The Calissons Puzzle

An interactive web app for automatic
puzzle generation and solving

Supervised by:
Dr Vincent van Oostrom

Computer Science
School of Engineering and Informatics
University of Sussex
2026

Statement of Originality

This report is submitted as part requirement for the degree of computer science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged. I hereby give permission for a copy of this report to be loaned out to students in future years.

Candidate: [redacted]

Signed: Ben Sharp

Date: 13/04/2026

Acknowledgments

A big thank you to my family, partner, and friends for being there for me throughout this project.

I would like to thank my supervisor, Dr Vincent van Oostrom, for all his advice, feedback, and kindness throughout this project.

I would also like to thank Olivier Longuet, and the authors of "*The Calissons Puzzle*" [1] for providing the groundwork upon which this project was built.

Abstract

The Calissons Puzzle is a geometric puzzle in which the player is presented a grid of triangles inside a hexagon, with a selection of the grid edges in bold. The player can place tiles anywhere on the grid, with each tile being one of three directions, signified by a respective colour. The goal of the puzzle is to fill the grid with tiles such that every bold edge is adjacent to tiles of different directions/colours. The resulting tiling resembles that of a 3D stepped surface.

The goal of this project was to create an interactive web application to procedurally generate calissons puzzles and visualize a solving algorithm for any given puzzle instance, taking design inspiration from modern minimalist puzzle games to remain intuitive and accessible to users with no prior knowledge of the puzzle.

The web app was implemented using React, TypeScript, and Vite. The core of the project, the solving algorithm, works by representing the calissons puzzle in 3D as a directed acyclic graph, and finding a valid partition of the set of graph nodes into two sets. The solving algorithm is utilized to guarantee generated puzzles are solvable, as well as generating solutions to said puzzles.

The application was tested on six participants, each with no prior knowledge of the puzzle. Participants were told the rules of the puzzle, then tasked with solving six puzzles, three without the application's tutorial, then three more after reading the tutorial. Participants were permitted to skip puzzles if they felt stuck. Testing ended with a brief questionnaire, with answers being on a scale from 1 to 5.

Before the tutorial, observed participants showed no difficulty interacting with the application's interface. The number of puzzles skipped dropped from 14 before the tutorial, to 6 after the tutorial. Average puzzle completion time improved from 2:34 to 2:07, and the average satisfaction for manual solving and automatic solving was 4.67/5 and 4.83/5 respectively.

Results suggest the application successfully conveys the puzzle to new users, with the in-app tutorial meaningfully improving both solve rate and completion time.

Contents

1

Introduction

The Calissons Puzzle (*le jeu des calissons*) was created in 2022 by Olivier Longuet, a French mathematics teacher. The puzzle was named after the calisson, a French candy traditionally associated with the French town of Aix-en-Provence [2]. The shape of these sweets resembles that of a rhombus or diamond, much like the tiles that make up the puzzle.

Figure 1.1: Calissons in a tin. [2]

A calisson (in context of the puzzle) is a rhombus comprised of two equilateral triangles joined on one side, they can either be red, blue, or yellow, depending on their direction. The puzzle's rules are simple: the player is given a triangular grid in the shape of a hexagon, alongside some edges to the grid. They must place calisson tiles such that no given edge is overlapped by a calisson, and each given edge is adjacent to two calissons of different colours (directions).

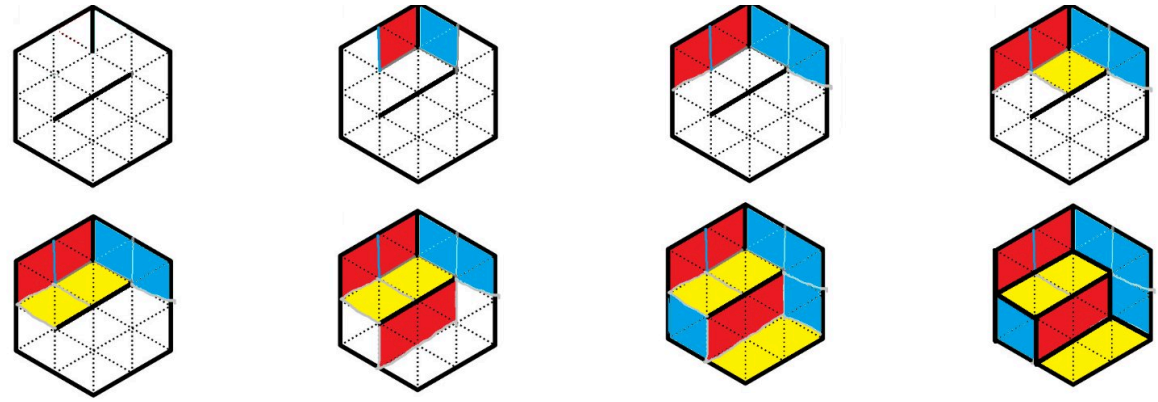


Figure 1.2: An example of a calissons puzzle, solved step-by-step. [3]

1.1 Project Overview

Outside of Longuet's blog [3], there exist limited ways to interact digitally with the calissons puzzle. This project addresses that gap by developing an interactive web application for playing procedurally generated calissons puzzles, incorporating a visualized automatic solving algorithm.

The application targets a broad audience, from young adults onwards, and was designed to be minimalistic and intuitive, taking inspiration from modern puzzle games such as Wordle [4], Pips [5], and 2048 [6]. React, TypeScript, and Vite were chosen to support responsive interaction and efficient rendering. A round of user testing was performed at the end of development to evaluate whether users with no prior knowledge of the puzzle could understand and interact with the application effectively.

