as Scratch.

However, this approach does not appear to be best suited for Ariadne. Scratch scripts "are anchored on game characters called sprites which are perceived as the behaving entities" [MWW12] and are therefore not ideal for describing algorithms which usually are *not* part of a single sprite's implementation but rather a more general part of the implementation – especially when a number of sprites (i.e. avatars) need to be coordinated.

Also, Scratch would require tedious preparation of the environment to be able to display mazes and then require the implementation of custom functions (e.g. "follow path") in order to provide the right level of abstraction and didactic reduction suitable for beginners.

Analysis of Scratch and similar learning environments has quickly shown that forcefully trying to use these systems in a way they were not supposed to be used is not worth the effort and that creating a new learning environment would be necessary. However, re-creating visual programming capabilities is not required. There are a number of frameworks available which allow creation of custom visual programming languages. The best suited candidate proved to be the Blockly library, which has already been presented in Chapter 2. It is…

- …based on client-side Web technology (HTML5 and JavaScript).
- …able to start locally without Internet access.
- …able to generate JavaScript code that can be run within the browser.

- . . . easy to extend with custom blocks.

Ariadne uses `BlocklyComponent` as a wrapper for Blockly-related functionality. However, displaying blocks, managing their interaction and generating code are built-in features of Blockly and little enhancements are necessary.

7.4.2 Design of Custom Blocks

The individual blocks that are required by Ariadne have been determined in Chapter 6. Modelling them for Blockly in the *Block Factory*[6] is an easy task since this tool auto-generates most block layout code for the programmer. Listing 7.5 shows the layout code for the multiple-avatar “found finish?” block as one example.

```
export default class AriadneBlocks {

    // [...]

    /**
     * Loads custom blocks into a Blockly instance.
     *
     * @param {Blockly} blockly
     */
    loadBlocksIntoBlockly(blockly) {

        // [...]

        blockly.Blocks[’multi_found_finish’] = {
            init: function () {
                this.appendDummyInput()
                    .appendField("avatar");
                this.appendValueInput("AVATAR")
                    .setCheck("TYPE_AVATAR");
                this.appendDummyInput()
                    .appendField("found finish?");
                this.setInputsInline(true);
                this.setOutput(true, "Boolean");
                this.setColour(20);
            }
        };

        // [...]

    }

    // [...]

}
```

Listing 7.5: Multiple-avatar “found finish?” block layout in `AriadneBlocks.js`

[6] See `https://blockly-demo.appspot.com/static/demos/blockfactory/index.html`

Loading this programmatic description into Blockly results in a concrete block language construct that is rendered as shown in Figure 7.17.

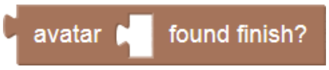


Figure 7.17: Multiple-avatar "found finish?" block display

`AriadneBlocks.js` goes on to define all blocks that are eventually shown to the user. The two categories of blocks ("Single avatar" and "Multiple avatar") are shown in Figure 7.18.

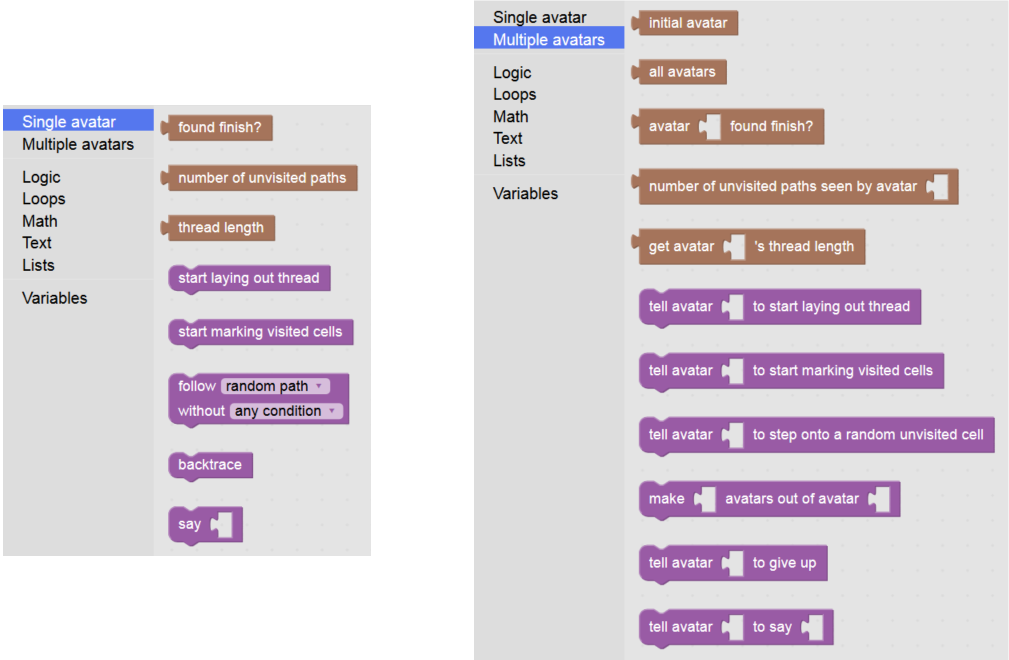
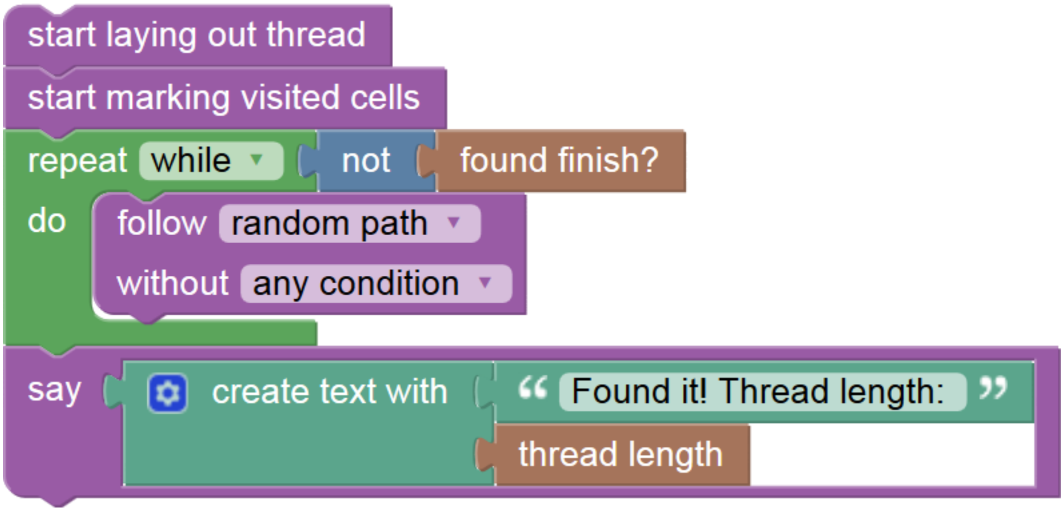


