

JSXGraph-Based Exercises in Moodle STACK

Shahab Abtahi

Suggested citation:

Abtahi, Shahab. 2025. "JSXGraph-Based Exercises in Moodle STACK." Hannover: Hochschule Hannover. <https://doi.org/10.25968/opus-3668>.

Abstract

Digital assessments in graph theory often resort to static images, offering neither interactivity nor variation between students. This thesis closes that gap by extending Moodle's STACK plugin with JSXGraph visualisation and Maxima-based evaluation to deliver fully interactive, randomised exercises for Depth-First Search, Breadth-First Search, Dijkstra's, and Kruskal's algorithms.

The system operates in five steps: (1) teachers specify controllable random-seed parameters; (2) Maxima constructs a tailored graph; (3) the graph is rendered live in the browser via JSXGraph; (4) students manipulate nodes and edges to express their solution; and (5) STACK evaluates the submission with algorithm-aware Maxima scripts, returning immediate feedback.

A unifying input-capture layer means the same workflow supports all four algorithms, and a custom evaluation engine simulates each student's logic to verify correctness. Pilottests with dozens of randomly generated instances confirm reliable grading and highlight the framework's scalability. The work demonstrates how symbolic computation and client-side interactivity can be combined to raise both pedagogical value and assessment fidelity in digital learning environments.

Terms of use

CC BY 4.0

**HOCHSCHULE
HANNOVER**
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
*Fakultät IV
Wirtschaft und
Informatik*

JSXGraph-Based Exercises in Moodle STACK

Shahab Abtahi

Bachelor-Arbeit im Studiengang Mediendesigninformatik

May 5, 2025



Autor Shahab Abtahi
1530453
shahab.abtahi@hotmail.com

Erstprüferin: Prof. Dr. Frauke Sprengel
Abteilung Informatik, Fakultät IV
Hochschule Hannover
frauке.sprengel@hs-hannover.de

Zweitprüfer: Prof. Dr. Volker Ahlers
Abteilung Informatik, Fakultät IV
Hochschule Hannover
volker.ahlers@hs-hannover.de



Except where otherwise noted, content of this report is licensed under a Creative Commons Attribution 4.0 International license. This does not apply for citations and material used under another allowance. To see the conditions of the license, follow <https://creativecommons.org/licenses/by/4.0/>.

Declaration of Authenticity

I declare that I completed the thesis independently without external assistance and used only these materials that are listed. All materials used, from published as well as unpublished sources, whether directly quoted or paraphrased, are duly reported.

If AI tools (as aids) were used, I take full responsibility for the selection, adoption, and all results of the AI-generated components used in this thesis; this applies in particular to any AI-generated plagiarism. In the directory "Overview of tools used," I have listed all AI tools used with their product names.

Hannover, May 5, 2025

Signature

Abstract

Digital assessments in graph theory often resort to static images, offering neither interactivity nor variation between students. This thesis closes that gap by extending Moodle's STACK plugin with JSXGraph visualisation and Maxima-based evaluation to deliver fully interactive, randomised exercises for Depth-First Search, Breadth-First Search, Dijkstra's, and Kruskal's algorithms.

The system operates in five steps: (1) teachers specify controllable random-seed parameters; (2) Maxima constructs a tailored graph; (3) the graph is rendered live in the browser via JSXGraph; (4) students manipulate nodes and edges to express their solution; and (5) STACK evaluates the submission with algorithm-aware Maxima scripts, returning immediate feedback.

A unifying input-capture layer means the same workflow supports all four algorithms, and a custom evaluation engine simulates each student's logic to verify correctness. Pilot tests with dozens of randomly generated instances confirm reliable grading and highlight the framework's scalability. The work demonstrates how symbolic computation and client-side interactivity can be combined to raise both pedagogical value and assessment fidelity in digital learning environments.

Contents

1. Introduction: Motivation and Objectives	8
1.1. Background / Context	8
1.2. Motivation	9
1.3. Problem Statement	10
1.4. Objectives	11
1.5. Approach / Methodology Summary	11
1.6. Structure of the Thesis	12
2. Theoretical Foundations	14
2.1. Learning Platforms and Tools	14
2.1.1. Moodle	14
2.1.2. STACK	15
2.1.3. Maxima	15
2.1.4. JSXGraph	15
2.2. Basics of Graph Theory	16
2.3. Graph Algorithms Relevant to the Project	17
2.3.1. Depth-First Search (DFS)	17
2.3.2. Breadth-First Search (BFS)	17
2.3.3. Kruskal's Algorithm	18
2.3.4. Dijkstra's Algorithm	19
3. Requirements Analysis	21
3.1. User Roles and Interaction Needs	21
3.2. Functional Requirements	21
3.3. Non-Functional Requirements	22
3.4. Use Case Scenarios	23
4. System Design and Implementation	25
4.1. Overview of the Interactive Exercise Generation Process	25
4.2. Parameter Definition	26
4.2.1. Initial Full Graph	26
4.2.2. Vertex Count Definition	28
4.2.3. Edge Density Factor	28
4.2.4. Edge Weights	28

4.3. Graph Construction in Maxima	29
4.3.1. Vertex Selection:	29
4.3.2. Edge Filtering	30
4.3.3. Weight Assignment	33
4.3.4. Matrix Transfer to JavaScript:	35
4.4. Interactive Graph Rendering	35
4.4.1. Initialization of the JSXGraph Board	35
4.4.2. Importing Parameters from Maxima	36
4.4.3. Drawing Vertices	36
4.4.4. Drawing Edges	37
4.4.5. Start Node Initialization	38
4.5. Student Interaction	38
4.5.1. Vertex Interaction	39
4.5.2. Edge Interaction	41
4.6. Answer Submission and Variable Binding	44
4.7. Evaluation	46
4.7.1. Evaluation Strategy for Breadth-First Search (BFS)	46
4.7.2. Evaluation Strategy for Depth-First Search (DFS)	48
4.7.3. Evaluation Strategy for Kruskal’s Algorithm	50
4.7.4. Evaluation Strategy for Dijkstra’s Algorithm	52
5. Results and Conclusion	55
5.1. Summary of Achievements	55
5.2. Verification and Testing	56
5.3. Limitations	57
5.4. Conclusion	58
5.5. Future Work	59
Overview of Tools Used	60
Appendix	60
A. Appendix: Source Code Listings	61
A.1. BFS Algorithm	61
A.1.1. BFS Question Variables	61
A.1.2. BFS Question Text	64
A.1.3. BFS Feedback Variables	69
A.2. DFS Algorithm	70
A.2.1. DFS Question Variables	70
A.2.2. DFS Question Text	73
A.2.3. DFS Feedback Variables	78

A.3. Kruskal Algorithm	79
A.3.1. Kruskal Question Variables	79
A.3.2. Kruskal Question Text	82
A.3.3. Kruskal Feedback Variables	86
A.4. Dijkstra Algorithm	89
A.4.1. Dijkstra Question Variables	89
A.4.2. Dijkstra Question Text	93
A.4.3. Dijkstra Feedback Variables	97
Bibliography	98

List of Figures

2.1. Example of an undirected graph with 7 vertices and 12 edges.	16
4.1. The Initial Full Graph used for generating all question variants.	27
4.2. Visual representation of the filtered graph after edge reduction.	32
4.3. Graph rendering with weighted edges after applying random weight assignment.	34
4.4. Visual appearance of a Dijkstra graph before and after vertex selection. .	41
4.5. Visualization of edge selection sequence in Kruskal's algorithm. The labels on selected edges show both the edge weight and the selection order.	44

1. Introduction: Motivation and Objectives

The digital transformation of education has gained global momentum in recent years, especially in the wake of the COVID-19 pandemic, which significantly accelerated the adoption of digital tools in classrooms. While digital platforms have long supported the management of educational data and content delivery, they are increasingly being recognized for their potential to personalize learning, enhance teaching effectiveness, and inform decision-making across educational systems [OEC23].

The use of digital platforms in education has become standard practice worldwide, particularly in mathematics and science instruction. Platforms such as Moodle provide a foundation for delivering both learning materials and dynamic exercises that support the learning process and assist in evaluating student performance [Moo24]. To enable automated assessment of mathematical problems, the STACK plugin offers a powerful framework for integrating symbolic computation into Moodle [STA24c]. However, certain mathematical topics—especially those involving graphical representations—remain pose challenges for assessment through traditional input methods. For example, exercises in graph theory often require graphical user interface (GUI)-based interaction, which is not supported by the basic functionality of STACK.

This thesis aims to design and implement an interactive solution for graph-based questions by making use of JSXGraph within the STACK environment.

This chapter introduces the context of the project, outlines the motivation and objectives, and provides an overview of the methodology and structure of the thesis.

1.1. Background / Context

Moodle is the digital learning platform used at Hochschule Hannover. It serves as a central system for sharing teaching and learning materials, submitting and managing homework, and facilitating communication between students and instructors. Students can easily access their enrolled courses and participate in various online learning activities [Hoc24].

Moodle is a Learning Management System (LMS) designed to provide educators, administrators, and learners with a secure and flexible environment for personalized digital education. As an open-source platform, it is freely available and widely adopted across institutions worldwide. Moodle supports a wide range of extensions through plugins, enabling it to adapt to various educational scenarios and subject-specific requirements [Moo24].

One of these plugins is STACK (System for Teaching and Assessment using a Computer Algebra Kernel), which allows the creation and automated evaluation of complex mathematical questions within Moodle. STACK is implemented as a question-type plugin, enabling instructors to design tasks that accept algebraic expressions as input and assess them automatically [STA24a]. At Hochschule Hannover, STACK is used within the Moodle environment in Faculty IV, particularly in mathematics and statistics courses. It supports symbolic input, automated feedback, and the generation of dynamic questions involving algebraic expressions.

STACK relies on the computer algebra system (CAS) Maxima to manipulate mathematical expressions symbolically. A CAS can perform operations such as simplification, expansion, and transformation of mathematical expressions. In STACK, Maxima is used to evaluate relevant mathematical properties of students' answers, generate structured random content, and plot mathematical functions [STA24c].

Although Maxima offers basic plotting capabilities, its visualization features are limited to static output. This presents a challenge when creating mathematical exercises, especially in graph theory, that require interactive and user-driven input. Tasks such as constructing graphs, placing nodes, or drawing edges require a level of interactivity that Maxima's native graphical tools do not provide. Its functions are primarily intended for generating visual representations rather than supporting direct manipulation or dynamic user interaction [STA24b].

1.2. Motivation

Digital assessment tools have become increasingly important in mathematics education, offering new opportunities for automation, feedback, and personalized learning experiences. At Hochschule Hannover, the combination of Moodle and the STACK plugin is already used within Faculty IV to evaluate student work in mathematical courses.

However, certain mathematical topics—particularly graph theory—require visual and interactive elements that go beyond traditional symbolic input. At Hochschule Hannover, graph-related tasks are currently implemented using static images or basic drag-and-drop components. These formats do not allow for meaningful interaction with graph

structures, nor do they support algorithm-specific manipulations such as edge selection or traversal sequences.

Furthermore, there is currently no built-in support for parameter randomization in these types of exercises. As a result, all students receive the same static problem instance, which limits opportunities for personalized practice, iterative improvement, and fairness in assessment.

An interactive solution that supports randomized graph generation and dynamic student input would enable more engaging, varied, and pedagogically effective assessments in graph theory. This gap in functionality forms the central motivation for the present work: to develop a system that allows for interactive, randomized graph algorithm tasks directly within Moodle’s STACK environment.

1.3. Problem Statement

In the current implementation of graph theory exercises for the course “Mathematik 1” at Faculty IV of Hochschule Hannover, tasks are typically created using static images combined with Moodle’s built-in drag-and-drop question types. These question types, such as drag-and-drop markers, provide only limited interactivity and do not support meaningful manipulation of graph structures—such as constructing edges, selecting paths, or editing nodes.

Additionally, the current system lacks functionality for parameter randomization and the dynamic generation of graph-based content. This results in all students receiving identical tasks, which reduces variety and limits opportunities for individualized learning and iterative practice with varied inputs.

Although JSXGraph is technically supported within STACK as an optional tool for dynamic visualizations, it is not integrated as a native input mechanism for graph-related tasks. Its usage currently requires manual scripting and lacks built-in support for capturing student interactions or evaluating submitted graph structures.

This thesis addresses these limitations by developing a structured integration of JSXGraph within the STACK environment. The goal is to enable the creation of interactive, randomized graph theory exercises in Moodle, thereby enhancing both engagement and variability in mathematical assessment.

1.4. Objectives

The overarching goal of this thesis is to expand the capabilities of the STACK plugin within Moodle to support interactive, algorithm-driven exercises in graph theory. This involves equipping students with an engaging, visual platform for exploring classical algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), Kruskal's algorithm, and Dijkstra's algorithm.

To achieve this, the project pursues the following specific objectives:

- **Architectural Design:** Develop a reusable and modular system architecture that integrates JSXGraph into STACK, enabling seamless interaction between graphical input and symbolic evaluation.
- **Interactive Input Support:** Implement mechanisms that allow students to interact directly with graph elements—selecting and manipulating nodes and edges in response to algorithmic tasks.
- **Randomization and Variation:** Enable the generation of randomized graph structures and edge weights using Maxima, ensuring each student receives a unique and pedagogically meaningful problem instance.
- **Automated Evaluation:** Design and implement algorithm-specific evaluation scripts in Maxima that validate student submissions for logical correctness, based on traversal order, edge structure, and algorithm-specific criteria.
- **Feedback Integration:** Incorporate real-time feedback using STACK's Potential Response Tree (PRT) framework, including visual indicators of correct or incorrect solutions to support learning through trial and error.
- **Configurability and Compatibility:** Ensure full compatibility with existing Moodle and STACK infrastructure, while allowing question authors to define customizable parameters such as graph size, edge density, and weight range.

Collectively, these objectives aim to enhance digital assessment in graph theory by fostering interactivity, supporting individualized learning experiences, and bridging the gap between visual problem-solving and symbolic reasoning.

1.5. Approach / Methodology Summary

The methodology of this thesis combines software development, algorithm design, and pedagogical considerations to create an interactive assessment system for graph theory within Moodle's STACK environment. The process followed a structured development

pipeline, closely tied to the capabilities and constraints of the Moodle platform, and can be summarized in five main phases:

1. **Requirements Analysis and Design:** The project began with identifying the limitations of current digital assessments for graph theory—namely, the lack of interactivity and randomization. From this, key objectives were defined: enabling interactive graph manipulation, supporting multiple graph algorithms, ensuring automated evaluation, and maintaining compatibility with STACK.
2. **Technology Integration:** The system was built on Moodle using the STACK plugin for mathematical evaluation, extended with JSXGraph to enable interactive graph-based input. JavaScript was used to track student interactions with graph elements (nodes and edges), while Maxima was used to generate randomized question parameters and evaluate submitted responses.
3. **Dynamic Question Generation:** Each exercise instance begins with randomized graph construction using Maxima. Teachers define parameters such as vertex count, edge density, and weight ranges. The system then filters a base graph, assigns weights if necessary, validates the graph structure, and exports the resulting adjacency matrix to JavaScript.
4. **Interactive User Interface:** The graph is rendered in the browser using JSXGraph. Students interact by selecting vertices and/or edges according to the task (e.g., DFS, BFS, Kruskal, Dijkstra). Their selections are automatically captured and written into STACK’s hidden input fields in a format suitable for Maxima evaluation.
5. **Custom Evaluation Logic:** Upon submission, the student’s answer is evaluated in STACK using algorithm-specific Maxima scripts. These scripts simulate the expected algorithm behavior based on the question’s randomized parameters and compare the computed result to the student’s input. The use of Potential Response Trees (PRTs) allows for immediate feedback on correctness.

This methodology ensures a balance between technical feasibility, educational value, and user engagement. It also provides a reproducible template for developing similar interactive assessments in other algorithmic domains.

1.6. Structure of the Thesis

This thesis is structured into five chapters, each contributing to a comprehensive understanding of the development and educational significance of interactive graph algorithm assessments within Moodle:

- **Chapter 1 – Introduction:** Presents the motivation, background, and objectives of the thesis. It outlines the current limitations of digital assessments for graph theory and introduces the project’s goals.
- **Chapter 2 – Theoretical Foundations:** Introduces the key technologies used—Moodle, STACK, Maxima, and JSXGraph—and explains fundamental graph theory concepts and algorithms relevant to the system.
- **Chapter 3 – Approach / Methodology Summary:** Summarizes the methodological framework adopted during development, including system design strategy, integration steps, and evaluation workflow.
- **Chapter 4 – System Design and Implementation:** Describes the technical implementation in detail, covering parameter generation, interactive graph rendering, student input handling, and algorithm-specific evaluation logic.
- **Chapter 5 – Results and Conclusion:** Reports on the system’s functionality, testing outcomes, and limitations. It concludes with a reflection on the project’s achievements and potential for further development.