1.2 Professional and Ethical Considerations

Due to user testing at the end of the project there were ethical considerations that needed to be taken into account. This project aligned with the BCS Code of Conduct (see section A.3), and the Software Testing With Users Compliance Form has been attached in the appendix (see section A.4).

For further detail as to how this project complied with ethical standards, see Appendix A.

2

Requirements Analysis

This chapter presents the objectives of this project, and the requirements needed to achieve these objectives.

2.1 Primary Objectives

1. Develop an interactive web-based visualization of the Calissons Puzzle.
2. Implement procedurally generated puzzles.
3. Visualize an automatic solving algorithm for any given puzzle instance.
4. Perform a round of user testing to evaluate success criteria.

2.2 Extension Objectives

5. Automatically assign a difficulty rating to any generated puzzle.
6. Implement a leaderboard for competitive player statistics (e.g. number of moves, time taken).
7. Ensure compatibility with most modern devices.
8. Puzzle save states and reset option.
9. A daily puzzle mode akin to Wordle.

2.3 Requirements

This section outlines the requirements necessary to achieve the primary objectives of this project. Requirements are labelled with the objective they correlate to.

2.3.1 User Requirements

The following user requirements describe the expected behaviour and functionality of the system from a user perspective.

Usability

- The system must provide an intuitive interface for puzzle interaction. (1)
- The system must present clear visual feedback for user actions. (1)

Interaction

- The system must allow users to place and remove calisson tiles on the grid. (1)
- The system must clearly convey when a puzzle is solved. (1)

- The system must allow users to generate new puzzles. (2)

Guidance

- The system must provide users with a concise tutorial of the rules and controls. (1)
- The system must provide users with a description of the application and its features. (1)

Solving Algorithm

- The system must provide users the ability to automatically generate a solution for any puzzle. (3)
- The system must visualize the solving algorithm in a clear manner. (3)

2.3.2 System Requirements

The following system requirements describe the expected behaviour and functionality of the system.

Correctness

- The system must ensure automatically generated puzzles are solvable. (2)
- The system must ensure automatically generated solutions are valid. (3)
- The system must handle invalid user inputs without causing runtime errors. (1)

Performance

- The system must efficiently generate reasonably sized puzzles. (1) (2)
- The system must efficiently generate solutions for reasonably sized puzzles. (1) (3)

3

Background Research

This chapter provides design and technical research into similar projects, required reading, and development tools.

3.1 Minimalist Puzzle Web Apps

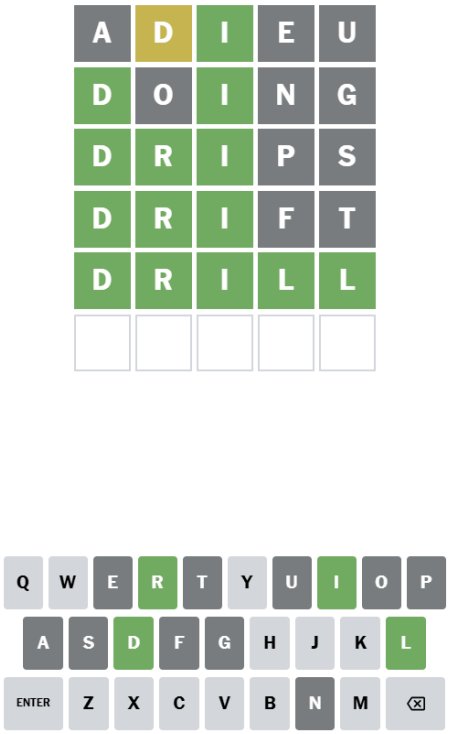


Figure 3.1: A solved Wordle puzzle.

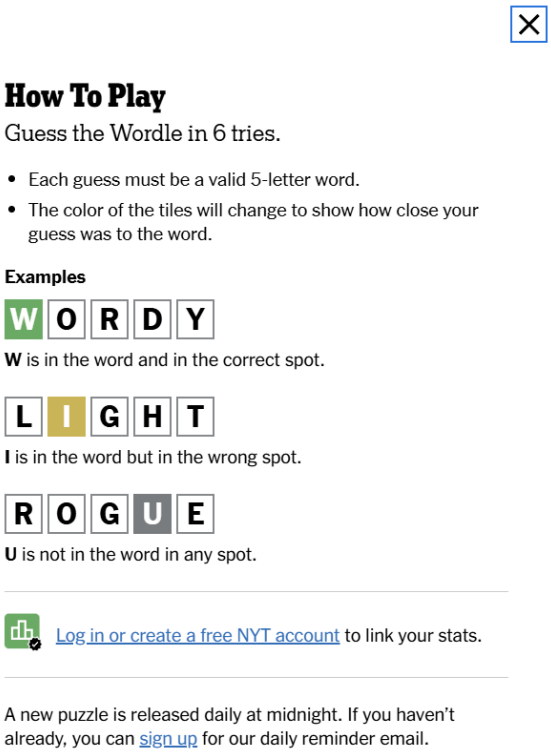


Figure 3.2: Wordle's tutorial screen that displays for first time players.

Three minimalist puzzle games informed the design of this project: Wordle [4], Pips [5], and 2048 [6]. Each demonstrated that a limited colour palette, simple win conditions, and accessible controls are sufficient to achieve broad appeal. Wordle's pop-up tutorial (3.2), Pips' drag-and-drop control scheme, and 2048's swipe gestures were all considered. The pop-up tutorial design was utilized for the final design, but the drag-and-drop was ultimately rejected in favour of point-and-click based on research by Inkpen [7].

3.2 Required Reading

The solving algorithm was implemented using the "advancing surface algorithm" from the paper "*The Calissons Puzzle*" [1] (see chapter 6). The concepts presented were non-trivial and required careful interpretation, posing a significant challenge during development.