Figure 7.18: Single avatar and multiple avatar blocks

Note that the text inside blocks is written in lower case in order to prepare students for text-based programming languages which usually define keywords in lower case as well. The colours were chosen to allow easy distinction between blocks that execute commands and blocks that return values. Furthermore, the short block descriptions as given in Chapter 6 are shown as tooltips when the user moves the mouse pointer over blocks.

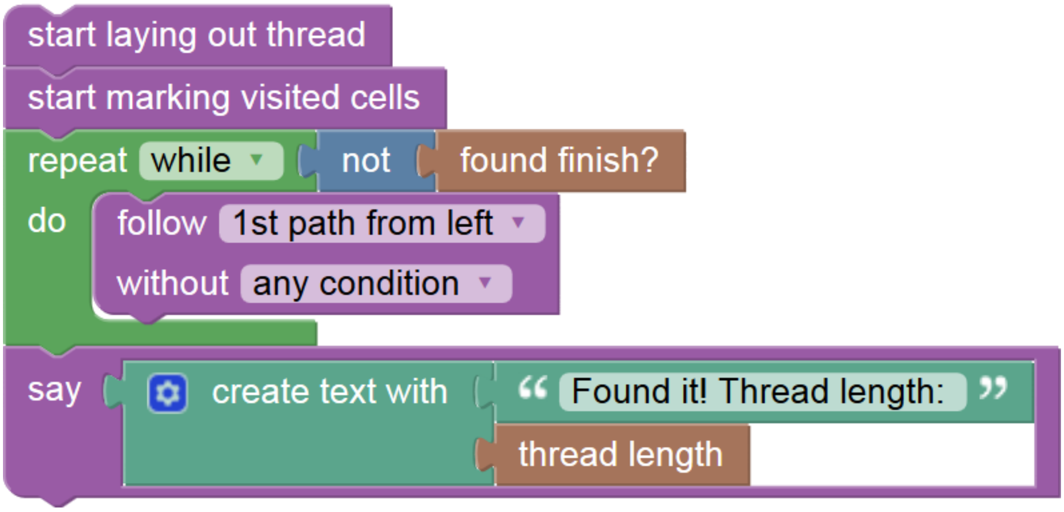
7.4.3 Implementation of Maze Solving Algorithms

Figures 7.19, 7.20, 7.21, 7.22 and 7.23 show how the pre-defined maze solving algorithms in Ariadne are implemented in block language. For the formal description of these algorithms, please refer to Chapter 5.



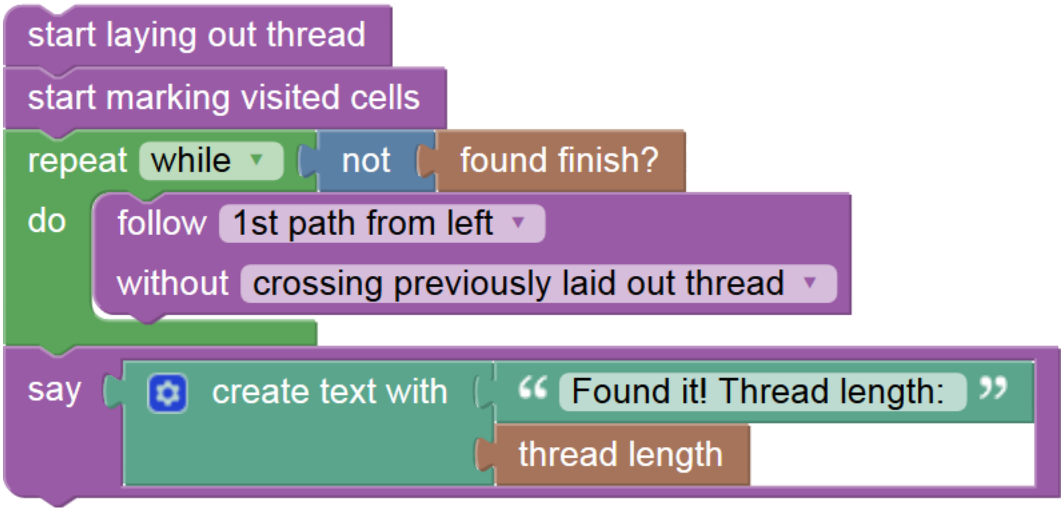
```
start laying out thread
start marking visited cells
repeat while ▾ not found finish?
do  follow random path ▾
    without any condition ▾
say  create text with  " Found it! Thread length: "
                       thread length
```