This structure ensures a logical progression from conceptual foundations to technical execution and critical reflection.

2. Theoretical Foundations

This chapter outlines the core technologies and concepts underlying the interactive graph theory assessment system developed for Moodle. It introduces the digital tools used—Moodle, the STACK plugin, the Maxima computer algebra system, and the JSX-Graph visualization library—and explains their roles in enabling symbolic computation, interactivity, and automated evaluation.

The chapter also provides a brief overview of fundamental graph theory concepts, including vertices, edges, and weights, followed by the graph algorithms implemented in the project: Depth-First Search (DFS), Breadth-First Search (BFS), Kruskal’s algorithm, and Dijkstra’s algorithm. These algorithms are widely taught and well-suited for interactive educational formats.

2.1. Learning Platforms and Tools

The system developed in this thesis is built on Moodle, a widely used learning management system, extended by the STACK plugin for automated mathematical assessment. STACK uses the Maxima computer algebra system for symbolic evaluation and is enhanced with JSXGraph to support interactive graph visualizations. Together, these tools enable the creation and evaluation of algorithm-based graph theory exercises in a fully digital environment.

2.1.1. Moodle

Moodle is an open-source learning management system (LMS) widely used in education since its launch in 2002 [Moo24]. Its modular architecture supports plugin-based extensions, making it adaptable to various institutional needs.

In this project, Moodle provides the infrastructure for delivering interactive graph exercises through seamless integration with the STACK plugin and JSXGraph. Its open-source model ensures flexibility, sustainability, and compatibility with custom assessment solutions.

2.1.2. STACK

STACK (System for Teaching and Assessment using a Computer algebra Kernel) is a Moodle plugin designed for assessing mathematical input using symbolic computation [STA24a]. It integrates the Maxima computer algebra system to evaluate expressions symbolically, allowing for open-ended responses and detailed feedback through potential response trees (PRTs) [STA24c].

STACK supports randomized variables, custom evaluation logic, and interactive enhancements via JavaScript. In this thesis, it serves as the main framework for implementing algorithmic graph theory questions. To support visual input and interaction—especially for graph algorithms—STACK is extended using JSXGraph and custom JavaScript handling.

2.1.3. Maxima

Maxima is an open-source computer algebra system (CAS) used for symbolic computation in mathematical applications [Max24]. It supports tasks such as differentiation, integration, equation solving, and expression manipulation, making it well-suited for use in digital assessments.

In this project, Maxima serves as the backend engine of the STACK plugin, used to generate question variables, process symbolic input, and evaluate graph-theoretic conditions such as shortest paths or spanning trees. While Maxima operates in a sandboxed mode within Moodle—limiting some functionality—it remains essential for implementing custom logic and automated feedback.

2.1.4. JSXGraph

JSXGraph is a JavaScript library for interactive mathematical visualizations in the browser, supporting dynamic elements like points, lines, and graphs [STA24d]. In Moodle, it integrates with STACK via the `[[jsxgraph]]` tag, enabling interactive input directly within assessment questions.

In this project, JSXGraph is used to allow students to interact with graph structures—such as selecting vertices or edges—in algorithmic tasks. These inputs are captured via JavaScript and evaluated by STACK, bridging visual interaction with symbolic assessment. Its integration is sandboxed for security and enables rich, responsive question formats that enhance engagement with graph theory content.

2.2. Basics of Graph Theory

A graph is a mathematical structure used to represent relationships between objects. It consists of a set of vertices V and a set of edges E , where each edge connects a pair of vertices. In this thesis, we consider only *undirected graphs*, where edges have no direction — meaning that $\{u, v\}$ is equivalent to $\{v, u\}$.

Graphs can also be *weighted*, meaning that each edge is assigned a numerical value representing cost, distance, or another application-specific measure. These weights are particularly important for algorithms such as shortest path or minimum spanning tree computations [Spr07].

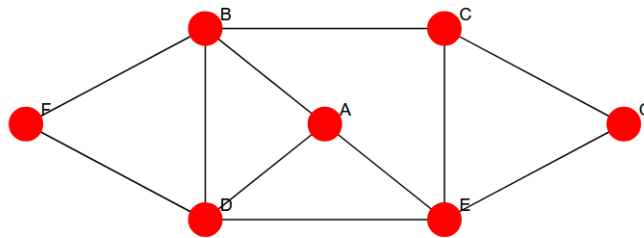


Figure 2.1.: Example of an undirected graph with 7 vertices and 12 edges.

Figure 2.1 illustrates a graph with vertex set $V = \{A, B, C, D, E, F, G\}$ and edge set:

$$E = \{\{A, B\}, \{A, D\}, \{A, E\}, \{B, C\}, \{B, D\}, \{B, F\}, \\ \{C, E\}, \{C, G\}, \{D, E\}, \{D, F\}, \{E, G\}, \{F, G\}\}.$$

2.3. Graph Algorithms Relevant to the Project

This section outlines key graph algorithms implemented in the project: DFS, BFS, Kruskal's, and Dijkstra's algorithms. These are foundational in graph theory education and well-suited for interactive assessment formats due to their stepwise, visualizable structure.

2.3.1. Depth-First Search (DFS)

Depth-First Search (DFS) is a fundamental algorithm for exploring all nodes reachable from a starting vertex in a graph. It follows a recursive strategy, visiting a node, marking it as explored, and continuing to an unexplored neighbor until reaching a dead end, at which point it backtracks [KT14].

This process produces a tree of discovery, known as the DFS tree, which captures the traversal order and structure.

```
Input: Graph  $G = (V, E)$ , starting node  $u \in V$   
Output: DFS tree rooted at  $u$   
Mark  $u$  as "Explored" and add  $u$  to the result set  $R$ ;  
foreach edge  $(u, v)$  incident to  $u$  do  
|   if  $v$  is not marked "Explored" then  
|   |   Recursively invoke DFS( $v$ );  
|   end  
end
```

Algorithm 1: Depth-First Search

2.3.2. Breadth-First Search (BFS)

Breadth-First Search (BFS) is used to explore a graph level by level from a starting node s , making it effective for finding shortest paths in unweighted graphs [KT14]. It uses a queue to ensure nodes are processed in the order they are discovered, maintaining a first-in, first-out (FIFO) sequence.

Each layer L_i contains nodes at distance i from s , and the traversal builds a BFS tree representing these distances.

Input: Graph $G = (V, E)$, start node $s \in V$
Output: BFS tree rooted at s and distance layers L
Set $\text{Discovered}[s] \leftarrow \text{true}$ and $\text{Discovered}[v] \leftarrow \text{false}$ for all $v \neq s$;
Initialize $L[0] \leftarrow \{s\}$, set $i \leftarrow 0$;
 $T \leftarrow \emptyset$;
while $L[i]$ is not empty **do**
 Initialize $L[i + 1] \leftarrow \emptyset$;
 foreach $u \in L[i]$ **do**
 foreach edge (u, v) incident to u **do**
 if $\text{Discovered}[v] = \text{false}$ **then**
 Set $\text{Discovered}[v] \leftarrow \text{true}$;
 Add edge (u, v) to T ;
 Add v to $L[i + 1]$;
 end
 end
 end
 $i \leftarrow i + 1$;
end
return T and L

Algorithm 2: Breadth-First Search

2.3.3. Kruskal's Algorithm

Kruskal's algorithm is a greedy method for finding a minimum spanning tree (MST) in a connected, weighted, undirected graph. It selects the smallest-weight edges that connect disjoint components, avoiding cycles by using a disjoint-set (Union-Find) structure [KT14].

The algorithm sorts all edges by weight and adds them one by one to the MST, merging components when the added edge connects previously separate sets. This continues until $|V| - 1$ edges have been added.

Kruskal's algorithm is efficient on sparse graphs and runs in $O(E \log E)$ time.[KT14]

Input: Graph $G = (V, E)$ with edge weights $w(e)$
Output: Minimum Spanning Tree $T \subseteq E$
Sort edges in E by increasing weight;
Initialize disjoint sets for each $v \in V$;
 $T \leftarrow \emptyset$;
foreach edge (u, v) in sorted E **do**
 if u and v are in different components **then**
 Add edge (u, v) to T ;
 Merge components of u and v ;
 end
end
return T

Algorithm 3: Kruskal's Algorithm

2.3.4. Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest path problem for graphs with non-negative edge weights. Given a graph $G = (V, E)$ and a start node s , it computes the minimum distance $d[v]$ from s to every other node v [KT14].

The version below is adapted to an adjacency matrix representation, matching the structure used in this project.

Input: Adjacency matrix M of size $n \times n$, start node s
Output: Shortest distances $d[i]$ from s to each node i

```
for  $i \leftarrow 1$  to  $n$  do  
  |  $d[i] \leftarrow \infty$   
end  
 $d[s] \leftarrow 0$ ;  
 $S \leftarrow \emptyset$ ;  
while  $|S| < n$  do  
  | Select node  $u \notin S$  with minimal  $d[u]$ ;  
  | Add  $u$  to  $S$ ;  
  | for  $v \leftarrow 1$  to  $n$  do  
  |   | if  $v \notin S$  and  $M[u][v] > 0$  and  $d[u] + M[u][v] < d[v]$  then  
  |   |   |  $d[v] \leftarrow d[u] + M[u][v]$ ;  
  |   |   end  
  |   end  
  end  
end  
return  $d$ 
```

Algorithm 4: Dijkstra's Algorithm using Adjacency Matrix

3. Requirements Analysis

In this chapter, the requirements and analytical foundations for the development of an interactive, algorithm-based graph theory assessment system within Moodle are outlined. The goal is to ensure a clear understanding of what the system must achieve, how users are expected to interact with it, and what limitations or challenges must be considered during design and implementation.

The requirements are derived from both pedagogical needs — such as improving the engagement and variability of graph theory exercises — and technical conditions — including Moodle’s plugin architecture and STACK’s evaluation model. The analysis also addresses different user roles (teachers and students) and the system features required to support those interactions effectively.

3.1. User Roles and Interaction Needs

The interactive system developed in this thesis involves two primary user roles: **teachers** and **students**. These roles differ significantly in how they interact with the system.

- **Teachers** are responsible for preparing algorithm-based graph theory questions within the Moodle STACK environment. Their interaction involves defining configuration parameters such as the number of vertices, edge density, and whether weights should be used. These parameters are used to generate randomized graph structures that serve as the basis for student tasks.
- **Students** interact with the generated graph exercises during quizzes or practice activities. Their task is to solve the given graph algorithm problem by interacting with the visualized graph using a mouse or touch interface, and then submit their solution for automated evaluation.

3.2. Functional Requirements

This section describes the core functionalities that the system must support in order to create and evaluate interactive graph theory questions. Each requirement is based on

the needs of either the teacher or the student, and directly supports the goal of providing a dynamic and algorithm-driven assessment environment.

- **Definition of Question Parameters:** Teachers must be able to define key parameters that control the structure and content of each question. These include the number of vertices, edge density, and the use of edge weights. Some parameters apply only to specific algorithms, such as weights for Kruskal's and Dijkstra's algorithms.
- **Random Generation of Graphs:** The system must be able to create randomized graph structures using the defined parameters. Each student should receive a unique graph instance, ensuring variation between attempts and supporting individual learning paths.
- **Interactive Graph Display:** The graph must be rendered interactively in the browser using JSXGraph. Students should be able to clearly view and interact with nodes and edges, with visual indicators such as color changes or labels showing selected elements.
- **Student Interaction with the Graph:** Students must be able to select nodes or edges according to the rules of the specific algorithm. The selection order must be recorded when necessary, such as in traversal algorithms like DFS and BFS.
- **Answer Capture and Submission:** The system must capture the student's selection as a structured response and submit it for evaluation. This includes converting the input into a string format that can be interpreted by the STACK question engine and processed by Maxima within the Potential Response Tree.
- **Automated Evaluation of Responses:** Submitted answers must be evaluated automatically using algorithm-specific control functions implemented in Maxima. This evaluation is handled through STACK's Potential Response Tree (PRT), which interprets the input and determines correctness based on the rules of the selected algorithm.

3.3. Non-Functional Requirements

This section outlines the non-functional requirements that apply to the system developed in this thesis. While functional requirements define what the system must do, non-functional requirements describe how well the system should perform within the given technical environment. Particular focus is placed on usability and the limitations of the Moodle-STACK infrastructure provided by Hochschule Hannover.

- **Usability:** Usability defines how easily teachers and students interact with the system. While Moodle is a highly customizable platform, such customizations generally require administrative permissions that are not available to teachers. Therefore, the teacher’s interaction with the system is limited to parameter configuration, and the design should ensure that this process is as simple and code-light as possible.
 - **Teacher Usability:** Teachers interact with the system by setting predefined parameters (e.g., number of vertices, edge density, and weights) when deploying a question in Moodle-STACK. The configuration process should require minimal code adjustments and no advanced technical skills.
 - **Student Usability:** The JSXGraph-based graph is displayed directly within the Moodle interface. Students interact with the graph by selecting nodes or edges according to the algorithm’s rules. Since the interface is kept simple and task-focused, no additional usability features are required.
- **Platform Constraints:** The system operates within the Moodle platform using the STACK plugin, subject to administrative policies at Hochschule Hannover. Certain Maxima functions are restricted, and no external JavaScript libraries or files may be imported. All interactive functionality must be embedded directly into the STACK question using inline JavaScript and HTML.
- **Robustness Within System Boundaries:** The system must remain robust despite platform limitations. Generated graphs must be algorithmically valid, and all user input must be handled without error. If a selection is invalid or incomplete, it should not cause the question to break. Instead, input should be ignored or appropriate fallback behavior should be applied.

3.4. Use Case Scenarios

To illustrate how the system is used in a real teaching context, the following scenarios describe typical interactions from both the teacher’s and the student’s perspective.

Use Case 1: Teacher Creates a Randomized Kruskal Question

Actor: Teacher

Goal: Create an interactive graph question for Kruskal’s algorithm with randomized weights.

- The teacher opens the STACK question editor in Moodle and selects a template for Kruskal's algorithm.
- The teacher defines parameters such as the number of vertices (e.g., min: 9, max: 13), edge density (e.g., 70%), and weight range (e.g., 1 to 9).
- STACK uses Maxima to generate a unique weighted graph instance for each student based on the specified parameters.
- The question is saved and deployed as part of a Moodle quiz.

Use Case 2: Student Solves a BFS Question

Actor: Student

Goal: Solve a randomly generated BFS task by selecting edges in the correct traversal order.

- The student opens the assigned Moodle quiz and sees a BFS graph rendered interactively using JSXGraph.
- The student clicks on the edges and vertices they believe form a valid BFS traversal starting from the root node.
- The system records the selection order and stores it as a response in the STACK input field.
- After submission, the Potential Response Tree evaluates the student's solution using Maxima logic.
- Based on the evaluation, the system provides immediate feedback indicating whether the answer is correct or incorrect.

4. System Design and Implementation

This chapter presents the structure and implementation of the interactive STACK-based questions developed for graph theory. It explains how key components of the system, such as graph generation, parameter randomization, interactive user input, and automated evaluation, are integrated to support exercises based on classical graph algorithms. These algorithms include traversal and optimization procedures that are fundamental in graph theory education.

4.1. Overview of the Interactive Exercise Generation Process

The development of interactive graph-based exercises in this system follows a consistent multi-phase workflow. Although the underlying logic and evaluation criteria differ for each algorithm, the technical pipeline for generating, displaying, and processing graph-based questions remains largely the same.

This workflow can be divided into five main phases:

1. **Parameter Definition:** The teacher specifies configuration parameters such as the number of nodes, edge density, weight range, and randomization settings within the STACK system.
2. **Graph Construction in Maxima:** Based on these parameters, Maxima code is executed server-side to generate a graph structure, including an adjacency matrix. Randomized filtering ensures variety across question instances while preserving pedagogical value.
3. **Interactive Graph Rendering:** The generated graph is passed to the browser and visualized using JSXGraph. Students can interact with the graph elements (nodes and edges) in a dynamic, task-specific way.
4. **Student Interaction:** User actions, such as selecting or deselecting vertices and edges, are tracked and stored in JavaScript variables. These variables are converted into a form readable by STACK and Maxima for automated evaluation.

5. **Answer Submission and Variable Binding:** Upon submission, the captured answer is passed to the Potential Response Tree (PRT) for algorithm-specific evaluation. Each algorithm uses a custom Maxima script to validate correctness by simulating the student's process.