3.3 Development Tools

The tools React, TypeScript, and Vite were chosen for this project. This section justifies these choices, and explains what each tool is for.

3.3.1 React

React was chosen as the front-end framework for this project due to its strong ecosystem, popularity (hence extensive documentation), and great synergy with Vite and TypeScript.

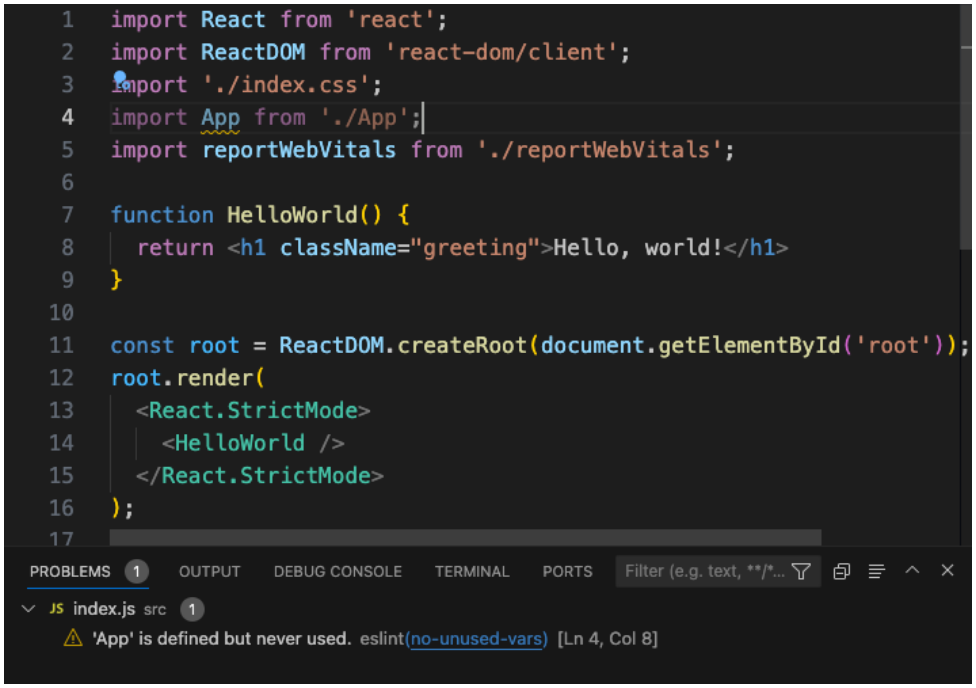


Figure 3.3: An example React program (hello world) written using JSX in VS Code.

React is a component-based [8] JavaScript library, when components change, it performs necessary calculations on those components alone, without having to spend computational power on components that haven't updated. This makes React good for performance, which is important for a lightweight project like mine.

3.3.2 TypeScript

TypeScript [9] extends JavaScript with static typing, catching type errors at compile-time rather than runtime. This was particularly valuable given the complex interactions between puzzle state, procedural generation, and the user interface.

3.3.3 Vite

Vite was chosen as the build tool for its fast startup times, live reloading, and easy integration with React and TypeScript. Vite is a frontend build tool that provides a faster and more efficient alternative to traditional tools. Vite monitors file edits/saves and updates the page without the need of a page refresh, through a process called Hot Module Replacement (HMR) [10]. Vite was chosen over other build tools (such as Webpack [11]) for its speed and ease of use.

4

Early Development

4.1 Creating the app

The initial step in the development of this project was the creation of a new React application using Vite. This process required npm, a package manager bundled with Node.js that handles the installation and management of project dependencies. The project was initialized using the following command:

```
npm create vite@latest calissons-puzzle -- --template react-ts
```

During development, a local server was run using:

```
npm run dev
```

This enabled rapid iteration and testing. For deployment, the application was built into static assets using:

```
npm run build
```

This process bundles and optimizes the application, producing a production-ready output suitable for hosting online (section 4.3).

4.2 Getting to Grips with React

Initial development required familiarization with React, a library with which I had no previous experience. This introduced many new concepts, the most notable being the use of hooks for state management, and React's component-based architecture.

4.2.1 React Components

React components are JavaScript functions that return markup for the user interface. Components can take parameters known as props, which allow for the component's appearance and behaviour to be dynamically controlled. This modularity and reusability helped maintain clean and readable code throughout the project, as well as saving on development time.

```
<TutorialModal
    show={showTutorial}
    onHide={() => {
        setShowTutorial(false);
        localStorage.setItem("hasSeenTutorial", "true");
    }}
/>
```

Listing 1: The tutorial modal component, with state and an event handler passed as props. The handler updates the visibility state and records that the tutorial has been viewed in local storage

4.2.2 Hooks and State Management

Several React hooks were used throughout the application to manage state and side effects. The *useState* hook stored local component state, such as the current grid size, tile placements, and puzzle completion status. The *useEffect* hook triggered actions in response to state changes, for example, detecting when the puzzle had been solved and stopping the timer accordingly (see Listing 3).

```
const [gridSize, setGridSize] = useState(2);
```

Listing 2: Example use of the *useState* hook to manage the puzzle grid size.

For this project, the use of hooks and states, combined with a logical component hierarchy, allowed data to be easily shared between components, whilst ensuring that any changes made to a state would be reflected throughout the entire user interface.

```
useEffect(() => {
    const solved = isPuzzleSolved(tiles, edges, gridSize);

    if (solved && !wasSolvedRef.current) {
        if (!autoSolved) setPuzzlesSolved((prev) => prev + 1);
        setIsSolved(true);

        if (startTime) {
            setElapsedTime(Date.now() - startTime); // Stop timer
        }
    }

    wasSolvedRef.current = solved;
}, [tiles]);
```

Listing 3: Use of the *useEffect* hook to detect when the puzzle has been solved and update application state accordingly.