Figure 7.19: Block language implementation of the Random Walk Algorithm



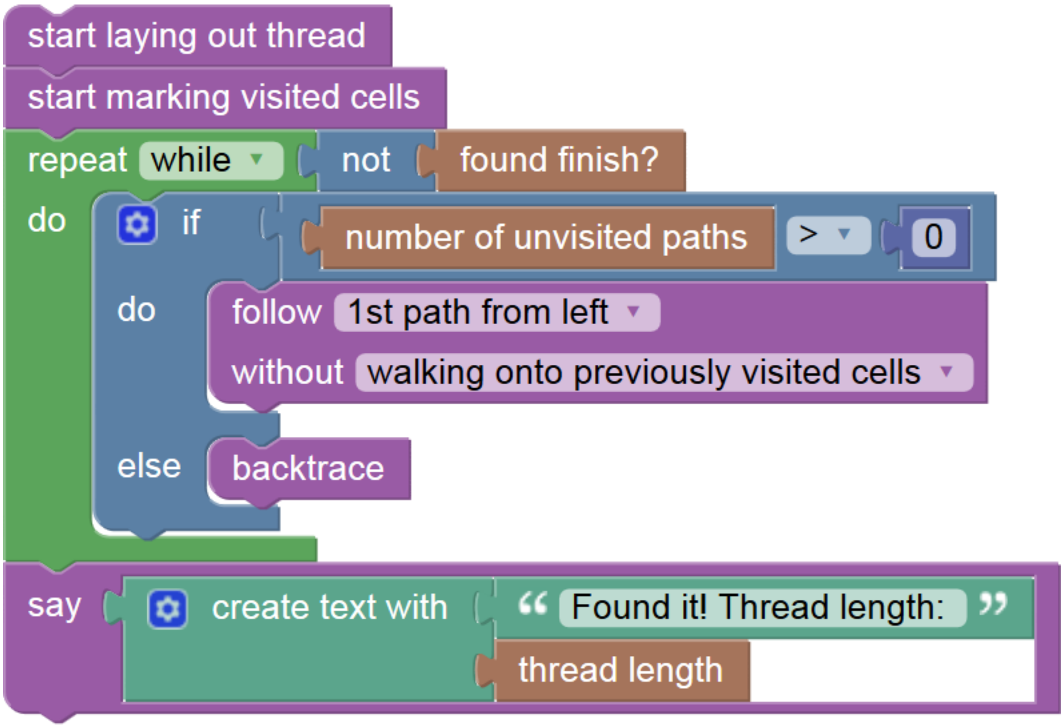
```
start laying out thread
start marking visited cells
repeat while ▾ not found finish?
do  follow 1st path from left ▾
    without any condition ▾
say  create text with  " Found it! Thread length: "
                       thread length
```

Figure 7.20: Block language implementation of the Left Wall Algorithm



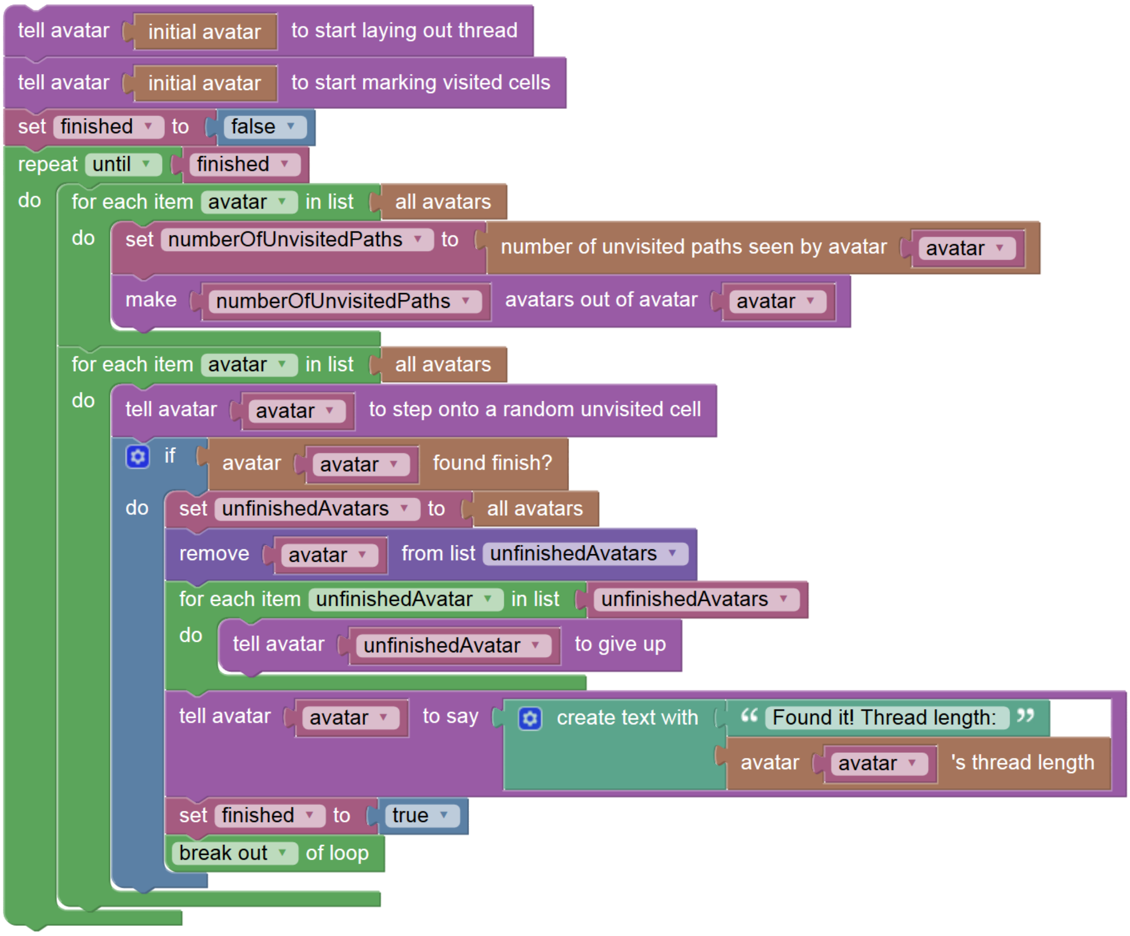
```
start laying out thread
start marking visited cells
repeat while ▾  not  found finish?
do  follow 1st path from left ▾
    without crossing previously laid out thread ▾
say  create text with  “ Found it! Thread length: ”
                       thread length
```