This overview reflects the common infrastructure shared across all question types. The following sections detail each of these stages with respect to their implementation and how they support different graph algorithms.

4.2. Parameter Definition

Each exercise instance is generated based on a set of parameters defined in the **Question Variables** section of the STACK system. These include both fixed values and randomly generated variables, which together determine the structure and variation of the question.

Teachers can either define exact values or specify value ranges, allowing them to control the diversity and complexity of the generated questions. This supports the creation of individualized tasks while maintaining fairness and consistent assessment standards.

4.2.1. Initial Full Graph

To support algorithmic graph questions with meaningful training value, a predefined base graph called the *Initial Full Graph* is used. This graph includes a fixed number of vertices and is designed to avoid visual clutter such as edge crossings.

All subgraphs used in individual question instances are derived from this Initial Full Graph. This approach ensures that the visual layout remains clean and consistent across all variants, while still allowing for meaningful structural diversity.

Two foundational variables are defined in the **Question Variables** section to achieve this:

- **Vertex Coordinates:** A fixed list of coordinate pairs that determine the position of each vertex on the JSXGraph canvas. These are stored in the variable `vertex_positions`:

Listing 4.1: Predefined vertex coordinates

```
vertex_positions: [  
  [3,3],  
  [2,4], [4,4], [2,2], [4,2],  
  [0.5,3], [5.5,3], [3,5.5], [3,0.5],
```

```

    [1,5], [5,5], [5,1], [1,1]
];

```

- **Full Adjacency Matrix:** A symmetric 13×13 matrix, named `initial_graph`, representing all possible valid edges between the vertices without visual overlap:

Listing 4.2: Initial full adjacency matrix

```

initial_graph: matrix(
  [0,1,1,1,1,0,0,0,0,0,0,0,0],
  [1,0,1,1,0,1,0,1,0,1,0,0,0],
  [1,1,0,0,1,0,1,1,0,0,1,0,0],
  [1,1,0,0,1,1,0,0,1,0,0,0,1],
  [1,0,1,1,0,0,1,0,1,0,0,1,0],
  [0,1,0,1,0,0,0,0,0,1,0,0,1],
  [0,0,1,0,1,0,0,0,0,0,1,1,0],
  [0,1,1,0,0,0,0,0,0,1,1,0,0],
  [0,0,0,1,1,0,0,0,0,0,0,1,1],
  [0,1,0,0,0,1,0,1,0,0,0,0,0],
  [0,0,1,0,0,0,1,1,0,0,0,0,0],
  [0,0,0,0,1,0,1,0,1,0,0,0,0],
  [0,0,0,1,0,1,0,0,1,0,0,0,0]
);

```

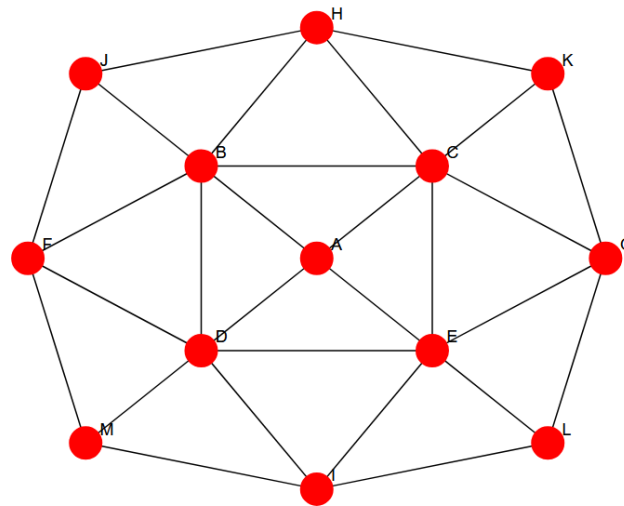


Figure 4.1.: The Initial Full Graph used for generating all question variants.

To keep the graph both visually structured and algorithmically meaningful, the number of vertices used in this thesis is limited to thirteen. The layout is arranged such that the

first vertex is placed at the center of the canvas, while the remaining twelve vertices are positioned around it in a structured and evenly spaced layout. If needed, the teacher can modify these default definitions and define a custom vertex layout or edge matrix to suit specific pedagogical goals.

4.2.2. Vertex Count Definition

To introduce variation between question instances, the number of active vertices used in the generated subgraph is selected randomly within a predefined range. This ensures that each generated graph is sufficiently complex while remaining easy to read and solve.

The teacher defines the minimum and maximum allowed number of vertices using the following variables:

```
min_vertex : 9;  
max_vertex : 13;
```

4.2.3. Edge Density Factor

To vary the structure of the generated subgraphs and ensure uniqueness across question instances, the teacher can define an edge density factor using the variable `edge_density`. This factor determines the percentage of edges from the full graph that will be retained in each question instance.

```
edge_density : 70;
```

The value is given as a percentage from 0 to 100. A value of 100 retains all edges of the selected subgraph, while lower values randomly remove a corresponding percentage of edges. For example, a value of 70 retains approximately 70% of the edges, creating sparser graphs. This mechanism allows the teacher to control the complexity of each exercise by adjusting how densely connected the graph is.

4.2.4. Edge Weights

For question types involving weighted graphs, such as Kruskal's and Dijkstra's algorithms, the teacher can define a minimum and maximum value for the edge weights. The system then randomly assigns a weight to each edge within this specified range.

```
min_weight : 1;  
max_weight : 9;
```

4.3. Graph Construction in Maxima

After the parameters are defined by the teacher, the process of generating a question variant begins. This is handled server-side, where the STACK system executes the Maxima code defined in the `Question Variables` section.

The goal of this step is to calculate an adjacency matrix that defines the structure of the graph used in the question instance. This matrix is derived from the predefined `initial_graph` and adapted according to the random values generated within the ranges provided by the teacher.

The generation process follows these steps:

4.3.1. Vertex Selection:

A random number is selected between `min_vertex` and `max_vertex`, which defines how many of the available vertices will be active in the current question instance. This value is stored in the variable `vertex_num`.

Listing 4.3: Random selection of the number of active vertices

```
vertex_num : min_vertex + rand(max_vertex - min_vertex + 1);
```

Now the original matrix `initial_graph` is updated by setting all rows and columns beyond the selected number of active vertices to zero. This operation preserves the overall matrix size but effectively disables the unused vertices.

Listing 4.4: Subsetting the initial graph matrix based on randomly selected vertex count

```
n : length(initial_graph);
initial_graph : genmatrix(
  lambda([i, j],
    if i <= vertex_num and j <= vertex_num then
      initial_graph[i, j]
    else 0
  ),
  n, n
);
```

For example, if the randomly selected number of vertices is 5, the resulting adjacency matrix becomes:

Listing 4.5: Resulting adjacency matrix for vertex_num = 5

```
initial_graph: matrix(  
  [0,1,1,1,1,0,0,0,0,0,0,0,0],  
  [1,0,1,1,0,0,0,0,0,0,0,0,0],  
  [1,1,0,0,1,0,0,0,0,0,0,0,0],  
  [1,1,0,0,1,0,0,0,0,0,0,0,0],  
  [1,0,1,1,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0],  
  [0,0,0,0,0,0,0,0,0,0,0,0,0]  
);
```

4.3.2. Edge Filtering

A random selection of edges is removed from the current adjacency matrix based on the percentage specified in `edge_density`. This reduction introduces structural variation in the graph, influencing its overall connectivity.

However, reducing edges may result in graphs that are disconnected or overly simple, such as trees, which may not provide sufficient learning value for algorithm exercises. Although graph algorithms can technically be applied to any graph structure, disconnected or trivial graphs are avoided in this system.

To address this, the edge reduction function is executed within a loop. If the generated graph does not meet a minimum structural threshold, the edge density is incrementally increased until the graph becomes suitable. The suitability of the graph is evaluated using the function `graph_is_valid`, which returns `true` only if the graph meets connectivity requirements.

Listing 4.6: Validation loop to ensure graph is connected and not a tree

```
graph_valid: false;  
  
while not graph_valid do (  
  filtered_edges: reduce_edges(initial_graph , edge_density),  
  edge_density: edge_density + 1,  
  graph_valid: graph_is_valid(filtered_edges , vertex_num)  
)
```

In the `reduce_edges` function, each existing edge (i.e., each non-zero value in the matrix) is checked against a randomly generated number. If the number is greater than the given `edge_density`, the edge is removed. Approximately `edge_density%` of the edges are retained.

Listing 4.7: Edge filtering function in Maxima

```

reduce_edges(M, p) := block(
  [n, i, j, filtered_M],
  n : length(M),
  filtered_M : copy(M),
  for i:1 thru n do (
    for j:i+1 thru n do (
      if M[i][j] = 1 and rand(100) > p then (
        filtered_M[i][j] : 0,
        filtered_M[j][i] : 0
      )
    )
  ),
  return(filtered_M)
)$

```

An example result of this edge filtering process for a selected `vertex_num = 7` is shown in Figure 4.2.

Listing 4.8: Filtered adjacency matrix with `edge_density` applied

```

matrix(
  [0,1,1,0,1,0,0,0,0,0,0,0,0],
  [1,0,1,0,0,1,0,0,0,0,0,0,0],
  [1,1,0,0,1,0,0,0,0,0,0,0,0],
  [0,0,0,0,1,1,0,0,0,0,0,0,0],
  [1,0,1,1,0,0,1,0,0,0,0,0,0],
  [0,1,0,1,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,1,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0]
);

```

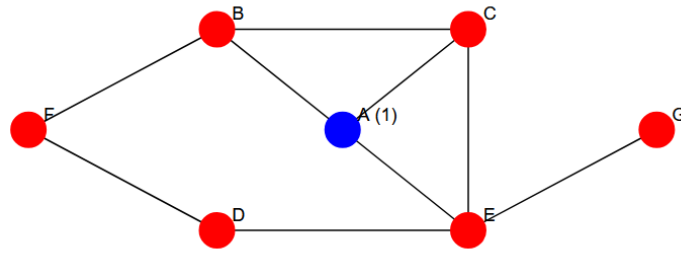


Figure 4.2.: Visual representation of the filtered graph after edge reduction.

To ensure the graph remains connected and non-trivial, the following validation is applied using a Depth-First Search (DFS) traversal:

Listing 4.9: Validation function to check for connected, non-tree graphs

```
graph_is_valid(M, vertex_num) := block(
  [n, visited, dfs_tree, result],
  result : true,
  n : length(M),
  visited : makelist(false, i, 1, n),
  dfs_tree : zeromatrix(n, n),
  dfs(M, n, 1, visited, dfs_tree),

  if has_isolated_node(dfs_tree, vertex_num) or equal(M,
    dfs_tree) then
    result : false
  else
    result : true,
  return(result)
)$
```

Listing 4.10: Check for isolated nodes in the DFS tree

```
has_isolated_node(M, vertex_num) := block(
  [i, j, rowsum, returnVal],
  returnVal : false,
  for i : 1 thru vertex_num do (
    rowsum : 0,
    for j : 1 thru vertex_num do (
      rowsum : rowsum + M[i][j] + M[j][i]
    ),
    if rowsum = 0 then (returnVal : true)
  ),
  ),
```

```

return(returnVal)
)$

```

Listing 4.11: Depth-First Search traversal to build a DFS tree

```

dfs(M, n, node, visited, R) := block(
  [i],
  if not visited[node] then (
    visited[node] : true,
    for i : 1 thru n do (
      if (M[node][i] = 1 or M[i][node] = 1) and not visited[i]
        then (
          R[node][i] : 1,
          R[i][node] : 1,
          dfs(M, n, i, visited, R)
        )
    )
  )
)$

```

4.3.3. Weight Assignment

For weighted graph algorithms, the filtered adjacency matrix is updated with random weight values. The weights are generated as integers between `min_weight` and `max_weight` and are assigned to all remaining edges. This transforms the matrix into a weighted graph, where each non-zero entry represents the weight of the corresponding edge.

Listing 4.12: Assigning random weights to edges in the adjacency matrix

```

assign_random_weights(M, min_weight, max_weight) := block(
  [n, i, j, W],
  n : length(M),
  W : copy(M),
  for i : 1 thru n do (
    for j : i+1 thru n do (
      if M[i][j] # 0 then (
        W[i][j] : min_weight + rand(max_weight - min_weight + 1)
        ,
        W[j][i] : W[i][j]
      )
    )
  )
  ,
  return(W)
)

```

)\$

An example of a resulting weighted adjacency matrix is shown below. Each non-zero value indicates the assigned weight for a valid edge:

Listing 4.13: Example of a weighted adjacency matrix

```
matrix(
  [0,8,9,8,0,0,0,0,0,0,0,0,0,0],
  [8,0,0,0,0,0,0,0,0,0,0,0,0,0],
  [9,0,0,0,0,0,8,6,0,0,0,0,0,0],
  [8,0,0,0,2,4,0,0,1,0,0,0,0,0],
  [0,0,0,2,0,0,7,0,1,0,0,0,0,0],
  [0,0,0,4,0,0,0,0,0,0,0,0,0,0],
  [0,0,8,0,7,0,0,0,0,0,0,0,0,0],
  [0,0,6,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,1,1,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
  [0,0,0,0,0,0,0,0,0,0,0,0,0,0]
);
```

The corresponding visualization of this weighted graph is shown in Figure 4.3.

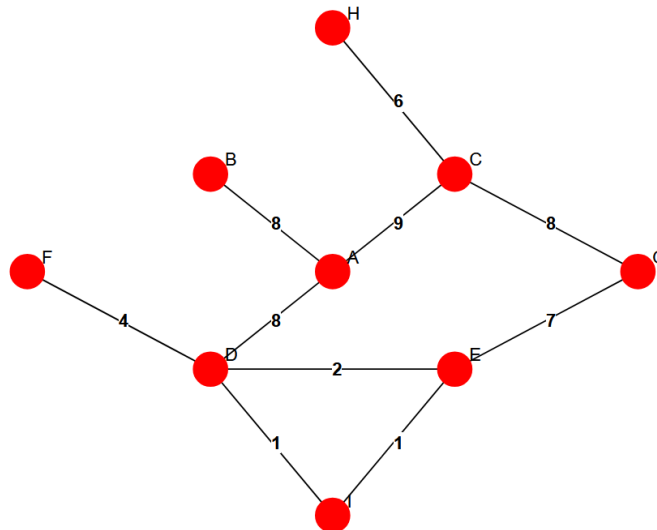


Figure 4.3.: Graph rendering with weighted edges after applying random weight assignment.

4.3.4. Matrix Transfer to JavaScript:

Since Maxima matrices cannot be passed directly to JavaScript, the final adjacency matrix is converted into a list of lists to make it readable in the browser environment. This format can be interpreted as a standard JavaScript array structure and used to render the graph using JSXGraph.

Listing 4.14: Conversion of Maxima matrix to list-of-lists format

```
edges_list : args(matrixmap(lambda([r], r), filtered_edges));
```

4.4. Interactive Graph Rendering

Once the graph structure has been generated server-side using Maxima, it is transferred to the client for interactive rendering. This is accomplished using JSXGraph, a JavaScript library designed for visualizing and interacting with mathematical structures. JSXGraph is embedded directly into the STACK question through a custom HTML and JavaScript block within the question text.

The rendering process proceeds as follows:

4.4.1. Initialization of the JSXGraph Board

The rendering process begins with the initialization of a JSXGraph board using the `initBoard` function. This board serves as the canvas on which all visual elements of the graph are displayed. To ensure a clean and distraction-free interface, auxiliary elements such as coordinate axes, grid lines, and navigation controls are explicitly disabled.

```
var board = JXG.JSXGraph.initBoard(divid, {
  boundingbox: [0, 6, 6, 0],
  axis: false,
  grid: false,
  showNavigation: false,
  showCopyright: false
});
```

Listing 4.15: JSXGraph board initialization

4.4.2. Importing Parameters from Maxima

The essential parameters required for graph rendering are passed from Maxima to the JavaScript environment using STACK's variable substitution system. These parameters include:

- `vertex_positions`: a list of coordinate pairs used to place each vertex on the board,
- `vertex_num`: the number of active vertices in the current question instance,
- `edges_list`: the adjacency matrix, formatted as a list of lists for compatibility with JavaScript,
- `first_node`: a randomly selected starting vertex used in traversal-based algorithms.