4.3 Hosting the Application

The application required a hosting solution capable of serving static assets generated by the Vite build process. The original intention was to use the university servers, but as the project did not require a backend, a static site hosting platform was deemed most appropriate.

Several options were considered, including GitHub Pages and Netlify. GitHub Pages was not suitable as the repository remained private during development.

Netlify was selected as it provides seamless deployment for static applications, including automated build and deployment pipelines, and minimal configuration requirements [12]. This made it well-suited to the needs of the project and allowed for efficient deployment of the final application[1].

[1]The application is accessible on `https://thecalissonspuzzle.netlify.app/` until the end of the marking period at the least.

5

Puzzle Visualization and Interaction

This chapter focuses on the implementation and design choices of the puzzle's visualization and interaction. It's important to note that this chapter is not concerned with the solvability of the puzzle (see chapter 6).

5.1 Choice of Visualization Technology

Due to being a geometry puzzle, the choice of technology used to visualize the puzzle was very important to consider. The puzzle needed a visualization technology that was dynamic (due to user interaction), and scalable (due to device compatibility and dynamic grid sizes).

Scalable Vector Graphics (SVG) fit these criteria nicely. SVGs define shapes using mathematical formulas, allowing them to be scaled to any degree without losing resolution. The mathematical aspect of SVGs integrated well with the geometric nature of the puzzle, facilitating crisp visualization of 2D pixel projections.

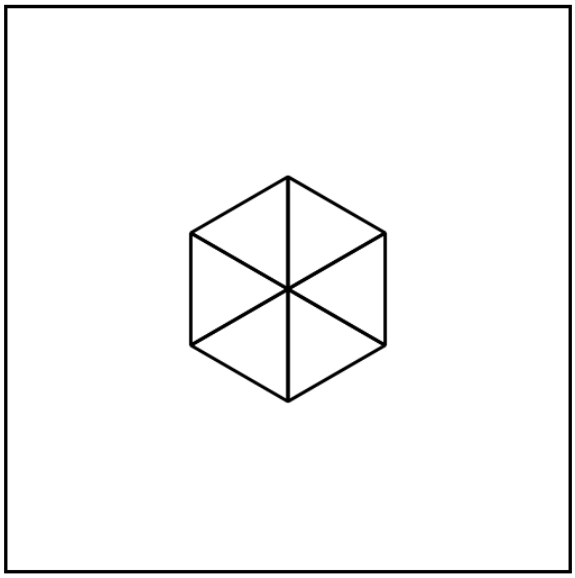


Figure 5.1: Early prototype to test the functionality of SVGs

The alternatives: canvas and plain HTML/CSS, were less suited to these requirements compared to SVG. They introduce greater complexity in handling interaction, as SVG elements exist within the DOM and can be directly manipulated via event handlers, whereas canvas-based rendering requires more manual interaction logic.

5.2 Data Structure

5.2.1 Choosing a Data Structure

A graph-based data structure was chosen to represent the puzzle grid, with grid points as nodes and edges connecting neighbours. As TypeScript has no built-in graph structure, two custom classes were implemented: *Node*, storing a value and adjacency list, and *Graph*, maintaining a collection of nodes and an *addEdge* function that assigns two nodes as mutual neighbours.

5.2.2 Building the Graph

The graph is built using a function that takes the size of the grid as a parameter, and returns a graph that represents the puzzle grid. An initial implementation used an axial coordinate system based on the *Red Blob Games* hexagonal grid guide [13], but this was refactored to a full 3D cube coordinate system to better align with the solving algorithm's coordinate system (see chapter 6).

When projecting a 3D cube to 2D, due to losing a dimension, multiple 3D points can project to the same point in 2D. In the context of "*The Calissons Puzzle*" [1] the projection function φ is defined as "the projection of the 3D space $\mathbb{R}^3$ onto a plane H of equation $x + y + z = h$ in the direction $(1, 1, 1)$". In simpler terms, this means that we are viewing the cube from a specific direction (facing a corner of the cube) and projecting the 3D cube coordinates along said direction. As shown in [1], we have

$$\varphi(x, y, z) = \varphi(x + k, y + k, z + k),$$

since points differing by a shift in the direction $(1, 1, 1)$ lie on the same projection line. As our goal is to draw a 2D puzzle grid, a graph with multiple nodes correlating to the same 2D point would be redundant. Using this logic, the refactored function iterates through all cube coordinates[1] (bounded by size), and calculates if the coordinate had been drawn before by iterating through all k (where k is height). A node is only be added to the graph if it's coordinate doesn't project to the same point as another node.

```
let drawn = false

for (const node of graph.nodes) {
    for (let k = 0; k <= size; k++) {
        if (q - k === node.value.q && r - k === node.value.r && s - k === node.value.s) {
            drawn = true
        }
    }
}
```

Listing 4: Section from graph building function that iterates through nodes, then iterates through k to check if the current coordinate minus k is a node already in the graph.

Nodes added to the graph are assigned a value consisting of their 3D coordinates, and their projected 2D coordinates. The projected 2D coordinates are derived from the 3D coordinates via a projection mapping from *"The problem of the calissons"* [14].

```
const px = -((r * -(Math.sqrt(3)/2)) + (q * (Math.sqrt(3)/2)))
const py = -((r * -(1/2)) + (q * -(1/2)) + (s))
```

Listing 5: Projection mapping used to translate 3D coordinates into 2D coordinates.

[1]Note that, for the 2D visualization implementation, coordinates q, r, and s were used instead of x, y, and z, to differentiate it from the distinct coordinate system used in the solving algorithm.