Figure 7.21: Block language implementation of the Simple Ariadne Thread Algorithm



```
start laying out thread
start marking visited cells
repeat while ▾  not  found finish?
do  if  number of unvisited paths  > ▾  0
    do  follow 1st path from left ▾
        without walking onto previously visited cells ▾
    else  backtrace
say  create text with  “ Found it! Thread length: ”
                       thread length
```

Figure 7.22: Block language implementation of the Optimised Ariadne Thread Algorithm



```
tell avatar initial avatar to start laying out thread
tell avatar initial avatar to start marking visited cells
set finished to false
repeat until finished
do  for each item avatar in list all avatars
    do  set numberOfUnvisitedPaths to number of unvisited paths seen by avatar avatar
        make numberOfUnvisitedPaths avatars out of avatar avatar
    for each item avatar in list all avatars
    do  tell avatar avatar to step onto a random unvisited cell
        if  avatar avatar found finish?
        do  set unfinishedAvatars to all avatars
            remove avatar from list unfinishedAvatars
            for each item unfinishedAvatar in list unfinishedAvatars
            do  tell avatar unfinishedAvatar to give up
            tell avatar avatar to say create text with " Found it! Thread length: "
                                                       avatar avatar 's thread length
            set finished to true
            break out of loop
```

Figure 7.23: Block language implementation of the Breadth-First Algorithm

7.5 Algorithm Execution

This section describes Ariadne’s code generation rules, how the generated code is executed and how that execution is visualised.

7.5.1 Code Generation Rules

`AriadneBlocks` contains block design descriptions (as seen before in Section 7.4) as well as code generation rules for Blockly. In order to improve modularisation of the source code and therefore facilitate separation of concerns, these code generation rules do not include the actual business logic that has to be executed once the block itself is executed within an algorithm. Instead, the blocks generate code that calls a runtime library, as seen in Listing 7.6 using the “found finish” block as an example once more.

```
export default class AriadneBlocks {

    // [...]

    /**
     * Loads custom blocks into a Blockly instance.
     *
     * @param {Blockly} blockly
     */
    loadBlocksIntoBlockly(blockly) {

        // [...]

        blockly.JavaScript[’multi_found_finish’] = function (block) {
            let value_avatar = blockly.JavaScript.valueToCode(
                block,
                ’AVATAR’,
                blockly.JavaScript.ORDER_ATOMIC
            );

            return [
                ’__multi_foundFinish(’ + value_avatar + ’)’,
                blockly.JavaScript.ORDER_NONE
            ];
        };

        // [...]

    }

    // [...]

}
```

Listing 7.6: Multiple-avatar “found finish?” code generation rules in `AriadneBlocks.js`

This means that on execution, the “found finish?” block will run `__multi_foundFinish()`.

7.5.2 Sandboxed JavaScript Execution

Code generated by Blockly is not executed within the normal browser runtime context because of potential security and performance risks. Instead, JS-Interpreter[7] is used to parse the generated JavaScript code and execute it within a sandbox that cannot access browser data directly. Accessing the “real world” is only possible through explicitly defined “external API’s”[8].

Continuing with the example of the “found finish?” block, lines 15–23 of Listing 7.7 shows how such an external API is defined for the previously mentioned `__multi_foundFinish()` interface.

```
 1 export default class AriadneRuntimeLibrary {
 2
 3     // [...]
 4
 5     constructor(blocklyComponent, mazeController) {
 6         this._blocklyComponent = blocklyComponent;
 7         this._mazeController = mazeController;
 8     }
 9
10     initializeInterpreter(enhancedInterpreter, jsInterpreter, scope) {
11         this._enhancedInterpreter = enhancedInterpreter;
12
13         // [...]
14
15         jsInterpreter.setProperty(
16             scope,
17             ’__multi_foundFinish’,
18             jsInterpreter.createNativeFunction(
19                 (avatar) => jsInterpreter.createPrimitive(
20                     this.multi_foundFinish(avatar.data)
21                 )
22             )
23         );
24
25         // [...]
26     }
27
28     multi_foundFinish(avatar) {
29         return
30             this._mazeController.locateOverlayCell(avatar) ==
31             this._mazeController.getFinishCell();
32     }
33
34     // [...]
```

[7] See https://neil.fraser.name/software/JS-Interpreter/

[8] Application Programming Interfaces

```
35
36 }
```

Listing 7.7: Multiple-avatar “found finish?” external API definition and business logic in `AriadneRuntimeLibrary.js`

Finally, the external API delegates the call to the actual business logic implementation of the “found finish?” block in lines 28–32. This is where maze and avatar data is analysed and manipulated according to the executed block’s semantics.