These parameters provide the necessary input for constructing the graph structure and enabling user interaction.

```
var vertex_positions = {#vertex_positions#};
var vertexNum = {#vertex_num#};
var edges = {#edges_list#};
var first_node = {#first_node#};
```

Listing 4.16: Importing data from Maxima

4.4.3. Drawing Vertices

Each vertex is rendered as a fixed point on the JSXGraph board using the coordinates specified in the `vertex_positions` array. A loop iterates over the number of active vertices defined by `vertex_num` and creates a JSXGraph `point` object at each corresponding location.

Each point is assigned an index representing its position in the graph and is stored in an array called `pointList` for later use in rendering and interaction logic. In addition, a new property called `baseName` is defined for each point. This property stores the originally assigned name of the vertex and enables dynamic renaming of vertices during selection and deselection processes.

```
for (var i = 0; i < vertexNum; i++) {
  var point = board.create('point', vertex_positions[i], {
    color: '#FF0000',
    size: 10,
    highlight: false,
```

```

    showInfobox: false,
    fixed: true
  });
  point.baseName = point.name;
  point.index = i;
  pointList.push(point);
}

```

Listing 4.17: Creating vertex points

4.4.4. Drawing Edges

Edges between vertices are rendered as undirected line segments using JSXGraph's `segment` objects. The adjacency structure of the graph serves as a guide for rendering these edges. For each non-zero entry in the adjacency matrix, the corresponding vertices are retrieved from the `pointList` array, and a new segment is created between them.

Since all graphs in this thesis are undirected, only the upper triangle of the adjacency matrix is examined to prevent duplicate edge rendering. For graph algorithms that involve weighted edges, such as Dijkstra and Kruskal, each edge displays its corresponding weight, which is taken from the adjacency matrix as the edge's value. This is implemented by enabling the segment label and assigning the weight as its name.

To improve readability, especially in cases where labels might overlap with other elements on the canvas, a white background is applied to each edge label.

Additionally, a custom property named `weight` is assigned to each edge object to store the numeric weight. This allows the original weight to be easily accessed later during selection or deselection operations, without referring back to the adjacency matrix.

An example of a rendered weighted graph is shown in Figure 4.3 in Section 4.3.3.

```

for (var i = 0; i < vertexNum; i++) {
  for (var j = i + 1; j < vertexNum; j++) {
    if (edges[i][j] != 0) {
      var edge = board.create('segment', [pointList[i],
        pointList[j]], {
        strokeColor: 'black',
        strokeWidth: 1,
        highlight: false,
        withLabel: true,
        label: {
          position: 'middle',

```

```
        offset: [0, 0],
        cssStyle: 'background-color: white; font-size: 14px;
                  font-weight: bold;'
    }
  });

  edge.weight = edges[i][j];
  changeName(edge, edge.weight);
}
}
}
```

Listing 4.18: Rendering weighted undirected edges

4.4.5. Start Node Initialization

To execute algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra’s algorithm, a starting vertex is required. This vertex is randomly selected from the set of displayed vertices and stored as `first_node` in the question text.

As will be described in more detail in Section ??, the function `vertexSelect(point)` is designed to handle vertex selection for the user’s answer. For algorithms requiring a starting point, `first_node` is automatically selected as the first active vertex in the graph.

```
var first_nodeIndx = first_node - 1;
vertexSelect(pointList[first_nodeIndx]);
pointList[first_nodeIndx].off('down');
```

Listing 4.19: Preselecting the starting vertex

Since deselecting vertices is normally permitted, this action is explicitly disabled for the starting vertex at the beginning of the question to ensure its role is preserved.

4.5. Student Interaction

This section describes how students interact with the graph to execute the requested algorithm. Since the process of creating and rendering the graph is consistent across all question types, the interaction logic is also largely unified.

Students can select and deselect vertices and edges to construct their responses. This interactivity allows them to correct mistakes by modifying their selections. A key limitation is that, for algorithms where the order of selection matters, only the most recently selected element can be deselected. In addition, if a starting vertex is defined by the algorithm, it is locked during the initialization phase and cannot be deselected.

Student responses are captured in two distinct variables:

1. An adjacency matrix that stores the set of selected edges.
2. An array that records the ordered sequence of selected vertices (if vertex selection is required).

For algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's algorithm, students are required to select both the relevant vertices and the associated edges. The sequence of selected vertices is used for evaluation, as it reflects the algorithm's traversal order. While the selected edges must form a valid solution structure, the order in which they are chosen is not assessed. This flexibility allows students to adjust edge selections as needed to construct an appropriate spanning structure.

In contrast, for Kruskal's algorithm, only edge selection is permitted, and the order in which edges are selected is essential for evaluation. Vertex selection is not allowed in Kruskal-type questions.

4.5.1. Vertex Interaction

The method of user interaction with vertices is limited to clicking with a mouse. As described in Section ??, each vertex is rendered as a JSXGraph point. During graph generation, an event handler is assigned to every created point object:

```
point.on('down', function(evt) {
  if (selectedVertexList[selectedVertexList.length - 1] === this
    .index + 1) {
    vertexDeselect(this);
  } else if (!selectedVertexList.includes(this.index + 1)) {
    vertexSelect(this);
  }
});
```

Listing 4.20: Vertex click event listener

This event checks whether the clicked vertex is already selected. If it is the most recently selected vertex, it can be deselected. Otherwise, if it is not yet selected, it is added to

the selection. The selection state is tracked using the `selectedVertexList` array, which stores the indices of selected vertices in order.

The functions `vertexSelect` and `vertexDeselect` update both the visual appearance of the vertex and the internal data structure that records the student's answer. After each change in a vertex's state, the function `updateAns` is called to synchronize the displayed graph with the answer fields. This function will be explained in detail in Subsection ??

```
function vertexSelect(point) {
  counter++;
  point.setAttribute({color: 'blue'});

  var newName = point.baseName + " (" + counter + ")";
  changeName(point, newName);

  selectedVertexList.push(point.index + 1);
  updateAns();
}

function vertexDeselect(point) {
  counter--;
  selectedVertexList.pop();

  var newName = point.baseName;
  changeName(point, newName);
  point.setAttribute({color: 'red'});

  updateAns();
}
```

Listing 4.21: Vertex selection and deselection handlers

Selected and deselected vertices are visually distinct. Selected vertices are shown in blue and are annotated with their selection order, while deselected vertices revert to their original red color and name. Figure 4.4 illustrates the visual difference before and after a vertex is selected in a Dijkstra graph.

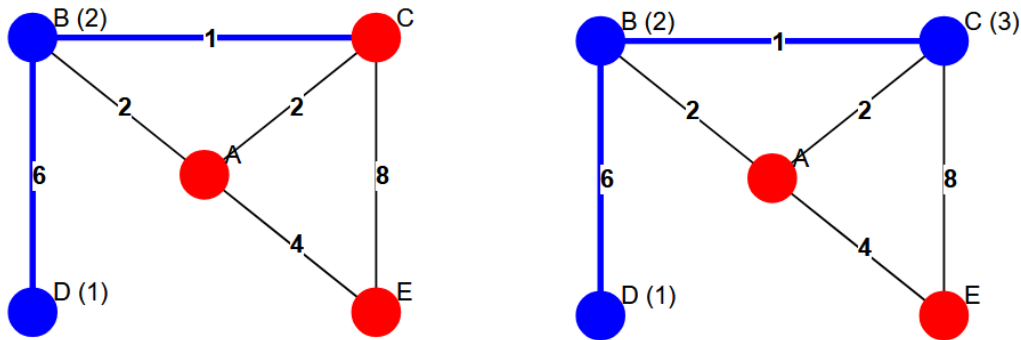


Figure 4.4.: Visual appearance of a Dijkstra graph before and after vertex selection.

4.5.2. Edge Interaction

Similar to vertices, students can interact with edges by clicking them with the mouse to toggle their selection status. During graph generation, each created edge is assigned an event handler that enables this interactivity.

When an edge is clicked, the system checks whether it is currently selected. If not, it is added to the student's response; if it is already selected, it can be deselected, depending on the algorithm's constraints.

In algorithms such as Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's algorithm, the order in which edges are selected is not relevant for evaluation. Therefore, a matrix named `selectedEdgeMatrix` is used to track the state of each edge. If the corresponding entry in the matrix is zero, the edge is considered unselected; otherwise, it is marked as selected.

```
edge.on('down', function(evt) {
  var index1 = this.point1.index;
  var index2 = this.point2.index;

  if ((selectedEdgeMatrix[index1][index2] === 0) ||
      (selectedEdgeMatrix[index2][index1] === 0)) {
    edgeSelect(this);
  }
});
```

```
    } else {  
      edgeDeselect(this);  
    }  
  });
```

Listing 4.22: Edge interaction for unordered selection algorithms

The functions `edgeSelect` and `edgeDeselect` update the visual appearance of the edge and invoke `edgeMatrixUpdate` to update the corresponding entry in `selectedEdgeMatrix`. The updated value reflects whether the edge is selected (1) or not (0).

```
function edgeSelect(edge) {  
  edgeMatrixUpdate(edge, 1);  
  edge.StrokeBase = ['blue', 3];  
  changeEdgeStroke(edge, edge.StrokeBase);  
}  
  
function edgeDeselect(edge) {  
  edgeMatrixUpdate(edge, 0);  
  edge.StrokeBase = ['black', 1];  
  changeEdgeStroke(edge, edge.StrokeBase);  
}  
  
function edgeMatrixUpdate(edge, val) {  
  let p1idx = edge.point1.index;  
  let p2idx = edge.point2.index;  
  selectedEdgeMatrix[p1idx][p2idx] = val;  
  selectedEdgeMatrix[p2idx][p1idx] = val;  
  updateAns();  
}
```

Listing 4.23: Edge selection and matrix update functions

For Kruskal's algorithm, where the order of edge selection is essential, a different approach is employed. A list named `selectedEdgeList` stores the sequence in which edges are selected. Only the most recently selected edge can be deselected, mirroring the logic used for vertex interaction in traversal-based algorithms.

```
edge.on('down', function(evt) {  
  if (selectedEdgeList[selectedEdgeList.length - 1] === this) {  
    edgeDeselect();  
  } else {  
    edgeSelect(this);  
  }  
});
```

Listing 4.24: Edge interaction for Kruskal's algorithm

In this case, the `edgeSelect` and `edgeDeselect` functions also update the edge label to reflect its selection order. In addition, unlike other algorithms, the `selectedEdgeMatrix` stores the order number of each selected edge instead of a binary value.

```
function edgeSelect(edge) {
  if (selectedEdgeList.includes(edge)) { return; }
  counter++;
  changeName(edge, edge.weight + " (" + counter + ")");
  edge.StrokeBase = ['blue', 3];
  changeEdgeStroke(edge, edge.StrokeBase);
  selectedEdgeList.push(edge);
  edgeMatrixUpdate(edge, counter);
}

function edgeDeselect() {
  edge = selectedEdgeList.pop();
  counter--;
  changeName(edge, edge.weight);
  edge.StrokeBase = ['red', 1];
  changeEdgeStroke(edge, edge.StrokeBase);
  edgeMatrixUpdate(edge, 0);
}
```

Listing 4.25: Edge selection logic for Kruskal's algorithm

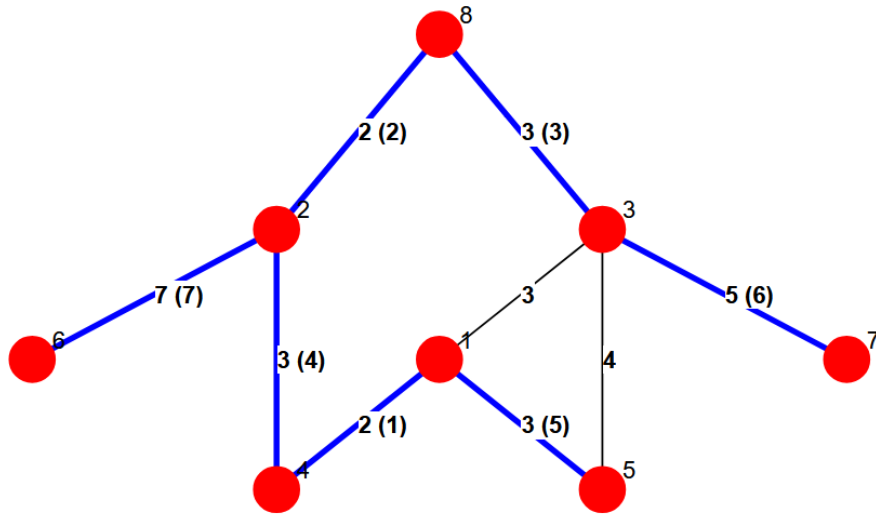


Figure 4.5.: Visualization of edge selection sequence in Kruskal's algorithm. The labels on selected edges show both the edge weight and the selection order.

4.6. Answer Submission and Variable Binding

In STACK, submitting an answer is handled by a built-in mechanism that captures the student's input and passes it to the Potential Response Tree (PRT) for evaluation. To enable this process, at least one input variable must be defined in the question. In a typical STACK question, this is done by adding the following line to the question text:

```
<p>[[input:ans1]] [[validation:ans1]]</p>
```

Listing 4.26: Standard STACK input field

This creates a visible input field where students can manually enter their answers. However, in this thesis, students submit responses exclusively through interaction with the graph. Therefore, the input fields are hidden from view but are still required for capturing and submitting responses.

```
<p hidden>[[input:ans1]] [[validation:ans1]]</p>
<p hidden>[[input:ans2]] [[validation:ans2]]</p>
```

Listing 4.27: Hidden input fields for automatic submission

Depending on the algorithm, a different number of input variables are used:

- For Kruskal’s algorithm, only one input variable (`ans1`) is required.
- For DFS, BFS, and Dijkstra’s algorithm, two input variables (`ans1` and `ans2`) are defined: one to capture the selected edge structure, and one for the sequence of selected vertices.

STACK supports integration with JSXGraph by allowing its input fields to be referenced and updated via JavaScript using `input-ref` attributes. These references are accessed and updated during interaction:

```
[[jsxgraph input-ref-ans1="ans1Ref" input-ref-ans2="ans2Ref"]]

var input1 = document.getElementById(ans1Ref);
var input2 = document.getElementById(ans2Ref);
```

Listing 4.28: Binding and accessing STACK input variables via JSXGraph

To submit a valid response to Maxima for evaluation, the input type must be set to "Algebraic." This ensures that the submitted value can be parsed correctly by Maxima. The function `updateAns()` is responsible for converting the JavaScript data structures into formats readable by Maxima and writing them into the STACK input fields. It should be called whenever the student updates their selection.

One important constraint is that the length of the submitted arrays must remain consistent. Any mismatch in array size will result in a STACK evaluation error. The function handles padding or resetting the sequence accordingly.

```
function updateAns() {
  var matrixExpr = "matrix(";
  for (var i = 0; i < 13; i++) {
    matrixExpr += "[" + ergMatrix[i].join(",") + "];";
    if (i < 12) matrixExpr += ",";
  }
  matrixExpr += ")";

  if (input1) {
    input1.value = matrixExpr;
    input1.dispatchEvent(new Event('change'));
  } else {
    console.log("input1 not found");
  }

  if (input2) {
    let sequenceWithPadding = indexList.slice();
    while (sequenceWithPadding.length < vertexNum) {
      sequenceWithPadding.push(0);
    }
  }
}
```

```
    }
    if (sequenceWithPadding.length > vertexNum) {
        sequenceWithPadding = Array(vertexNum).fill(-1);
    }
    input2.value = "[" + sequenceWithPadding.join(",") + "]";
    input2.dispatchEvent(new Event('change'));
} else {
    console.log("input2 not found");
}
}
```

Listing 4.29: JavaScript function to update STACK input variables

4.7. Evaluation

For all algorithms considered in this thesis, multiple valid solutions may exist for a given question. Therefore, it is not sufficient to precompute a single correct answer and compare it directly with the student's response. Instead, each algorithm includes a custom evaluation script implemented in Maxima, which programmatically checks the validity of the student's answer based on the logic of the specific algorithm.

These evaluation scripts simulate the algorithm step by step, using the student's submitted sequence of vertices and edges. At each step, the script verifies whether the decision made by the student is consistent with the algorithm's expected behavior. As the simulation proceeds, a result graph is constructed to represent the answer as interpreted by the system.

After the traversal or construction is complete, this generated result is compared with the student's submitted graph. If the structures match and all steps are valid, the Maxima script sets a Boolean variable `result` to `true`; otherwise, it is set to `false`. In all algorithms, this variable is used by the Potential Response Tree (PRT) in STACK to determine whether the submitted answer is correct.

4.7.1. Evaluation Strategy for Breadth-First Search (BFS)

The evaluation of a student's response to a BFS-based question is performed using a custom Maxima script that simulates the traversal according to the student's submitted vertex order. The goal is to reconstruct a BFS tree and compare it to the student's selected edges.