5.3 Time Complexity and Performance

The graph construction contains a triple nested iteration over the cube coordinates, giving a base complexity of $\mathcal{O}(n^3)$. Duplicate detection introduces an additional polynomial factor, though this poses no performance risk given the relatively small grid sizes used in this project. This could be optimized by populating the hash-based index during node generation rather than after, allowing duplicate checks in constant time, however this was not necessary given the project's scope.

As the graph represents a fixed-size grid, it does not need to be reconstructed for each new puzzle unless the grid size changes, minimizing unnecessary computation.

5.4 Drawing the Graph

After the graph has been constructed, it is rendered in SVG using the JavaScript *map* method to iterate over nodes and their neighbours, generating corresponding SVG elements using their stored 2D pixel coordinates. Boundary nodes, those with 4 or fewer neighbours, are identified and drawn with a thicker border edge. The function also handles the rendering of interactive nodes and generated puzzle edges[2], discussed in the sections below.

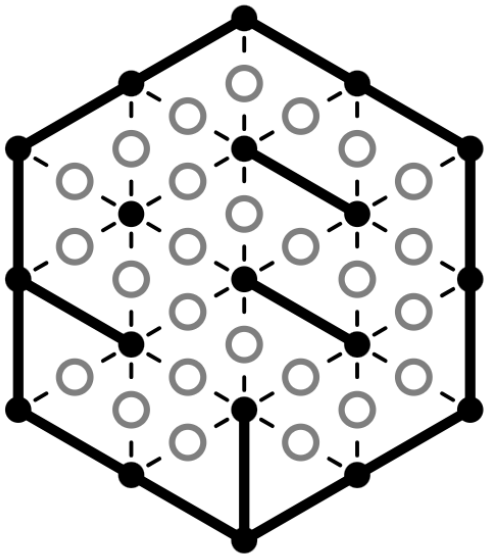


Figure 5.2: Result of the *DrawInteractiveGrid* function.

5.5 Interactivity

Interactive nodes are defined as nodes located between adjacent grid nodes, representing valid positions for user interaction. These nodes allow the user to place tiles by clicking or tapping on the corresponding location.

[2]Solvable edges are generated later in the project.

```
<circle
    cx={(node.value.px + n.value.px) / 2}
    cy={(node.value.py + n.value.py) / 2}

    onClick={() =>
        onNodeClick(
            findTilePoints(node, n),
            findTileNodes(node, n),
        )
    }

    ...
/>
```

Listing 6: Simplified SVG element highlighting the use of event handlers to trigger state updates via functions passed as props.

When an interactive node is clicked, the event handler will invoke the function passed as props, with the function taking the surrounding nodes as parameters, these surrounding nodes are used to draw the corresponding calisson tile. The tiles array is maintained as a state in the app component, which will redraw the tiles if a tile is added or removed. These functions are also invoked when a node is hovered, this is used to change the colour of the node to that of the tile that would be placed at said node.

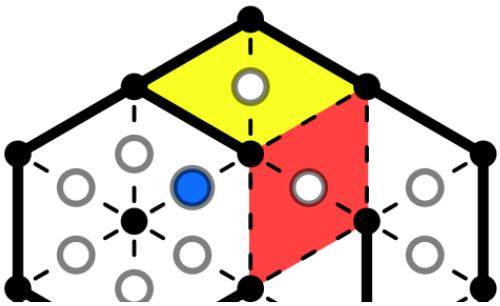


Figure 5.3: Tiles placed by clicking interactive nodes. Hovered node shows the colour of the tile that would be placed.

6

The Advancing Surface Algorithm

This chapter provides a simplified description the advancing surface algorithm outlined in [1]. For details into the implementation of the algorithm, see chapter 7.

Grasping the concepts from this paper was not trivial and took considerable time out of the project. Diagrams not cited from [1] were created for this report. These diagrams have been provided for areas that were particularly difficult to comprehend.

6.1 Overview

The advancing surface algorithm takes as input the grid size n, and the puzzle edges, and outputs a minimal solution[1] tiling. For regularly shaped puzzles, the algorithm runs in $\mathcal{O}(n^3)$ time. A solved calisson puzzle can be interpreted as a stepped surface of cubes inside a 3D space, the algorithm uses this conceptual 3D space to derive constraints from the input edges, which are used to construct a solution.

The 3D space uses the same coordinate system as the application's graph building function. This system assumes the grid we are viewing is the projection of a cube, with the stepped surface being the faces of smaller cubes placed inside this 3D space.

[1] Minimal solution meaning, within the 3D context, the solution stepped surface comprised of the minimal number of cubes.

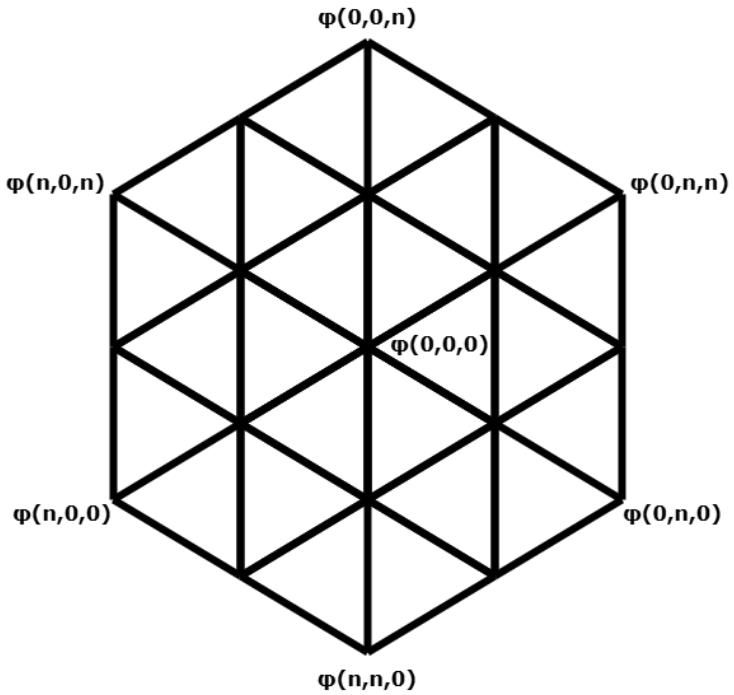


Figure 6.1: Grid of size n, with φ denoting the projected point of the corresponding coordinates.