7.5.3 Execution Control

JS-Interpreter allows continuously running code at varying speeds, pausing it as well as stepping through it. Ariadne makes use of all these features by providing an execution control panel to the user as shown in Figure 7.24.

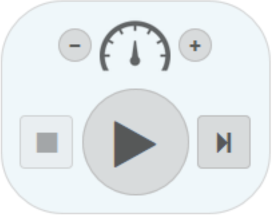


Figure 7.24: The Ariadne execution control panel after start-up

The following bullet points describe the execution control behaviour:

- The *plus* and *minus* signs allow adjusting the execution speed.
- The *play* button starts continuous execution, while the *step* button executes exactly one block before pausing execution.
- During continuous execution, the *step* button is replaced with a *pause* button.
- Pressing the *stop* button stops algorithm execution and resets the maze to its original state (which includes moving the avatar back to its starting position). As soon as the algorithm has terminated, the three bottom control buttons are replaced by a *restart* button which also resets the maze.

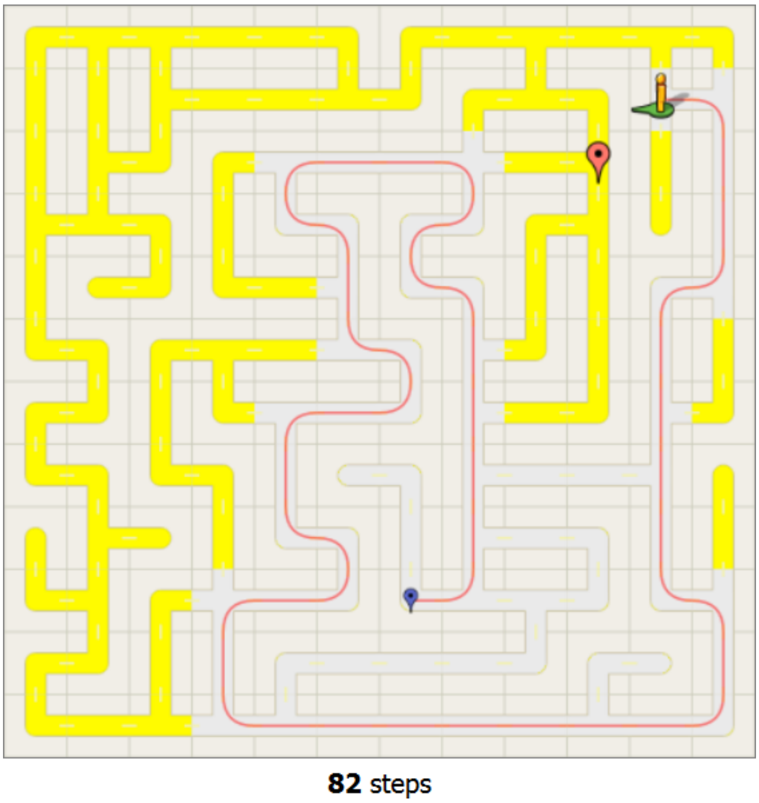
Execution controls are imitating media player controls that are familiar to most people and are intuitive to use.

7.5.4 Execution Visualisation

When a visual programming block’s business logic requests movement of an avatar, `MazeController` receives the command and animates the micro steps necessary to move or rotate an avatar on top of the maze.

While moving, if the avatar has received the “start marking visited cells” command, visited paths are “painted” gray. If the avatar has received the “start laying out thread” command, laid out thread is painted along the path in red colour. Figure 7.25 shows a maze during algorithm execution with both options activated. Note also that the total number of steps taken by the avatar(s) in the maze is printed below the maze grid.

Speech bubbles can be shown with the “say” block and are rendered on top of all other graphics for 3 seconds, as shown in Figure 7.26.



82 steps

Figure 7.25: Marking visited cells and laying out thread during algorithm execution

Additionally to the immediate feedback within the maze, the user of Ariadne also sees the currently executing block highlighted within the Blockly editor, as shown in Figure 7.27. This allows students to immediately connect algorithm formulation and algorithm execution visualisation, as suggested by the didactic analysis in Chapter 4.