The evaluation process includes the following steps:

- The student's answer is split into two components:
 - `ansMatrix` — an adjacency matrix representing the selected edges.
 - `ansNodeOrder` — an ordered list of selected vertices.
- A zero-initialized matrix named `result_matrix` is created to store the reconstructed BFS tree.
- A standard BFS queue is used to simulate the traversal. At each level, the algorithm identifies the unvisited neighbors of the current node and collects them in a `sub_queue`.
- For every group of neighbors discovered during traversal, the script checks whether the student submitted the same nodes in any order. This is validated by comparing the corresponding number of entries from `ansNodeOrder` with the nodes found in `sub_queue`. Any mismatch results in an incorrect outcome.
- Each valid neighbor is marked as visited and added to the result matrix. The ordered group of accepted neighbors is appended to the BFS queue.
- After traversal is complete, the constructed `result_matrix` is compared to the student's submitted `ansMatrix`. If they differ, the answer is marked as incorrect.

This evaluation ensures that the student's traversal order

Listing 4.30: Maxima code for evaluating a BFS solution

```

ansMatrix: ans1;
ansNodeOrder : copylist(ans2);

result : true;
result_matrix : genmatrix(lambda([i,j], 0), 13, 13);

n: vertex_num;
M: filtered_edges;
start_node: first_node;
visited : makelist(false, i, 1, n);
queue : [start_node];
visited[start_node] : true;
bfs_order : [];

if first(ansNodeOrder) # start_node then (
  result : false
);
ansNodeOrder : rest(ansNodeOrder);

while queue # [] and result do (

```

```
node : first(queue),
queue : rest(queue),
bfs_order : endcons(node, bfs_order),

sub_queue : [],
for i : 1 thru n do (
  if M[node, i] = 1 and not visited[i] then (
    visited[i] : true,
    result_matrix[node, i] : 1,
    result_matrix[i, node] : 1,
    sub_queue : endcons(i, sub_queue)
  )
),

ans_sub_queue : [],
for j : 1 thru length(sub_queue) do (
  if length(ansNodeOrder) = 0 then (
    result : false
  )
  else (
    ans_node : first(ansNodeOrder),
    ansNodeOrder : rest(ansNodeOrder),
    if not member(ans_node, sub_queue) then (
      result : false
    ),
    ans_sub_queue : endcons(ans_node, ans_sub_queue)
  )
),

queue : append(queue, ans_sub_queue)
);

if (not equal(result_matrix, ansMatrix)) then result : false;
```

4.7.2. Evaluation Strategy for Depth-First Search (DFS)

To evaluate a student's response to a DFS-based question, a custom Maxima script is implemented. This script simulates a DFS traversal by following the vertex order provided by the student and reconstructs the corresponding traversal tree.

The evaluation process consists of the following steps:

- The student's answer is divided into two components:

- `ansMatrix` — an adjacency matrix representing the selected edges.
- `ansNodeOrder` — an ordered list of the selected vertices.
- A zero-initialized matrix, `result_matrix`, is created to store the tree structure generated during the evaluation.
- The simulation begins at the student’s specified start node. At each step, the script checks whether the next vertex in the list is a valid unvisited neighbor of the current parent node in the original graph. This determines whether the student was allowed to traverse to that node according to DFS rules.
- If the move is valid, the node is marked as visited and the edge is added to `result_matrix`. If the move is invalid, the script first checks whether the student skipped any valid unvisited neighbors. If such a neighbor exists, the answer is marked incorrect. Otherwise, the algorithm backtracks to the previous parent node. If no valid parent remains, the evaluation terminates with an incorrect result.
- Once the traversal is complete, the generated `result_matrix` is compared to the student-submitted `ansMatrix`. If the two matrices do not match, the answer is marked as incorrect.

This evaluation approach ensures that the vertex sequence submitted by the student follows a valid DFS traversal and that the selected edges form a structurally correct traversal tree.

Listing 4.31: Maxima code for evaluating a DFS solution

```
ansMatrix: ans1;
ansNodeOrder : copylist(ans2);

result : true;
result_matrix : genmatrix(lambda([i,j], 0), 13, 13);

n: vertex_num;
M: filtered_edges;
start_node: first_node;
visited : makelist(false, i, 1, n);

nodePointer: 2;
parentPointer: 1;
visited[ansNodeOrder[parentPointer]]: true;

while (nodePointer <= n) do (
  node : ansNodeOrder[nodePointer],
  parent : ansNodeOrder[parentPointer],
```

```
if node > 0 then (  
  if ((equal(M[node][parent], 1) or equal(M[parent][node],  
    1)) and not visited[node]) then (  
    result_matrix[node][parent] : 1,  
    result_matrix[parent][node] : 1,  
    visited[node] : true,  
    parentPointer : nodePointer,  
    nodePointer : nodePointer + 1  
  ) else block(  
    for i : 1 thru n do (  
      if ((equal(M[parent][i], 1) or equal(M[i][parent], 1))  
        and not visited[i]) then (  
        result : false,  
        nodePointer : n + 1  
      )  
    ),  
    parentPointer : parentPointer - 1,  
    if parentPointer <= 0 then (  
      result : false,  
      nodePointer : n + 1  
    )  
  )  
  ) else (  
    result : false,  
    nodePointer : n + 1  
  )  
);  
  
if (not equal(result_matrix, ansMatrix)) then result : false;
```

4.7.3. Evaluation Strategy for Kruskal's Algorithm

To evaluate the student's response to a Kruskal-based question, a custom Maxima implementation of Kruskal's Minimum Spanning Tree (MST) algorithm is used. The evaluation logic compares the sequence and validity of the student's selected edges against the expected behavior of Kruskal's algorithm.

The process follows these key steps:

- **Sorting edges:** Both the student's answer (`ans1`) and the correct filtered graph (`filtered_edges`) are converted into sorted edge lists, where edges are ordered by increasing weight. This mimics Kruskal's requirement of selecting the smallest edge that doesn't form a cycle.

- **Disjoint-set setup:** A `parent` array is initialized to implement a union-find structure. This structure is used to track connected components and avoid cycles.
- **Validation loop:** The script iteratively compares each student-selected edge with the correct minimal edge:
 - If the student skipped a lighter valid edge, or selected an edge forming a cycle, the result is marked as incorrect.
 - If the student's edge is valid and connects two disjoint sets, it is accepted and added to the MST by updating the `parent` array.
 - The selection order is preserved and must match Kruskal's greedy selection behavior.
- **Final check:** After processing, if fewer than `vertex_num - 1` edges were selected, the answer is incomplete and marked incorrect.

The evaluation ensures the student correctly followed Kruskal's algorithm, respecting both weight order and acyclic connectivity constraints.

Listing 4.32: Kruskal evaluation in Maxima

```

/* Find and union helpers omitted for brevity */

result : true;
sortedResult : matrix_to_sorted_edges(ans1);
sortedFilterdMatrix : matrix_to_sorted_edges(filtered_edges);
parent : makelist(i, i, 1, vertex_num);
selectedEdges : 0;

while length(sortedResult) > 0 do (
  sr1 : first(sortedResult),
  sr1U : sr1[2][1],
  sr1V : sr1[2][2],
  sr1W:filtered_edges[sr1U][sr1V],
  sfm1 : first(sortedFilterdMatrix),

  if sr1W > sfm1[1] then (
    /* Skipped lighter edge */
    if find(sfm1[2][1], parent) # find(sfm1[2][2], parent) then
      (
        result : false,
        return()
      ),
    sortedFilterdMatrix : rest(sortedFilterdMatrix)
  ) else if sr1W < sfm1[1] then (

```

```
/* Selected too small edge */
result : false,
return()
) else (
if find(sr1U, parent) = find(sr1V, parent) then (
/* Selected edge forms cycle */
result : false,
return()
),
unionEdge(sr1U, sr1V, parent),
selectedEdges : selectedEdges + 1,
sortedResult : rest(sortedResult),
sortedFilterdMatrix : delete(sr1, sortedFilterdMatrix)
)
);

if selectedEdges < vertex_num - 1 then result : false;
```

4.7.4. Evaluation Strategy for Dijkstra's Algorithm

To assess a student's response to a Dijkstra-based question, a custom Maxima script is used that simulates Dijkstra's algorithm on the same filtered graph shown to the student. The evaluation logic verifies both the correctness of the vertex order (based on shortest-path priorities) and the resulting structure of the path tree.

The evaluation follows these steps:

- The student's answer is split into:
 - `ansMatrix` — an adjacency matrix representing the selected edges.
 - `ansDistOrder` — the order in which nodes are selected during traversal.
- A distance list `d` is initialized with `inf` for all nodes, except the start node which is set to zero. A visited list keeps track of processed nodes.
- For each node in the submitted order, the algorithm checks whether the node has the current minimum tentative distance among all unvisited nodes. If a node is selected too early (i.e., another unvisited node has a smaller distance), the answer is marked incorrect.
- Upon valid selection, the node is marked as visited, and the relaxation step is performed: for each unvisited neighbor, the algorithm checks whether a shorter path through the current node exists. If so, the distance is updated, and the

corresponding edge in `result_matrix` is updated to reflect the most recent parent connection.

- For each such update, any previously recorded edge to the updated node is removed to ensure only the latest shortest connection is stored in the result matrix.
- After processing all nodes, the final `result_matrix` is compared to the student's submitted `ansMatrix`. A mismatch results in an incorrect evaluation.

This evaluation ensures the student has correctly followed Dijkstra's algorithm, both in terms of shortest-path logic and in maintaining the structure of the path tree.

Listing 4.33: Dijkstra evaluation in Maxima

```

ansMatrix: ans1;
ansDistOrder: ans2;

result : true;
result_matrix : genmatrix(lambda([i,j], 0), 13, 13);

n: vertex_num;
M: filtered_edges;
start_node: first_node;

d: makelist(inf, i, 1, n);
visited: makelist(false, i, 1, n);
d[start_node]: 0;

for ap: 1 thru n do(
  ansNode: ansDistOrder[ap],
  ansNodeDist: d[ansNode],

  for u: 1 thru n do (
    if not visited[u] and d[u] < ansNodeDist then (
      result: false,
      return()
    )
  ),

  visited[ansNode]: true,

  for v: 1 thru n do (
    if not visited[v] and M[ansNode][v] # 0 then (
      if d[ansNode] + M[ansNode][v] < d[v] then (
        d[v]: d[ansNode] + M[ansNode][v],

```

```
        for i: 1 thru n do (
            result_matrix[i, v]: 0,
            result_matrix[v, i]: 0
        ),

        result_matrix[ansNode, v]: 1,
        result_matrix[v, ansNode]: 1
    )
)
);

if (not equal(result_matrix, ansMatrix)) then result : false;
```

5. Results and Conclusion

This chapter summarizes the outcomes of the developed system, discusses its effectiveness in addressing the limitations of graph-based digital assessments in Moodle, and outlines opportunities for future improvement. The main objective was to design and implement an interactive and randomized environment for evaluating classical graph algorithms using STACK and JSXGraph.

5.1. Summary of Achievements

The system successfully extends the STACK plugin in Moodle to support interactive graph theory exercises that go beyond static images or symbolic input. The implementation enables teachers to define randomized parameters for question generation, and students to solve algorithmic tasks through direct interaction with graph elements.

The following core functionalities have been achieved:

- Use of JSXGraph within the existing STACK environment to render interactive graph canvases.
- Parameterized graph generation using Maxima, including:
 - Random selection of vertex count within a defined range.
 - Controlled edge density and automatic validation to ensure that the generated graph is both connected and pedagogically suitable for algorithm-based learning tasks.
 - Randomized assignment of edge weights for weighted algorithms.
- Simple and intuitive user interaction with the graph, allowing students to select or deselect nodes and edges directly through mouse clicks, without requiring any textual input.
- Custom evaluation scripts were implemented in Maxima for all supported graph algorithms (DFS, BFS, Dijkstra, and Kruskal), enabling automated correctness checks based on student input.

- Automatic validation of student responses through STACK’s Potential Response Trees (PRTs), with immediate feedback indicating whether the submitted solution is correct.

This framework enables the creation of unique question instances for each student, supports algorithm-specific evaluation logic, and promotes interactive learning in a digital environment.

5.2. Verification and Testing

The development and testing process was inherently sequential due to the structure of STACK questions. The creation of a functional exercise began in the `QuestionVariables` section, where parameters such as the number of vertices, edge density, and weight ranges were defined. These parameters were used to construct an initial graph in the form of an adjacency matrix, which served as the foundation for generating randomized graph instances in each question. Once the graph data was validated and rendered correctly in the `QuestionText` section using `JSXGraph`, it became possible to implement interactive elements and proceed to the development of the evaluation logic in the `FeedbackVariables` section.

Testing had to be performed at each stage before the next could begin. To verify the correctness of values generated in `QuestionVariables`, HTML was used in the `QuestionText` section to manually display variables, since Moodle STACK does not provide native tools for inspecting intermediate values. This method only worked if the entire Maxima code compiled without syntax errors, which made early debugging slow and restrictive.

In contrast, JavaScript debugging was more manageable, as the code executed client-side in the browser. Console output allowed real-time inspection of user interaction, making it easier to verify behavior during graph rendering and input capture.

Once earlier stages were stable, the evaluation logic was implemented and tested in the `FeedbackVariables` section. Dozens of randomized test cases were generated to verify the correctness of algorithm-specific validation. Both correct and incorrect solutions were manually submitted to check whether the system could reliably accept or reject them. This led to the discovery of a logic flaw in the initial BFS evaluation script, which was corrected after further testing.

Key aspects confirmed through testing include:

- Correct generation and rendering of randomized graph data in `JSXGraph`.

- User actions on the interactive graph (like selecting nodes or edges) were accurately reflected in STACK's input variables for evaluation
- Functional evaluation logic for all supported algorithms, with reliable detection of incorrect student answers and accurate feedback delivery through STACK's PRT system.

5.3. Limitations

While the developed system fulfills its intended purpose and enables functional, interactive graph algorithm assessments within STACK, several limitations were encountered during development. These arose from technical constraints and structural properties of Moodle and STACK.

- **Steep learning curve for Moodle and STACK:** Developing questions in Moodle with STACK requires a solid understanding of both platforms. A lack of foundational knowledge or incorrect assumptions about how STACK operates can easily lead to flawed design strategies or inefficient approaches, especially when attempting more complex, interactive tasks.
- **Restricted development environment:** Programming inside Moodle is limited by design. There is no proper debugging system, especially for the Maxima code executed on the server. Errors are often presented as vague messages, and trial-and-error debugging becomes necessary. Displaying variable values requires workarounds like printing to HTML — and only works if the code is syntactically correct.
- **Maxima behaves differently inside STACK:** While Maxima is well-documented as a standalone system, its use within Moodle/STACK is subject to sandboxing for security reasons. As a result, certain functions may be restricted or behave differently. Since not all of these differences are prominently documented within the STACK environment, unexpected behavior can occur, especially during initial development and debugging.
- **Limited modularity due to security policies:** Due to security restrictions defined by the Moodle system configuration (e.g., at Hochschule Hannover), it is not possible to include external files or separate logic into reusable modules. All code must be written directly within the three main STACK sections (`QuestionVariables`, `QuestionText`, and `FeedbackVariables`). This constraint limits the ability to structure code cleanly and makes it more difficult to reuse or maintain complex logic.

- **Sequential development constraint:** Because STACK questions are processed in a strict top-down order (QuestionVariables \rightarrow QuestionText \rightarrow FeedbackVariables), each section must be fully functional before the next can be developed. This dependency often delays the implementation of evaluation logic until all prior components are stable.
- **No formal user evaluation conducted:** While the system was tested thoroughly in development, it has not yet been evaluated in a real-world classroom setting. Its effectiveness for teaching and learning in practice remains to be studied.

These limitations reflect both technical boundaries of the STACK system and broader workflow constraints that arise when working within a controlled LMS environment like Moodle.

5.4. Conclusion

This thesis presented the design and implementation of a system for interactive graph algorithm exercises within the STACK plugin in Moodle. The goal was to support the teaching of classical graph algorithms—such as DFS, BFS, Dijkstra, and Kruskal—through randomized, interactive questions that allow students to engage directly with graph structures.

The system builds on existing integration between STACK and JSXGraph to render interactive graphs, and uses Maxima to generate randomized graph instances and evaluate student responses. Each supported algorithm is accompanied by a custom evaluation script, capable of verifying the correctness of student-submitted solutions based on algorithm-specific logic.