6.2 Building the DAG

6.2.1 What is a DAG?

Important to the advancing surface algorithm is the notion of a directed acyclic graph (DAG), a graph of nodes and directed edges, where a path of nodes and edges will never result in a loop. DAGs are important to the algorithm as they represent the transitive relationship of a stepped surface. A simple way to represent a transitive relationship is: if $A \implies B$, and $B \implies C$, then $A \implies C$. This applies to stepped surfaces, as the presence of a cube at a given height implies the existence of cubes for all valid heights below it.

6.2.2 Cube Set

The DAG is built using three sets: the *Cube* set, the *Front* set, and the *Back* set. The *Cube* set is the set of all cubes that fit inside the 3D region. Following [1], each cube is represented by the coordinates of its lower corner (6.2).

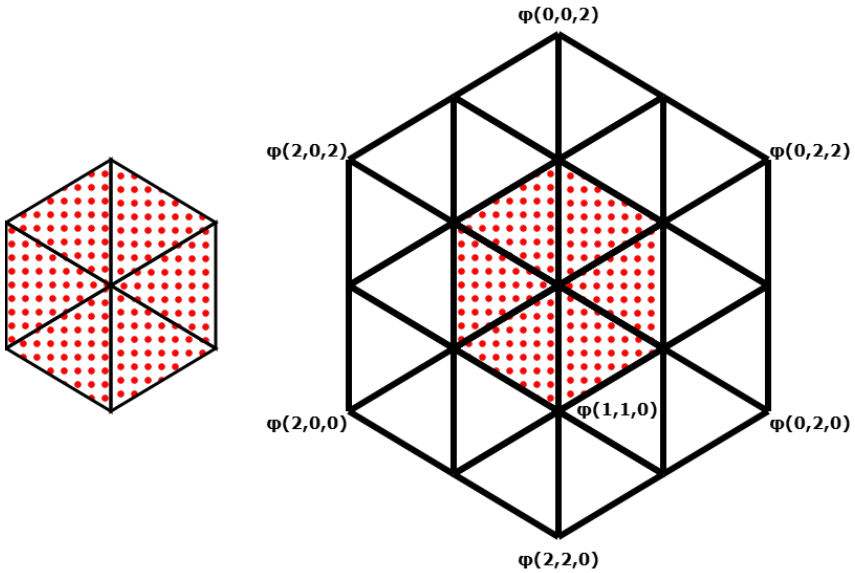


Figure 6.2: An example cube from the *Cube* set placed at $\varphi(1, 1, 0)$ on a grid of size 2. Due to perspective, this could also represent the cube at $\varphi(2, 2, 1)$.

6.2.3 Front and Back Sets

The sets *Front* and *Back* exist just outside the 3D region, acting somewhat like a padding layer. Formally the sets are defined in [1] as cubes "with two integral coordinates between 0 and $n - 1$" and the last coordinate being n for the *Front* set, and -1 for the *Back* set. This means that the sets sit outside the 3D region of the cube, with the *Front* set facing towards the figurative viewer, and the *Back* set facing away. The importance of these sets is made clear in later sections.

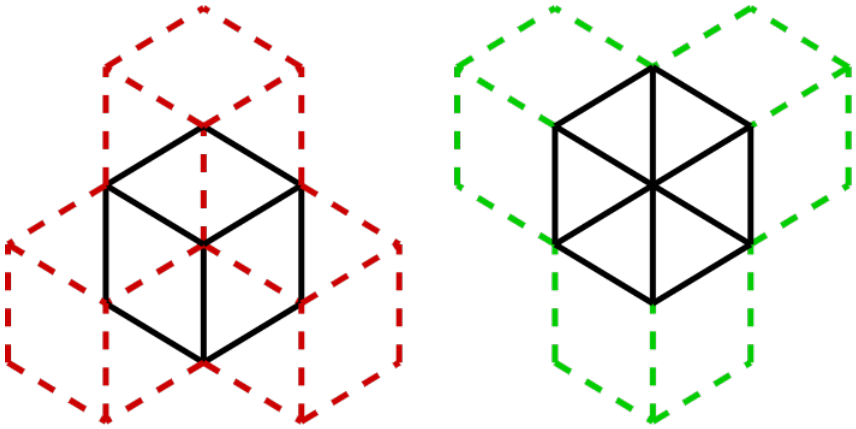


Figure 6.3: Sets *Front* (left) and *Back* (right) for a grid of size 1.

6.2.4 DAG Edges

Once the *Front*, *Back*, and *Cube* sets have been defined, directed edges are introduced to represent the relationships between cubes. We introduce the DAG structure on the set $Front \cup Cube \cup Back$ with an edge from any cube (x, y, z), to cubes $(x + 1, y, z)$, $(x, y + 1, z)$, and $(x, y, z + 1)$. This results in a DAG with edges rising from *Back* to *Front* (see Fig. 6.4).