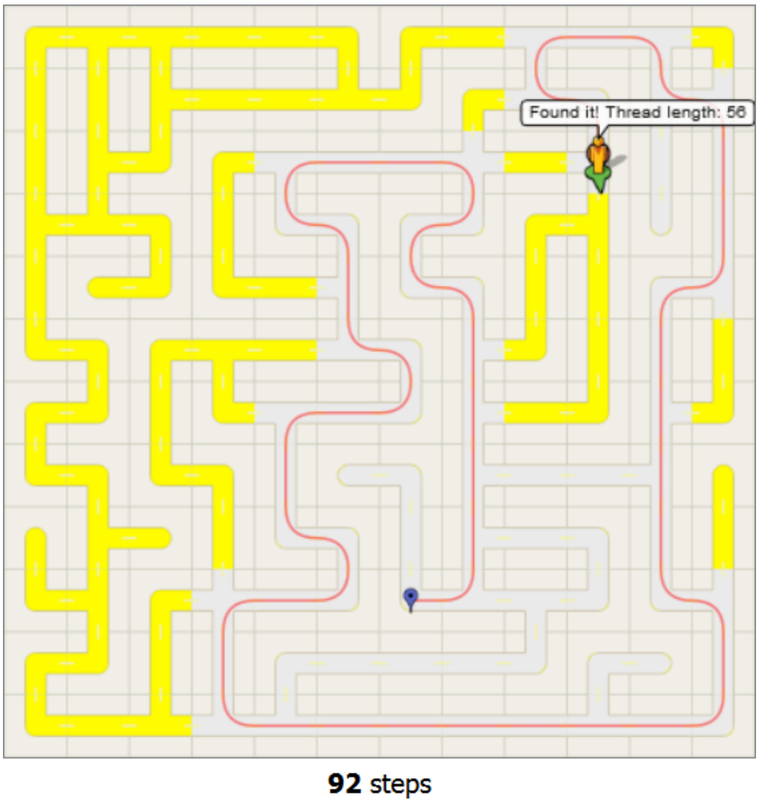
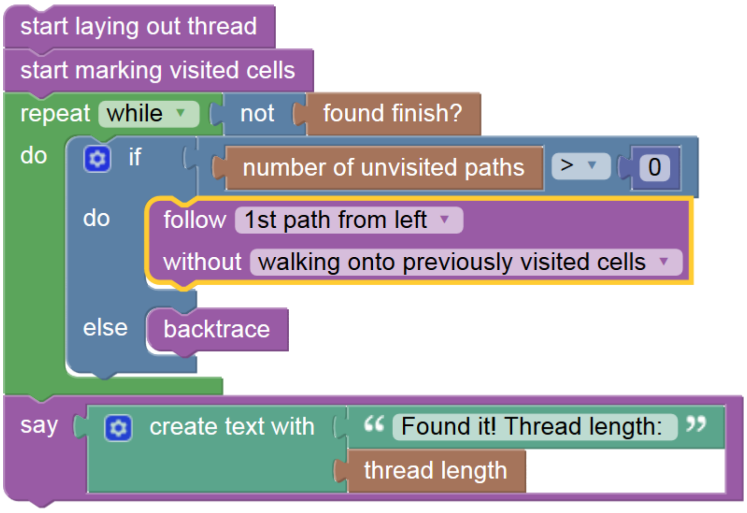


Figure 7.26: Rendering of a speech bubble



```
start laying out thread
start marking visited cells
repeat while not found finish?
do  if number of unvisited paths > 0
    do   follow 1st path from left
         without walking onto previously visited cells
    else backtrace
say create text with " Found it! Thread length: "
                     thread length
```

Figure 7.27: Highlighting of the currently executed block

CHAPTER 8

Evaluation of Results

This chapter evaluates whether the original requirements for Ariadne have been fulfilled (Section 8.1) and hints at possible future development options that have not been realised in the current implementation (Section 8.2).

8.1 Fulfilment of Requirements

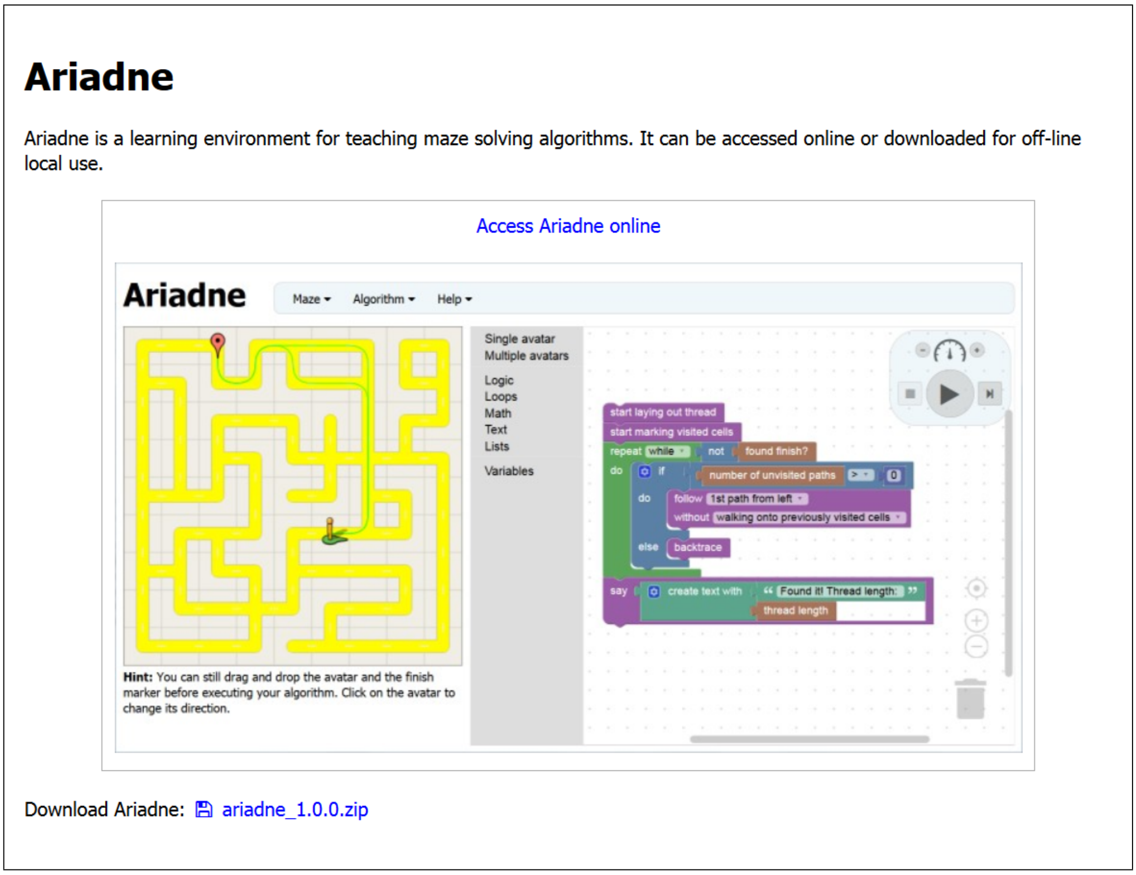
The presented “proof of concept” implementation of Ariadne fulfils all requirements that were stated in Chapter 3 and is ready to be used by students and teachers under the URL `http://ariadne.melbinger.org` as shown in Figure 8.1. It can be accessed online or downloaded for off-line use as a ZIP archive.