Throughout development, particular attention was paid to ensuring structural validity and pedagogical suitability of the generated graphs. The system includes mechanisms for edge filtering, graph connectivity validation, and weight assignment, making it adaptable to a range of question types and difficulty levels.

Testing was performed at each stage of development to confirm the reliability of graph generation, interaction capture, and answer evaluation. While no formal user study was conducted, the system was tested extensively across various parameter configurations to verify correct functionality.

Several technical limitations were encountered during development. These include constraints related to STACK's sequential structure, limited debugging capabilities, and the sandboxed behavior of Maxima within Moodle. Despite these challenges, the system

demonstrates that it is possible to implement rich, interactive algorithmic assessments using STACK's existing functionality.

The result is a reusable framework that enables instructors to create dynamic graph exercises and supports more engaging student interaction with algorithmic content. This work lays a foundation for further improvements and future research in the area of interactive mathematics education within learning management systems.

5.5. Future Work

While the current system fulfills its intended goals, several directions for future development could further enhance its functionality and educational value:

- Enabling students to build graphs by adding or removing nodes and edges, rather than only selecting existing elements.
- Conducting a formal classroom study to evaluate the pedagogical effectiveness of the system in real learning environments.
- Implementing a mechanism to allow students to revise or correct their submitted answers after receiving feedback.
- Providing detailed feedback with suggestions or even the correct solution in cases where the submitted answer is incorrect.

The development process offered valuable insights into the technical and practical aspects of integrating dynamic content into learning platforms. Working within the constraints of Moodle and STACK highlighted both the flexibility and the limitations of these systems. The project strengthened my skills in structured debugging, algorithm implementation, and system design under real-world restrictions.

Overview of Tools Used

Tool	Used for
DeepL	Verifying that the text sections correctly and purposefully reflect the intended meaning
ChatGPT	Generating ideas for new approaches, checking academic wording, and providing initial assistance with debugging and code correctness

A. Appendix: Source Code Listings

A.1. BFS Algorithm

A.1.1. BFS Question Variables

Listing A.1: BFS_QuestionVariables.js

```
/* Minimum and maximum number of vertices to include in a
   generated subgraph */
min_vertex : 4;
max_vertex : 7;

/* Percentage of edges to retain from the initial full graph
   */
edge_density : 60;

/* Coordinates for visual placement of all 13 vertices on the
   JSXGraph canvas */
vertex_positions : [
  [3,3],
  [2,4], [4,4], [2,2], [4,2],
  [0.5,3], [5.5,3], [3,5.5], [3,0.5],
  [1,5], [5,5], [5,1], [1,1]
];

/* Full adjacency matrix defining the initial complete graph
   structure.
   This matrix contains all possible valid edges between the
   13 vertices,
   carefully arranged to avoid visual overlap or edge
   crossings.
   All subgraphs are generated as subsets of this graph.
   */
initial_graph : matrix(
  [0,1,1,1,1,0,0,0,0,0,0,0,0],
  [1,0,1,1,0,1,0,1,0,1,0,0,0],
```

```

[1,1,0,0,1,0,1,1,0,0,1,0,0],
[1,1,0,0,1,1,0,0,1,0,0,0,1],
[1,0,1,1,0,0,1,0,1,0,0,1,0],
[0,1,0,1,0,0,0,0,0,1,0,0,1],
[0,0,1,0,1,0,0,0,0,0,1,1,0],
[0,1,1,0,0,0,0,0,0,1,1,0,0],
[0,0,0,1,1,0,0,0,0,0,0,1,1],
[0,1,0,0,0,1,0,1,0,0,0,0,0],
[0,0,1,0,0,0,1,1,0,0,0,0,0],
[0,0,0,0,1,0,1,0,1,0,0,0,0],
[0,0,0,1,0,1,0,0,1,0,0,0,0]
);

reduce_edges(M, p) := block(
  [n, i, j, new_M],
  n : length(M),
  filtered_M : copy(M),
  for i:1 thru n do (
    for j:i+1 thru n do (
      if M[i][j] = 1 and rand(100) > p then (
        filtered_M[i][j] : 0,
        filtered_M[j][i] : 0
      )
    )
  ),
  return(filtered_M)
)$

graph_is_valid(M, vertex_num) := block(
  [n, visited, dfs_tree, result],
  result : true,
  n : length(M),
  visited : makelist(false, i, 1, n),
  dfs_tree : zeromatrix(n, n),
  dfs(M, n, 1, visited, dfs_tree),

  if has_isolated_node(dfs_tree, vertex_num) or equal(M,
    dfs_tree) then
    result : false
  else
    result : true,
  return(result)
)$

```

```

has_isolated_node(M, vertex_num) := block(
  [i, j, rowsum, returnVal],
  returnVal: false,
  for i:1 thru vertex_num do (
    rowsum : 0,
    for j:1 thru vertex_num do (
      rowsum : rowsum + M[i][j] + M[j][i]
    ),
    if rowsum = 0 then (returnVal:true)
  ),
  return(returnVal)
)$

dfs(M, n, node, visited, R) := block(
  [i],
  if not visited[node] then (
    visited[node] : true,
    for i:1 thru n do (
      if (M[node][i] = 1 or M[i][node] = 1) and not visited[i]
      ] then (
        R[node][i] : 1,
        R[i][node] : 1,
        dfs(M, n, i, visited, R)
      )
    )
  )
)$

vertex_num: min_vertex + rand(max_vertex - min_vertex + 1);

first_node: 1+ rand(vertex_num);

n: length(initial_graph);
initial_graph: genmatrix(
  lambda([i, j],
    if i <= vertex_num and j <= vertex_num then initial_graph[
      i, j] else 0
  ),
  n, n
);

graph_valid: false;

while not graph_valid do (

```

```
    filtered_edges: reduce_edges(initial_graph , edge_density),
    edge_density: edge_density+1,
    graph_valid: graph_is_valid(filtered_edges , vertex_num)
)

edges_list : args(matrixmap(lambda([r], r), filtered_edges));
```

A.1.2. BFS Question Text

```
<p></p>
<p hidden>[[input:ans1]] [[validation:ans1]]</p>
<p hidden>[[input:ans2]] [[validation:ans2]]</p>

[[jsxgraph input-ref-ans1="ans1Ref" input-ref-ans2="ans2Ref"]]

var input1 = document.getElementById(ans1Ref);
var input2 = document.getElementById(ans2Ref);

function updateAns() {
  var matrixExpr = "matrix(";
  for (var i = 0; i < 13; i++) {
    matrixExpr += "[" + ergMatrix[i].join(",") + " ";
    if (i < 12) matrixExpr += ",";
  }
  matrixExpr += ")";

  if (input1) {
    input1.value = matrixExpr;
    input1.dispatchEvent(new Event('change'));
  } else {
    console.log("input1 not found");
  }

  if (input2) {
    let sequenceWithPadding = selectedVertexList.slice();
    while (sequenceWithPadding.length < vertexNum) {
      sequenceWithPadding.push(0);
    };
    input2.value = "[" + sequenceWithPadding.join(",") + "]"
    ;
    input2.dispatchEvent(new Event('change'));
  } else {
```

```
        console.log("input2 not found");
    }
}

function vertexSelect(point) {
    counter++;
    point.setAttribute({color: 'blue'});

    var newName = point.baseName + " (" + counter + ")";
    changeName(point, newName);

    selectedVertexList.push(point.index+1);
    updateAns();
}

function vertexDeselect(point) {
    counter--;
    selectedVertexList.pop();
    var newName = point.baseName;
    changeName(point, newName);
    point.setAttribute({color: 'red'});
    updateAns();
}

function edgeSelect(edge) {
    ergMatrixUpdate(edge, 1);
    edge.StrokeBase = ['blue', 3];
    changeEdgeStroke(edge, edge.StrokeBase);
}

function edgeDeselect(edge) {
    ergMatrixUpdate(edge, 0);
    edge.StrokeBase = ['red', 1];
    changeEdgeStroke(edge, edge.StrokeBase);
}

function changeEdgeStroke(edge, newStrk){
    edge.setAttribute({strokeColor: newStrk[0], strokeWidth:
        newStrk[1]});
}

function changeName(obj, givenName){
```

```
    obj.setAttribute({name: givenName});
  }

  function ergMatrixUpdate(edge, val){
    let p1idx = edge.point1.index;
    let p2idx = edge.point2.index;
    ergMatrix[p1idx][ p2idx  ] = val;
    ergMatrix[p2idx][ p1idx  ] = val;
    updateAns();
  }

  var board = JXG.JSXGraph.initBoard(divid, {
    boundingbox: [0, 6, 6, 0],
    axis: false,
    grid: false,
    showNavigation: false,
    showCopyright: false
  });

  var vertex_positions = {#vertex_positions#};
  var vertexNum = {#vertex_num#};
  var edges = {#edges_list#};
  var flatMtr=[];
  var first_node= {#first_node#};

  var edgesSize = edges.length;

  let ergMatrix = Array.from({ length: edgesSize }, () =>
    Array.from({ length: edgesSize }, () => 0)
  );

  var counter = 0;
  var pointList = [];
  var edgeClickable =true;

  var vertexOrderList = Array(vertexNum).fill(0);
  var selectedVertexList = [];
  var selectedEdgeList = [];

  for (var i = 0; i < vertexNum; i++) {
    var point = board.create('point', vertex_positions[i], {
      color: '#FF0000',
      size: 10,
      highlight: false,
    });
  }
}
```

```

        showInfobox: false,
        fixed: true
    });
    point.baseName = point.name;
    point.on('down', function(evt) {

        if(selectedVertexList[selectedVertexList.length - 1] ===
            this.index+1){
            vertexDeselect(this);}
        else if (!selectedVertexList.includes(this.index+1)){
            vertexSelect(this);
        }

    });

    point.on('mouseover', function(evt) {

        edgeClickable=false;
    });

    point.on('mouseout', function(evt) {

        edgeClickable=true;
    });

    point.index = i;
    pointList.push(point);
}
console.log(edges);

var first_nodeIndx = first_node-1;
console.log(first_nodeIndx);
vertexSelect(pointList[first_nodeIndx]);
pointList[first_nodeIndx].off('down');

for (var i = 0; i < vertexNum; i++) {
    for (var j = i+1; j < vertexNum; j++) {
        if (edges[i][j] != 0) {
            var edge = board.create('segment', [pointList[i],
                pointList[j]], {
                strokeColor: 'black',
                strokeWidth: 1,
                highlight: false,
                withLabel: false,

```

```

        label: {
            position: 'middle',
            offset: [0, 0],
            cssStyle: 'background-color: white; font-size:
                    14px; font-weight: bold;'
        }
    });

    edge.StrokeBase = ['black', 1];
    changeEdgeStroke(edge, edge.StrokeBase);

    edge.on('mouseover', function(evt) {
        if (edgeClickable) {

            changeEdgeStroke(this, ['lightblue', 2]);
        }
    });
    edge.on('mouseout', function(evt) {
        changeEdgeStroke(this, this.StrokeBase);
    });

    edge.on('down', function(evt) {
        if (!edgeClickable) {
            return;
        }
        var index1 = this.point1.index;
        var index2 = this.point2.index;
        if(      (ergMatrix[index1][ index2  ] === 0)
            ||
            (ergMatrix[index2][ index1  ] === 0)
        ){
            edgeSelect(this);

        }else{
            edgeDeselect(this);
        }
    });
    }
}
}
}

```

[[/jsxgraph]]</p>

Listing A.2: BFS_QuestionTextV.js

A.1.3. BFS Feedback Variables

Listing A.3: BFS_FeedbackVariables.js

```

ansMatrix: ans1;
ansNodeOrder : copylist(ans2);

result : true;
result_matrix : genmatrix(lambda([i,j], 0), 13, 13);

n: vertex_num;
M: filtered_edges;
start_node: first_node;
visited : makelist(false, i, 1, n);
queue : [start_node];
visited[start_node] : true;
bfs_order : [];

if first(ansNodeOrder) # start_node then (
  result : false
);
ansNodeOrder : rest(ansNodeOrder);

while queue # [] and result do (
  node : first(queue),
  queue : rest(queue),
  bfs_order : endcons(node, bfs_order),

  sub_queue : [], /* move this here */
  for i : 1 thru n do (
    if M[node, i] = 1 and not visited[i] then (
      visited[i] : true,
      result_matrix[node, i] : 1,
      result_matrix[i, node] : 1,
      sub_queue : endcons(i, sub_queue)
    )
  ),

  ans_sub_queue : [],
  for j : 1 thru length(sub_queue) do (
    if length(ansNodeOrder) = 0 then (
      result : false
    )
    else (
      ans_node : first(ansNodeOrder),

```

```
    ansNodeOrder : rest(ansNodeOrder),
    if not member(ans_node, sub_queue) then (
      result : false
    ),
    ans_sub_queue : endcons(ans_node, ans_sub_queue)
  )
),
queue : append(queue, ans_sub_queue)
);

if (not equal(result_matrix, ansMatrix)) then result : false;
```

A.2. DFS Algorithm

A.2.1. DFS Question Variables

Listing A.4: DFS_QuestionVariables.js

```
/* Minimum and maximum number of vertices to include in a
   generated subgraph */
min_vertex : 4;
max_vertex : 7;

/* Percentage of edges to retain from the initial full graph
   */
edge_density : 60;

/* Coordinates for visual placement of all 13 vertices on the
   JSXGraph canvas */
vertex_positions : [
  [3,3],
  [2,4], [4,4], [2,2], [4,2],
  [0.5,3], [5.5,3], [3,5.5], [3,0.5],
  [1,5], [5,5], [5,1], [1,1]
];

/* Full adjacency matrix defining the initial complete graph
   structure.
   This matrix contains all possible valid edges between the
   13 vertices,
```

```

    carefully arranged to avoid visual overlap or edge
    crossings.
    All subgraphs are generated as subsets of this graph.
*/
initial_graph : matrix(
  [0,1,1,1,1,0,0,0,0,0,0,0,0],
  [1,0,1,1,0,1,0,1,0,1,0,0,0],
  [1,1,0,0,1,0,1,1,0,0,1,0,0],
  [1,1,0,0,1,1,0,0,1,0,0,0,1],
  [1,0,1,1,0,0,1,0,1,0,0,1,0],
  [0,1,0,1,0,0,0,0,0,1,0,0,1],
  [0,0,1,0,1,0,0,0,0,0,1,1,0],
  [0,1,1,0,0,0,0,0,0,1,1,0,0],
  [0,0,0,1,1,0,0,0,0,0,0,1,1],
  [0,1,0,0,0,1,0,1,0,0,0,0,0],
  [0,0,1,0,0,0,1,1,0,0,0,0,0],
  [0,0,0,0,1,0,1,0,1,0,0,0,0],
  [0,0,0,1,0,1,0,0,1,0,0,0,0]
);

reduce_edges(M, p) := block(
  [n, i, j, new_M],
  n : length(M),
  filtered_M : copy(M),
  for i:1 thru n do (
    for j:i+1 thru n do (
      if M[i][j] = 1 and rand(100) > p then (
        filtered_M[i][j] : 0,
        filtered_M[j][i] : 0
      )
    )
  ),
  return(filtered_M)
)$

graph_is_valid(M, vertex_num) := block(
  [n, visited, dfs_tree, result],
  result : true,
  n : length(M),
  visited : makelist(false, i, 1, n),
  dfs_tree : zeromatrix(n, n),
  dfs(M, n, 1, visited, dfs_tree),

```

```

    if has_isolated_node(dfs_tree, vertex_num) or equal(M,
        dfs_tree) then
        result : false
    else
        result : true,
    return(result)
)$

has_isolated_node(M, vertex_num) := block(
    [i, j, rowsum, returnVal],
    returnVal: false,
    for i:1 thru vertex_num do (
        rowsum : 0,
        for j:1 thru vertex_num do (
            rowsum : rowsum + M[i][j] + M[j][i]
        ),
        if rowsum = 0 then (returnVal:true)
    ),
    return(returnVal)
)$

dfs(M, n, node, visited, R) := block(
    [i],
    if not visited[node] then (
        visited[node] : true,
        for i:1 thru n do (
            if (M[node][i] = 1 or M[i][node] = 1) and not visited[i]
                then (
                    R[node][i] : 1,
                    R[i][node] : 1,
                    dfs(M, n, i, visited, R)
                )
        )
    )
)$

vertex_num: min_vertex + rand(max_vertex - min_vertex + 1);

first_node: 1+ rand(vertex_num);

n: length(initial_graph);
initial_graph: genmatrix(
    lambda([i, j],