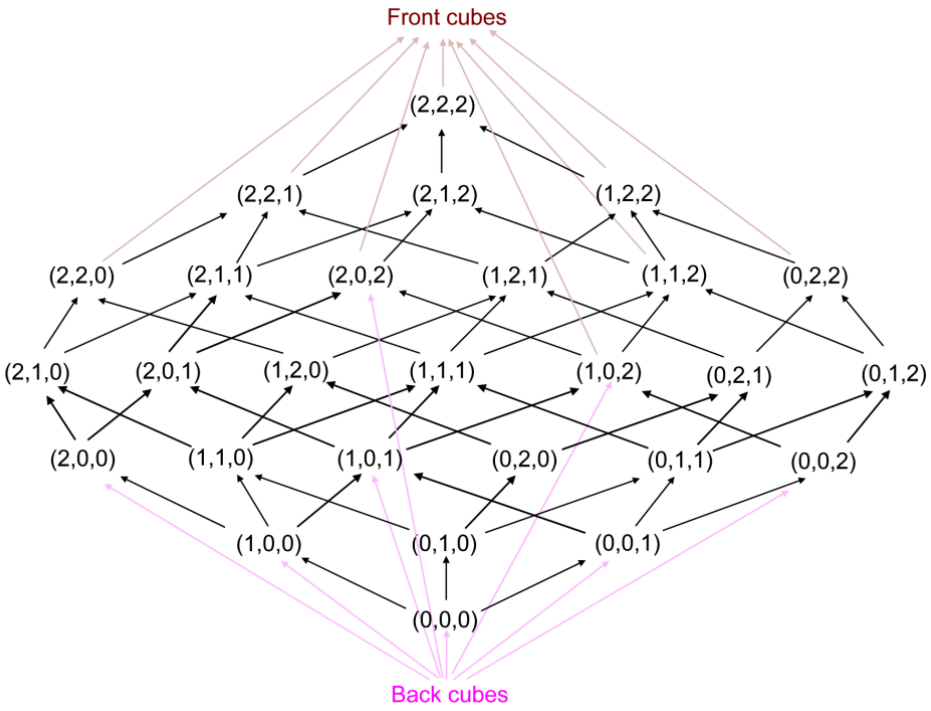


Figure 6.4: An example DAG for a grid of size 3, from [1]. Cubes of *Front* and *Back* are not individually represented.

6.2.5 DAG Cuts

The notion of a DAG cut is presented in [1]. A DAG cut is the partition of the set of nodes into two sets, a *low* set, and a *high* set. The *low* set contains all predecessors of its elements, and the *high* set contains all successors of its elements. The algorithm produces a valid solution by cutting the DAG, with the resulting stepped surface being the faces of the cubes left over from the cut, this is explained in more detail later.

6.3 Unbreakable Edges

Unbreakable edges are introduced in [1], they are the translation of inputted puzzle edges into undirected edges on the DAG that cannot be cut.

The algorithm creates unbreakable edges by determining the constraints intrinsically posed by a puzzle edge within the 3D space. As we are viewing the space from the direction of $(1, 1, 1)$, one puzzle edge can represent constraints on multiple cubes along said direction, meaning one edge usually creates several unbreakable edges.

6.3.1 Constraints

The paper [1] uses the notion of height k to represent where something sits along the $(1, 1, 1)$ direction. For a given puzzle edge, we represent the cubes this edge constraints at any k by F_k, B_k, L_k, R_k (see Fig. 6.5).

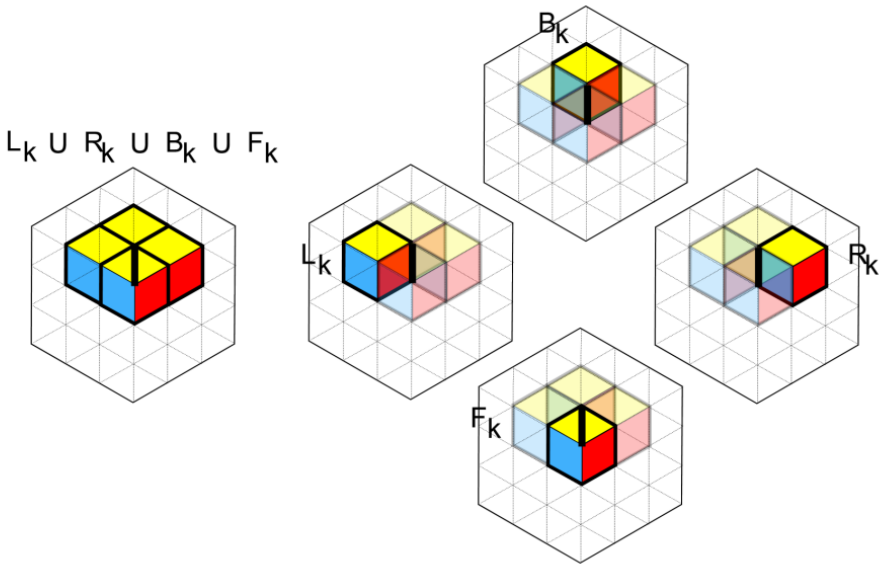


Figure 6.5: Example from [1] of the cubes F_k, B_k, L_k, R_k for a vertical edge at any k.

If the cubes F_k and B_{k+1} are separated by the cut, the common face of these cubes would overlap the edge, this violates the first rule of the puzzle, in that no tile can overlap an edge. Separating the cubes L_k and R_k would violate the second rule of the puzzle, in that tiles adjacent to an edge must be different directions, this is because the remaining faces of either L_k and B_k or R_k and B_k are of the same direction.

This means that for any valid height k, an unbreakable edge is created between the cubes F_k and B_{k+1}, and L_k and R_k. These edges cannot be cut, as cutting them would result in violating the conditions of the puzzle, hence they are "unbreakable".