Note, however, that the implementation makes use of several modern Internet browser features (e.g. HTML5, CSS3 and `requestAnimationFrame`) and therefore requires an up-to-date client. Ariadne has been successfully tested with Firefox 42, Google Chrome 46 and Internet Explorer 11.

8.2 Future Development

There are a number of improvements that could be made to Ariadne in the future. The following paragraph lists several suggestions for future development in no particular order:

- There might be a need for new custom blocks in order to allow further maze solving algorithms to be developed.
- Ariadne could support a new mode for describing and executing maze generation algorithms. Since these algorithms can be either “passage carvers” or “wall adders” (see Chapter 5), Ariadne would need to be able to provide grids that are either cells without any passages between them or cells that are each connected to all of



Ariadne

Ariadne is a learning environment for teaching maze solving algorithms. It can be accessed online or downloaded for off-line local use.

Access Ariadne online

```
start laying out thread
start marking visited cells
repeat [while] not found finish?
do  if number of unvisited paths [>] 0
    do  follow [1st path from left]
        without [walking onto previously visited cells]
    else  backtrace
say  create text with “ Found it! Thread length: ”
                      thread length
```

Hint: You can still drag and drop the avatar and the finish marker before executing your algorithm. Click on the avatar to change its direction.

Download Ariadne: ariadne_1.0.0.zip

Figure 8.1: The Ariadne Web page

their neighbours. Clearly, a new set of custom blocks would be required as well and the notion of “avatars” might become obsolete in that setting.

- If not only algorithmic thinking but also programming in specific languages is in scope, displaying generated code might be required. The generate code would not need to be the exact code that is executed within the browser but could emulate any programming language desired.

- Sound effects could be added, since they have proven to be very useful “for reinforcing visual views, conveying patterns, replacing visual views, and signaling exceptional conditions” [BH92].

- The possibility to manually edit mazes (i.e. adding and removing passages) by the user could be implemented.

- A tutorial mode might help to introduce the custom blocks one by one.

- Loading and saving mazes and algorithms could make use of more advanced file APIs in modern browsers.

It will be important to use Ariadne in real-world teaching situations and gather feedback from the students. After all, they need to enjoy their learning experience and make actual use of the provided tools.

CHAPTER 9

Conclusion

This thesis has built upon the existing Theseus program which is currently used to support the teaching of algorithmic thinking to budding students of computer science at the Vienna University of Technology. Analysis of its shortcomings has led to a vision for a new online learning environment called Ariadne. Using constructionist learning theory as a foundation to analyse and select didactic options has resulted in the concrete implementation of Ariadne as a web application that can be used online as well as off-line.

Like Theseus, Ariadne features algorithm animation to visualise algorithm execution, i.e. the search for paths within a maze. However, Ariadne adds another important aspect by integrating a visual block programming language for formulating maze solving algorithms, thus allowing students to gain deep understanding of the pre-defined algorithms, to modify them and to even to create new ones from scratch.

Bibliography

[AL02] Kathryn Alesandrini and Linda Larson. Teachers bridge to constructivism. *The Clearing House*, 75(3):118–121, 2002.

[Bae86] Ronald M. Baecker. An applications overview of program visualization. *Computer Graphics*, 20(4):325, 1986.

[Bau96] Rüdeger Baumann. *Didaktik der Informatik*, volume 19962. Klett Stuttgart, 1996.

[BC15] Jamis Buck and Jacquelyn Carter. *Mazes for Programmers.* Pragmatic Programmers, 2015.

[Beg96] Andrew Begel. LogoBlocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA*, 1996.

[BH92] Marc H Brown and John Hershberger. Color and sound in algorithm animation. *Computer*, 25(12):52–63, 1992.

[Bol06] Dietrich Boles. *Programmieren spielend gelernt mit dem Java-Hamster-Modell.* Springer, 3rd edition, 2006.

[Bos14] Mike Bostock. Visualizing algorithms. `http://bost.ocks.org/mike/algorithms/`, 2014. [Online; accessed 21 September 2015].

[Bro88] Marc H Brown. Perspectives on algorithm animation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 33–38. ACM, 1988.

[BS84] Marc H Brown and Robert Sedgewick. *A system for algorithm animation*, volume 18. ACM, 1984.