```

```

        if i <= vertex_num and j <= vertex_num then initial_graph[
            i, j] else 0
    ),
    n, n
);

graph_valid: false;

while not graph_valid do (

    filtered_edges: reduce_edges(initial_graph , edge_density),
    edge_density: edge_density+1,
    graph_valid: graph_is_valid(filtered_edges , vertex_num)
)

edges_list : args(matrixmap(lambda([r], r), filtered_edges));

```

A.2.2. DFS Question Text

```

<p></p>
<p hidden>[[input:ans1]] [[validation:ans1]]</p>
<p hidden>[[input:ans2]] [[validation:ans2]]</p>

[[jsxgraph input-ref-ans1="ans1Ref" input-ref-ans2="ans2Ref"]]

var input1 = document.getElementById(ans1Ref);
var input2 = document.getElementById(ans2Ref);

function updateAns() {
    var matrixExpr = "matrix(";
    for (var i = 0; i < 13; i++) {
        matrixExpr += "[" + ergMatrix[i].join(",") + " ";
        if (i < 12) matrixExpr += ",";
    }
    matrixExpr += ")";

    if (input1) {
        input1.value = matrixExpr;
        input1.dispatchEvent(new Event('change'));
    } else {
        console.log("input1 not found");
    }
}

```

```
    if (input2) {
      let sequenceWithPadding = selectedVertexList.slice();
      while (sequenceWithPadding.length < vertexNum) {
        sequenceWithPadding.push(0);
      };
      input2.value = "[" + sequenceWithPadding.join(",") + "]"
      ;
      input2.dispatchEvent(new Event('change'));
    } else {
      console.log("input2 not found");
    }
  }

  function vertexSelect(point) {
    counter++;
    point.setAttribute({color: 'blue'});

    var newName = point.baseName + " (" + counter + ")";
    changeName(point, newName);

    selectedVertexList.push(point.index+1);
    updateAns();

  }

  function vertexDeselect(point) {
    counter--;
    selectedVertexList.pop();
    var newName = point.baseName;
    changeName(point, newName);
    point.setAttribute({color: 'red'});
    updateAns();
  }

  function edgeSelect(edge) {
    ergMatrixUpdate(edge, 1);
    edge.StrokeBase = ['blue', 3];
    changeEdgeStroke(edge, edge.StrokeBase);
  }

  function edgeDeselect(edge) {
    ergMatrixUpdate(edge, 0);
    edge.StrokeBase = ['red', 1];
    changeEdgeStroke(edge, edge.StrokeBase);
  }

```

```
}  
function changeEdgeStroke(edge, newStrk){  
  edge.setAttribute({strokeColor: newStrk[0], strokeWidth:  
    newStrk[1]});  
}  
  
function changeName(obj, givenName){  
  obj.setAttribute({name: givenName});  
}  
  
function ergMatrixUpdate(edge, val){  
  let p1idx = edge.point1.index;  
  let p2idx = edge.point2.index;  
  ergMatrix[p1idx][ p2idx  ] = val;  
  ergMatrix[p2idx][ p1idx  ] = val;  
  updateAns();  
}  
  
var board = JSXGraph.initBoard(divid, {  
  boundingbox: [0, 6, 6, 0],  
  axis: false,  
  grid: false,  
  showNavigation: false,  
  showCopyright: false  
});  
  
var vertex_positions = {#vertex_positions#};  
var vertexNum = {#vertex_num#};  
var edges = {#edges_list#};  
var flatMtr=[];  
var first_node= {#first_node#};  
  
var edgesSize = edges.length;  
  
let ergMatrix = Array.from({ length: edgesSize }, () =>  
  Array.from({ length: edgesSize }, () => 0)  
);  
  
var counter = 0;  
var pointList = [];  
var edgeClickable =true;
```

```
var vertexOrderList = Array(vertexNum).fill(0);
var selectedVertexList = [];
var selectedEdgeList = [];

for (var i = 0; i < vertexNum; i++) {
  var point = board.create('point', vertex_positions[i], {
    color: '#FF0000',
    size: 10,
    highlight: false,
    showInfobox: false,
    fixed: true
  });
  point.baseName = point.name;
  point.on('down', function(evt) {

    if(selectedVertexList[selectedVertexList.length - 1] ===
      this.index+1){
      vertexDeselect(this);}
    else if (!selectedVertexList.includes(this.index+1)){
      vertexSelect(this);
    }
  });

  point.on('mouseover', function(evt) {

    edgeClickable=false;
  });

  point.on('mouseout', function(evt) {

    edgeClickable=true;
  });

  point.index = i;
  pointList.push(point);
}
console.log(edges);

var first_nodeIndx = first_node-1;
console.log(first_nodeIndx);
vertexSelect(pointList[first_nodeIndx]);
pointList[first_nodeIndx].off('down');
```

```

for (var i = 0; i < vertexNum; i++) {
  for (var j = i+1; j < vertexNum; j++) {
    if (edges[i][j] != 0) {
      var edge = board.create('segment', [pointList[i],
        pointList[j]], {
        strokeColor: 'black',
        strokeWidth: 1,
        highlight: false,
        withLabel: false,
        label: {
          position: 'middle',
          offset: [0, 0],
          cssStyle: 'background-color: white; font-size:
            14px; font-weight: bold;'
        }
      });

      edge.StrokeBase = ['black', 1];
      changeEdgeStroke(edge, edge.StrokeBase);

      edge.on('mouseover', function(evt) {
        if (edgeClickable) {

          changeEdgeStroke(this, ['lightblue', 2]);
        }
      });
      edge.on('mouseout', function(evt) {
        changeEdgeStroke(this, this.StrokeBase);
      });

      edge.on('down', function(evt) {
        if (!edgeClickable) {
          return;
        }
        var index1 = this.point1.index;
        var index2 = this.point2.index;
        if(      (ergMatrix[index1][ index2  ] === 0)
            ||
            (ergMatrix[index2][ index1  ] === 0)
        ){
          edgeSelect(this);
        }else{
          edgeDeselect(this);
        }
      });
    }
  }
}

```

```
        }
      });
    }
  }
}

[[/jsxgraph]]</p>
```

Listing A.5: DFS_QuestionText.js

A.2.3. DFS Feedback Variables

Listing A.6: DFS_FeedbackVariables.js

```
ansMatrix: ans1;
ansNodeOrder : copylist(ans2);

result : true;
result_matrix : genmatrix(lambda([i,j], 0), 13, 13);

n: vertex_num;
M: filtered_edges;
start_node: first_node;
visited : makelist(false, i, 1, n);

nodePointer: 2;
parentPointer: 1;
visited[ansNodeOrder[parentPointer]]: true;

while (nodePointer <= n) do (
  node : ansNodeOrder[nodePointer],
  parent : ansNodeOrder[parentPointer],
  if node > 0 then (
    if ((equal(M[node][parent], 1) or equal(M[parent][node],
      1)) and not visited[node]) then (
      result_matrix[node][parent] : 1,
      result_matrix[parent][node] : 1,
      visited[node] : true,
      parentPointer : nodePointer,
      nodePointer : nodePointer + 1
    ) else block(
      for i : 1 thru n do (
        if ((equal(M[parent][i], 1) or equal(M[i][parent], 1))
          and not visited[i]) then (
```

```

        result : false,
        nodePointer : n + 1
    )
),
parentPointer : parentPointer - 1,
if parentPointer <= 0 then (
    result : false,
    nodePointer : n + 1
)
)
) else (
    result : false,
    nodePointer : n + 1
)
);

if (not equal(result_matrix, ansMatrix)) then result : false;

```

A.3. Kruskal Algorithm

A.3.1. Kruskal Question Variables

Listing A.7: kruskal_QuestionVariables.js

```

/* Minimum and maximum number of vertices to include in a
   generated subgraph */
min_vertex : 4;
max_vertex : 9;

/* Percentage of edges to retain from the initial full graph
   */
edge_density : 60;

/* Range of possible edge weights for weighted graph types (e.
   g., Kruskal, Dijkstra) */
min_weight : 1;
max_weight : 9;

/* Coordinates for visual placement of all 13 vertices on the
   JSXGraph canvas */
vertex_positions : [

```

```

    [3,3],
    [2,4], [4,4], [2,2], [4,2],
    [0.5,3], [5.5,3], [3,5.5], [3,0.5],
    [1,5], [5,5], [5,1], [1,1]
];

/* Full adjacency matrix defining the initial complete graph
structure.
This matrix contains all possible valid edges between the
13 vertices,
carefully arranged to avoid visual overlap or edge
crossings.
All subgraphs are generated as subsets of this graph.
*/
initial_graph : matrix(
  [0,1,1,1,1,0,0,0,0,0,0,0,0],
  [1,0,1,1,0,1,0,1,0,1,0,0,0],
  [1,1,0,0,1,0,1,1,0,0,1,0,0],
  [1,1,0,0,1,1,0,0,1,0,0,0,1],
  [1,0,1,1,0,0,1,0,1,0,0,1,0],
  [0,1,0,1,0,0,0,0,0,1,0,0,1],
  [0,0,1,0,1,0,0,0,0,0,1,1,0],
  [0,1,1,0,0,0,0,0,0,1,1,0,0],
  [0,0,0,1,1,0,0,0,0,0,0,1,1],
  [0,1,0,0,0,1,0,1,0,0,0,0,0],
  [0,0,1,0,0,0,1,1,0,0,0,0,0],
  [0,0,0,0,1,0,1,0,1,0,0,0,0],
  [0,0,0,1,0,1,0,0,1,0,0,0,0]
);

reduce_edges(M, p) := block(
  [n, i, j, new_M],
  n : length(M),
  filtered_M : copy(M),
  for i:1 thru n do (
    for j:i+1 thru n do (
      if M[i][j] = 1 and rand(100) > p then (
        filtered_M[i][j] : 0,
        filtered_M[j][i] : 0
      )
    )
  ),
  return(filtered_M)
)$

```

```

graph_is_valid(M, vertex_num) := block(
  [n, visited, dfs_tree, result],
  result : true,
  n : length(M),
  visited : makelist(false, i, 1, n),
  dfs_tree : zeromatrix(n, n),
  dfs(M, n, 1, visited, dfs_tree),

  if has_isolated_node(dfs_tree, vertex_num) or equal(M,
    dfs_tree) then
    result : false
  else
    result : true,
  return(result)
)$

has_isolated_node(M, vertex_num) := block(
  [i, j, rowsum, returnVal],
  returnVal: false,
  for i:1 thru vertex_num do (
    rowsum : 0,
    for j:1 thru vertex_num do (
      rowsum : rowsum + M[i][j] + M[j][i]
    ),
    if rowsum = 0 then (returnVal:true)
  ),
  return(returnVal)
)$

dfs(M, n, node, visited, R) := block(
  [i],
  if not visited[node] then (
    visited[node] : true,
    for i:1 thru n do (
      if (M[node][i] = 1 or M[i][node] = 1) and not visited[i]
      ] then (
        R[node][i] : 1,
        R[i][node] : 1,
        dfs(M, n, i, visited, R)
      )
    )
  )
)$

```

```

assign_random_weights(M, min_weight, max_weight) := block(
  [n, i, j, W],
  n : length(M),
  W : copy(M),
  for i : 1 thru n do (
    for j : i+1 thru n do (
      if M[i][j] # 0 then (
        W[i][j] : min_weight + rand(max_weight - min_weight +
          1),
        W[j][i] : W[i][j]
      )
    )
  ),
  return(W)
)$

vertex_num: min_vertex + rand(max_vertex - min_vertex + 1);

n: length(initial_graph);
initial_graph: genmatrix(
  lambda([i, j],
    if i <= vertex_num and j <= vertex_num then initial_graph[
      i, j] else 0
  ),
  n, n
);

graph_valid: false;

while not graph_valid do (
  filtered_edges: reduce_edges(initial_graph , edge_density),
  edge_density: edge_density+1,
  graph_valid: graph_is_valid(filtered_edges, vertex_num)
)

filtered_edges: assign_random_weights(filtered_edges,
  min_weight, max_weight );

edges_list : args(matrixmap(lambda([r], r), filtered_edges));

```

A.3.2. Kruskal Question Text

```

<p></p>
<p hidden>[[input:ans1]] [[validation:ans1]]</p>

[[jsxgraph input-ref-ans1="ans1Ref"]]
var ans = document.getElementById(ans1Ref);
function updateAns() {
  var matrixExpr = "matrix(";

  for (var i = 0; i < 13; i++) {
    matrixExpr += "[" + ergMatrix[i].join(",") + "];";
    if (i < 12) matrixExpr += ",";
  }
  matrixExpr += ")";
  console.log(matrixExpr );

  if (ans) {
    ans.value = matrixExpr;
    ans.dispatchEvent(new Event('change'));
  }else{
    console.log("no input found");
  }
}

function edgeSelect(edge) {
  if (selectedEdgeList.includes(edge)){return;}
  counter++;
  changeName(edge, edge.weight + " (" +counter +")" );
  ergMatrixUpdate(edge, counter);
  edge.StrokeBase = ['blue', 3];
  changeEdgeStroke(edge, edge.StrokeBase);
  selectedEdgeList.push(edge);
}

function edgeDeselect() {
  edge = selectedEdgeList.pop();
  counter--;
  changeName(edge, edge.weight );
  edge.StrokeBase = ['red', 1];
  ergMatrixUpdate(edge, 0);
  changeEdgeStroke(edge, edge.StrokeBase);
}

function changeEdgeStroke(edge, newStrk){

```

```
    edge.setAttribute({strokeColor: newStrk[0], strokeWidth:
        newStrk[1]});
}

function changeName(obj, givenName){
    obj.setAttribute({name: givenName});
}

function ergMatrixUpdate(edge, val){
    let p1idx = edge.point1.index;
    let p2idx = edge.point2.index;
    ergMatrix[p1idx][p2idx] = val;
    ergMatrix[p2idx][p1idx] = val;

    updateAns();
}

var board = JXG.JSXGraph.initBoard(divid, {
    boundingbox: [0, 6, 6, 0],
    axis: false,
    grid: false,
    showNavigation: false,
    showCopyright: false
});

var vertex_positions = {#vertex_positions#};
var vertexNum = {#vertex_num#};
var edges = {#edges_list#};

var edgesSize = edges.length;

let ergMatrix = Array.from({ length: edgesSize }, () =>
    Array.from({ length: edgesSize }, () => 0)
);

var counter = 0;
var pointList = [];

var edgeClickable =true;
var selectedEdgeList = [];

for (var i = 0; i < vertexNum; i++) {
    var point = board.create('point', vertex_positions[i], {
        color: '#FF0000',
```

```
        name: i+1,
        size: 10,
        highlight: false,
        showInfobox: false,
        fixed: true
    });

    point.on('mouseover', function(evt) {

        edgeClickable=false;
    });

    point.on('mouseout', function(evt) {

        edgeClickable=true;
    });

    point.index = i;
    pointList.push(point);
}

for (var i = 0; i < vertexNum; i++) {
    for (var j = i+1; j < vertexNum; j++) {
        if (edges[i][j] != 0) {
            var edge = board.create('segment', [pointList[i],
                pointList[j]], {

                strokeColor: 'black',
                strokeWidth: 1,
                highlight: false,
                withLabel: true,
                label: {
                    position: 'middle',
                    offset: [0, 0],
                    cssStyle: 'background-color: white; font-size: 14
                        px; font-weight: bold;'
                }
            });

            edge.weight = edges[i][j];
            changeName(edge, edge.weight);
            edge.StrokeBase = ['black', 1];
            changeEdgeStroke(edge, edge.StrokeBase);
```

```

edge.on('mouseover', function(evt) {
  if (edgeClickable) {

    changeEdgeStroke(this,['lightblue',2]);
  }
});
edge.on('mouseout', function(evt) {
  changeEdgeStroke(this, this.StrokeBase);
});

edge.on('down', function(evt) {
  //prevent Clicking over verticies
  if (!edgeClickable) {
    return;
  }
  if(selectedEdgeList[selectedEdgeList.length - 1] ===
    this){
    edgeDeselect();
  }else{
    edgeSelect(this);
  }
});
}
}
}

```

[[/jsxgraph]]</p>

Listing A.8: kruskal_QuestionText.js

A.3.3. Kruskal Feedback Variables

Listing A.9: kruskal_FeedbackVariables.js

```

/* Global find with parent */
find(x, parent) := if parent[x] = x then x else (parent[x] :
  find(parent[x], parent))$

/* Global union with parent */
unionEdge(x, y, parent) := (parent[find(x, parent)] : find(y,
  parent))$

/* Convert weighted adjacency matrix to sorted edge list */

```

```

matrix_to_sorted_edges(M) := block(
  [n, edges : [], i, j],
  n : length(M),
  for i:1 thru n do (
    for j:i+1 thru n do (
      if M[i][j] # 0 then (
        edges : cons([M[i][j], [i, j]], edges)
      )
    )
  ),
  return(sort(edges, lambda([a, b], a[1] < b[1])))
)$

/* Kruskal MST builder */
kruskal_mst(edgeList, vertexCount) := block(
  [parent, mst_edges : [], i, u, v, w],
  parent : makelist(i, i, 1, vertexCount),
  for i : 1 thru length(edgeList) do (
    w : edgeList[i][1],
    u : edgeList[i][2][1],
    v : edgeList[i][2][2],

    if find(u, parent) # find(v, parent) then (
      unionEdge(u, v, parent),
      mst_edges : endcons([w, [u, v]], mst_edges)
    )
  ),
  return(mst_edges)
)$

result : true; /* Assume the answer is correct at start */

sortedResult : matrix_to_sorted_edges(ans1); /* Sorted student
  answer */
sortedFilterdMatrix : matrix_to_sorted_edges(filtered_edges);
/* Sorted correct edges */
originalSortedResult: copy(sortedResult);
originalSortedFilterd: copy(sortedFilterdMatrix);
parent : makelist(i, i, 1, vertex_num)$ /* Initialize disjoint
  set */
stepsDebug:["test"];
selectedEdges : 0; /* Counter for correct edges found */

while length(sortedResult) > 0 do (

```

```

sr1 : first(sortedResult), /* first student edge */
sr1U : sr1[2][1],
sr1V : sr1[2][2],
sr1W:filtered_edges[sr1U][sr1V],
sfm1 : first(sortedFilterdMatrix), /* first correct edge
*/

if sr1W> sfm1[1] then ( /* student skipped lighter edge */
  u : sfm1[2][1],
  v : sfm1[2][2],
  if find(u, parent) = find(v, parent) then ( /* does it
    form a loop? */
      sortedFilterdMatrix : rest(sortedFilterdMatrix) /*
        skip this edge */

    ) else ( /* it should have been selected */

      result : false,
      sortedFilterdMatrix : rest(sortedFilterdMatrix),
      stepsDebug: endcons("false1",stepsDebug ),
      return()

    )
) else if sr1W < sfm1[1] then ( /* student selected too
  small weight */
  result : false,
  stepsDebug: endcons("false2",stepsDebug ),
  sortedResult : rest(sortedResult),
  return()

) else ( /* weights are equal, now check if connection is
  valid */

  if find(sr1U, parent) = find(sr1V, parent) then ( /*
    loop found in student MST */

    result : false,
    stepsDebug: endcons("false3",stepsDebug ),
    sortedResult : rest(sortedResult),
    return()

  ) else ( /* correct edge */
    unionEdge(sr1U, sr1V, parent), /* update
      connection */

```

```

        selectedEdges : selectedEdges + 1, /* count valid
            edge */
        sortedResult : rest(sortedResult), /* remove edge
            from student list */
        removingElement: [sr1W,[sr1U,sr1V ]],
        stepsDebug: endcons("rightInput",stepsDebug ),
        sortedFilterdMatrix : delete(removingElement,
            sortedFilterdMatrix) /* remove from correct list
            */
    )
)
);
if (selectedEdges < vertex_num -1 ) then (
    result : false,
    stepsDebug: endcons("rightInpnot enough edge",stepsDebug )
);

```

A.4. Dijkstra Algorithm

A.4.1. Dijkstra Question Variables

Listing A.10: Dijkstra_QuestionVariables.js

```

/* Minimum and maximum number of vertices to include in a
    generated subgraph */
min_vertex : 4;
max_vertex : 9;

/* Percentage of edges to retain from the initial full graph
    */
edge_density : 60;

/* Range of possible edge weights for weighted graph types (e.
    g., Kruskal, Dijkstra) */
min_weight : 1;
max_weight : 9;

/* Coordinates for visual placement of all 13 vertices on the
    JSXGraph canvas */
vertex_positions : [

```

```

    [3,3],
    [2,4], [4,4], [2,2], [4,2],
    [0.5,3], [5.5,3], [3,5.5], [3,0.5],
    [1,5], [5,5], [5,1], [1,1]
];

/* Full adjacency matrix defining the initial complete graph
structure.
This matrix contains all possible valid edges between the
13 vertices,
carefully arranged to avoid visual overlap or edge
crossings.
All subgraphs are generated as subsets of this graph.
*/
initial_graph : matrix(
  [0,1,1,1,1,0,0,0,0,0,0,0,0],
  [1,0,1,1,0,1,0,1,0,1,0,0,0],
  [1,1,0,0,1,0,1,1,0,0,1,0,0],
  [1,1,0,0,1,1,0,0,1,0,0,0,1],
  [1,0,1,1,0,0,1,0,1,0,0,1,0],
  [0,1,0,1,0,0,0,0,0,1,0,0,1],
  [0,0,1,0,1,0,0,0,0,0,1,1,0],
  [0,1,1,0,0,0,0,0,0,1,1,0,0],
  [0,0,0,1,1,0,0,0,0,0,0,1,1],
  [0,1,0,0,0,1,0,1,0,0,0,0,0],
  [0,0,1,0,0,0,1,1,0,0,0,0,0],
  [0,0,0,0,1,0,1,0,1,0,0,0,0],
  [0,0,0,1,0,1,0,0,1,0,0,0,0]
);
reduce_edges(M, p) := block(
  [n, i, j, new_M],
  n : length(M),
  filtered_M : copy(M),
  for i:1 thru n do (
    for j:i+1 thru n do (
      if M[i][j] = 1 and rand(100) > p then (
        filtered_M[i][j] : 0,
        filtered_M[j][i] : 0
      )
    )
  ),
  return(filtered_M)
)$

```

```

graph_is_valid(M, vertex_num) := block(
  [n, visited, dfs_tree, result],
  result : true,
  n : length(M),
  visited : makelist(false, i, 1, n),
  dfs_tree : zeromatrix(n, n),
  dfs(M, n, 1, visited, dfs_tree),

  if has_isolated_node(dfs_tree, vertex_num) or equal(M,
    dfs_tree) then
    result : false
  else
    result : true,
  return(result)
)$

has_isolated_node(M, vertex_num) := block(
  [i, j, rowsum, returnVal],
  returnVal: false,
  for i:1 thru vertex_num do (
    rowsum : 0,
    for j:1 thru vertex_num do (
      rowsum : rowsum + M[i][j] + M[j][i]
    ),
    if rowsum = 0 then (returnVal:true)
  ),
  return(returnVal)
)$

dfs(M, n, node, visited, R) := block(
  [i],
  if not visited[node] then (
    visited[node] : true,
    for i:1 thru n do (
      if (M[node][i] = 1 or M[i][node] = 1) and not visited[i]
        then (
          R[node][i] : 1,
          R[i][node] : 1,
          dfs(M, n, i, visited, R)
        )
    )
  )
)$

```

```

assign_random_weights(M, min_weight, max_weight) := block(
  [n, i, j, W],
  n : length(M),
  W : copy(M),
  for i : 1 thru n do (
    for j : i+1 thru n do (
      if M[i][j] # 0 then (
        W[i][j] : min_weight + rand(max_weight - min_weight +
          1),
        W[j][i] : W[i][j]
      )
    )
  ),
  return(W)
)$

vertex_num: min_vertex + rand(max_vertex - min_vertex + 1);
first_node: 1+ rand(vertex_num);

n: length(initial_graph);
initial_graph: genmatrix(
  lambda([i, j],
    if i <= vertex_num and j <= vertex_num then initial_graph[
      i, j] else 0
  ),
  n, n
);

graph_valid: false;

while not graph_valid do (

  filtered_edges: reduce_edges(initial_graph , edge_density),
  edge_density: edge_density+1,
  graph_valid: graph_is_valid(filtered_edges , vertex_num)
)

filtered_edges: assign_random_weights(filtered_edges ,
  min_weight, max_weight );

edges_list : args(matrixmap(lambda([r], r), filtered_edges));

```

A.4.2. Dijkstra Question Text

```

<p></p>
<p hidden>[[input:ans1]] [[validation:ans1]]</p>
<p hidden>[[input:ans2]] [[validation:ans2]]</p>

[[jsxgraph input-ref-ans1="ans1Ref" input-ref-ans2="ans2Ref"]]

var input1 = document.getElementById(ans1Ref);
var input2 = document.getElementById(ans2Ref);

function updateAns() {
  var matrixExpr = "matrix(";
  for (var i = 0; i < 13; i++) {
    matrixExpr += "[" + selectedEdgeMatrix[i].join(",") +
      "];";
    if (i < 12) matrixExpr += ",";
  }
  matrixExpr += ")";

  if (input1) {
    input1.value = matrixExpr;
    input1.dispatchEvent(new Event('change'));
  } else {
    console.log("input1 not found");
  }

  if (input2) {
    let sequenceWithPadding = selectedVertexList.slice();
    while (sequenceWithPadding.length < vertexNum) {
      sequenceWithPadding.push(0);
    };
    input2.value = "[" + sequenceWithPadding.join(",") + "]"
      ;
    input2.dispatchEvent(new Event('change'));
  } else {
    console.log("input2 not found");
  }
}

function vertexSelect(point) {
  counter++;
  point.setAttribute({color: 'blue'});
  var newName = point.baseName + " (" + counter + ")";
  changeName(point, newName);
}

```

```
        selectedVertexList.push(point.index+1);
        updateAns();
    }

    function vertexDeselect(point) {
        counter--;
        selectedVertexList.pop();
        var newName = point.baseName;
        changeName(point, newName);
        point.setAttribute({color: 'red'});
        updateAns();
    }

    function edgeSelect(edge) {
        edgeMatrixUpdate(edge, 1);
        edge.StrokeBase = ['blue', 3];
        changeEdgeStroke(edge, edge.StrokeBase);
    }

    function edgeDeselect(edge) {
        edgeMatrixUpdate(edge, 0);
        edge.StrokeBase = ['black', 1];
        changeEdgeStroke(edge, edge.StrokeBase);
    }
    function changeEdgeStroke(edge, newStrk){
        edge.setAttribute({strokeColor: newStrk[0], strokeWidth:
            newStrk[1]});
    }

    function changeName(obj, givenName){
        obj.setAttribute({name: givenName});
    }

    function edgeMatrixUpdate(edge, val){
        let p1idx = edge.point1.index;
        let p2idx = edge.point2.index;
        selectedEdgeMatrix[p1idx][ p2idx ] = val;
        selectedEdgeMatrix[p2idx][ p1idx ] = val;
        updateAns();
    }

    var board = JXG.JSXGraph.initBoard(divid, {
        boundingbox: [0, 6, 6, 0],
```

```

    axis: false,
    grid: false,
    showNavigation: false,
    showCopyright: false
  });

  var vertex_positions = {#vertex_positions#};
  var vertexNum = {#vertex_num#};
  var edges = {#edges_list#};
  var first_node= {#first_node#};
  var edgesSize = edges.length;

  let selectedEdgeMatrix = Array.from({ length: edgesSize },
    () =>
      Array.from({ length: edgesSize }, () => 0)
  );

  var counter = 0;
  var pointList = [];
  var edgeClickable =true;

  var selectedVertexList = [];

  for (var i = 0; i < vertexNum; i++) {
    var point = board.create('point', vertex_positions[i], {
      color: '#FF0000',
      size: 10,
      highlight: false,
      showInfobox: false,
      fixed: true
    });
    point.baseName = point.name;
    point.on('down', function(evt) {

      if(selectedVertexList[selectedVertexList.length - 1] ===
        this.index+1){
        vertexDeselect(this);}
      else if (!selectedVertexList.includes(this.index+1)){
        vertexSelect(this);
      }
    });

    point.on('mouseover', function(evt) {
      edgeClickable=false;

```

```
});

point.on('mouseout', function(evt) {
    edgeClickable=true;
});

point.index = i;
pointList.push(point);
}

var first_nodeIndx = first_node-1;
vertexSelect(pointList[first_nodeIndx]);
pointList[first_nodeIndx].off('down');

for (var i = 0; i < vertexNum; i++) {
    for (var j = i+1; j < vertexNum; j++) {
        if (edges[i][j] != 0) {
            var edge = board.create('segment', [pointList[i],
                pointList[j]], {
                strokeColor: 'black',
                strokeWidth: 1,
                highlight: false,
                withLabel: true,
                label: {
                    position: 'middle',
                    offset: [0, 0],
                    cssStyle: 'background-color: white; font-size:
                        14px; font-weight: bold;'
                }
            });

            edge.weight = edges[i][j];
            changeName(edge, edge.weight);
            edge.StrokeBase = ['black', 1];
            changeEdgeStroke(edge, edge.StrokeBase);

            edge.on('mouseover', function(evt) {
                if (edgeClickable) {

                    changeEdgeStroke(this, ['lightblue', 2]);
                }
            });
            edge.on('mouseout', function(evt) {
                changeEdgeStroke(this, this.StrokeBase);
            });
        }
    }
}
```

```

    });

    edge.on('down', function(evt) {
        //prevent Clicking over verticies
        if (!edgeClickable) {
            return;
        }

        var index1 = this.point1.index;
        var index2 = this.point2.index;
        if((selectedEdgeMatrix[index1][ index2 ] === 0)
            ||
            (selectedEdgeMatrix[index2][ index1 ] === 0)
        ){
            edgeSelect(this);
        }else{
            edgeDeselect(this);
        }
    });
}
}
}

```

[[/jsxgraph]]</p>

Listing A.11: Dijkstra_QuestionText.js

A.4.3. Dijkstra Feedback Variables

Listing A.12: Dijkstra_FeedbackVariables.js

```

ansMatrix: ans1;
ansDistOrder: ans2;

/* Assume the answer is correct at start */
result : true;
result_matrix : genmatrix(lambda([i,j], 0), 13, 13);

/* Number of nodes */
n: vertex_num;
M: filtered_edges;
start_node: first_node;

```

```

/* Initialize distances and visited nodes */
d: makelist(inf, i, 1, n);
visited: makelist(false, i, 1, n);
d[start_node]: 0;

ansPointer: 1;
for ap: 1 thru n do(

    ansNode: ansDistOrder[ap],
    ansNodeDist: d[ansNode],

    for u: 1 thru n do (
        if not visited[u] and d[u] < ansNodeDist then (
            result: false,
            return()
        )
    ),

    /* Mark the selected node as visited */
    visited[ansNode]: true,

    /* Relaxation step: update neighbors */
    for v: 1 thru n do (
        if not visited[v] and M[ansNode][v] # 0 then (
            if d[ansNode] + M[ansNode][v] < d[v] then (
                d[v]: d[ansNode] + M[ansNode][v],

                /* Clear existing connections to v */
                for i: 1 thru n do (
                    result_matrix[i, v]: 0,
                    result_matrix[v, i]: 0
                ),

                /* Set the new edge in result_matrix */
                result_matrix[ansNode, v]: 1,
                result_matrix[v, ansNode]: 1
            )
        )
    )
);

if (not equal(result_matrix, ansMatrix)) then result : false;

```

Bibliography

- [Hoc24] Hochschule Hannover. Arbeiten mit moodle – infos für studierende. <https://www.hs-hannover.de/ueber-uns/organisation/servicezentrum-lehre/moodle-bbb-infos-fuer-studierende/arbeiten-mit-moodle>, 2024. Accessed April 2025.
- [KT14] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education Limited, Harlow, England, first edition, 2014.
- [Max24] Maxima Development Team. *Maxima Manual Version 5.47.0*, 2024. Accessed April 2025.
- [Moo24] Moodle Project. About moodle. https://docs.moodle.org/405/en/About_Moodle, 2024. Accessed April 2025.
- [OEC23] OECD. Education in the digital age: Paving the way for effective learning through technology. <https://www.oecd.org/education/education-in-the-digital-age-7f8d19e0-en.htm>, 2023. Accessed April 2025.
- [Spr07] Frauke Sprengel. Visual computing, 2007. Lecture script, Faculty IV, Department of Computer Science.
- [STA24a] STACK Development Team. Stack – computer algebra assessments in moodle. https://moodle.org/plugins/qtype_stack, 2024. Accessed April 2025.
- [STA24b] STACK Project. Discrete mathematics in stack. https://docs.stack-assessment.org/en/Topics/Discrete_mathematics/, 2024. Accessed April 2025.
- [STA24c] STACK Project. Stack assessment documentation. <https://docs.stack-assessment.org/en/>, 2024. Accessed April 2025.
- [STA24d] STACK Project. Stack documentation: Jsxgraph integration. https://docs.stack-assessment.org/en/Specialist_tools/JSXGraph/, 2024. Accessed April 2025.