[CD96] D Cunningham and T Duffy. Constructivism: Implications for the design and delivery of instruction. *Handbook of research for educational communications and technology*, pages 170–198, 1996.

[FG11] Stefano Federici and Elisabetta Gola. BloP: easy creation of online integrated environments to learn custom and standard programming languages. *ATTI DEL CONVEGNO*, page 102, 2011.

[Fle14] Herbert Fleischner. Algorithms in graph theory. 2014.

[Fol11] Martin Foltin. Automated maze generation and human interaction. *Masaryk University Faculty Of Informatics*, 2011.

[Fra12] Neil Fraser. Neil's news: Blockly. `https://neil.fraser.name/news/2012/06/19/`, 2012. [Online; accessed September 17, 2015].

[Fra13] Neil Fraser. Neil's news: Blockly rising. `https://neil.fraser.name/news/2013/12/31/`, 2013. [Online; accessed September 17, 2015].

[Fra14a] Neil Fraser. Blockly Wiki. `https://github.com/google/blockly/wiki`, 2014. [Online; accessed 6 November 2015].

[Fra14b] Neil Fraser. Neil's news: Blockly games. `https://neil.fraser.name/news/2014/08/21/`, 2014. [Online; accessed September 17, 2015].

[Fra15a] Neil Fraser. Blockly FAQ. `https://developers.google.com/blockly/about/faq`, 2015. [Online; accessed 23 September 2015].

[Fra15b] Neil Fraser. Blockly homepage. `https://developers.google.com/blockly/`, 2015. [Online; accessed 23 September 2015].

[Fra15c] Neil Fraser. Blockly language guidelines. `https://developers.google.com/blockly/custom-blocks/language`, 2015. [Online; accessed 5 November 2015].

[Fru06] Tim Fruth. Einstieg in die Programmierung mit dem Java-Hamster-Modell in einer 8. Klasse. 2006.

[FT03] Sebastian Funke and David Tepaße. Ein Konzept zum Einsatz von Struktogrammen als pädagogisches Werkzeug im Informatik-Unterricht. 2003.

[Giu12] Pier Giuliano. An interactive environment for the didactical manipulation of programs. Master's thesis, University of Cagliari, 2012.

[GS14] Bharat Gupta and Smriti Sehgal. Survey on techniques used in autonomous maze solving robot. In *2014 5th International Conference-The Next Generation Information Technology Summit*, pages 323–328. IEEE, 2014.

[HDS02] Christopher D Hundhausen, Sarah A Douglas, and John T Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, 2002.

[Hub13] Peter Hubwieser. *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele.* Springer-Verlag, 2013.

[Jár09] Antal A Járai. The uniform spanning tree and related models. 2009.

[Kin12] Shiva Kintali. How to teach algorithms? https://kintali.wordpress.com/2012/09/24/how-to-teach-algorithms/, 2012. [Online; accessed September 17, 2015].

[Kir15] Bob Kirkland. How to build a maze. http://www.mazeworks.com/mazegen/mazetut/index.htm, 2015. [Online; accessed 18 September 2015].

[KST01] Colleen Kehoe, John Stasko, and Ashley Taylor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *International Journal of Human-Computer Studies*, 54(2):265–284, 2001.

[Lon13] James Long. Making sprite-based games with canvas. http://jlongster.com/Making-Sprite-based-Games-with-Canvas, 2013. [Online; accessed September 17, 2015].

[LP02] Anany Levitin and Mary-Angela Papalaskari. Using puzzles in teaching algorithms. In *ACM SIGCSE Bulletin*, volume 34, pages 292–296. ACM, 2002.

[med14] medica. Stack exchange: Difference between “labyrinth” and “maze”. http://english.stackexchange.com/questions/144052/difference-between-labyrinth-and-maze, 2014. Online; accessed September 17, 2015[Online; accessed September 17, 2015].

[ML07] David J Malan and Henry H Leitner. Scratch for budding computer scientists. In *ACM SIGCSE Bulletin*, volume 39, pages 223–227. ACM, 2007.

[MRR$^+$10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[MWW12] Assaf Marron, Gera Weiss, and Guy Wiener. A decentralized approach for programming interactive applications with JavaScript and Blockly. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 59–70. ACM, 2012.

[NC15] Richard Noss and James Clayson. Reconstructing constructionism. *Constructivist Foundations*, 10(3):285–288, 2015.

[NS73] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *ACM Sigplan Notices*, 8(8):12–26, 1973.

[Pap80] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas.* Basic Books, Inc., 1980.

[Pat13] Amit Patel. Grids and graphs. `http://www.redblobgames.com/pathfinding/grids/graphs.html`, 2013. [Online; accessed September 17, 2015].

[Pul15] Walter Pullen. Maze algorithms. `http://www.astrolog.org/labyrnth/algrithm.htm`, 2015. [Online; accessed 18 September 2015].

[RMMH$^+$09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[SK09] Susanne Schiemann-Koch. Unterrichtsmaterialien zum imperativen Hamster-Modell. `http://www.java-hamster-modell.de/allerlei.html`, 2009. [Online; accessed September 17, 2015].

[Smu11] Boris Smus. Improving HTML5 canvas performance. `http://www.html5rocks.com/en/tutorials/canvas/performance/`, 2011. [Online; accessed September 17, 2015].

[SS11] Sigrid Schubert and Andreas Schwill. *Didaktik der Informatik.* Springer, 2011.

[TD72] Endel Tulving and Wayne Donaldson. *Organization of memory.* 1972.

[Win08] Jeannette M. Wing. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881):3717–3725, 2008.