6.4 Generating a Solution

Once the DAG is complete with directional edges and unbreakable edges, the algorithm can determine if the puzzle is solvable and produce a solution if so.

6.4.1 DAG to Solution

As briefly mentioned earlier, a solution is calculated by performing a DAG cut, partitioning the graph into two sets. A simple way to envision this is to think back to the *Cube* set. The *Cube* set includes all cubes that can fit inside the larger 3D cube space. Therefore, a valid DAG cut results in a separation of the *Cube* set, and the stepped surface we see is the faces of the remaining cubes.

A valid DAG cut is found by calculating all the nodes (cubes) now reachable from the *Front* set (due to unbreakable edges), and separating them from the *Back* and *Cube* sets.

To calculate the solution tiling from a valid DAG cut, a tile is drawn for every edge travelling from the *low* set into the *high* set. The direction of an outward edge correlates to the colour/direction of the resulting tile.

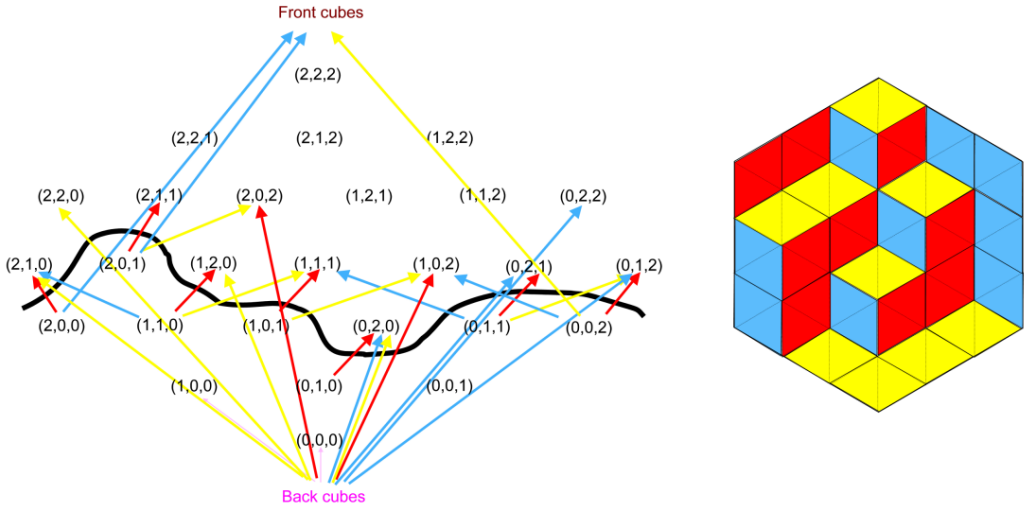


Figure 6.6: Example solution from [1]. On the left, a valid DAG cut with outward edges represented by the corresponding tile colour, with the resulting tiling on the right.

6.4.2 Checking Solvability

As unbreakable edges are undirected, they can create new paths between nodes that were impossible in the incomplete DAG. A calissons puzzle is deemed unsolvable if there exists no valid cut for the puzzle's completed DAG. This can be determined by calculating the nodes reachable from the *Front* set, if there exists a path from *Front* that connects to *Back*, the DAG cannot be cut, and the puzzle is unsolvable. Likewise, if there is no path connecting *Front* to *Back* then the puzzle is solvable.

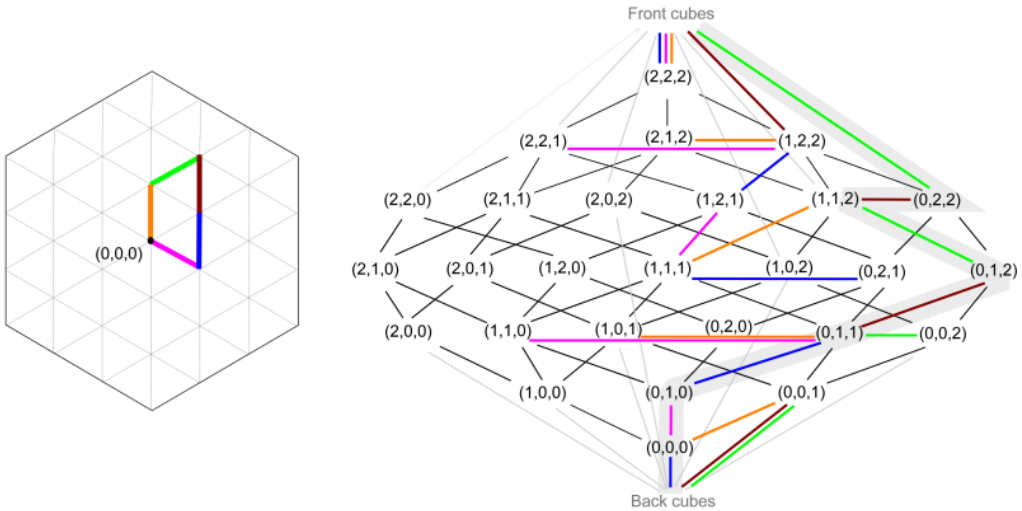


Figure 6.7: Example of an unsolvable puzzle from [1]. The unbreakable edges create a path from *Front* to *Back*, therefore a valid DAG cut is impossible.

6.4.3 Minimal Solution

To find the smallest solution for a given puzzle, we flip the direction of all DAG edges, such that *Back* now behaves like *Front*. We then do the same as before, by calculating the nodes reachable from *Back*, and separating them from the rest of the DAG. This will provide the smallest DAG cut, and therefore the smallest solution.

7

Implementing the Advancing Surface Algorithm

This chapter is dedicated to the implementation of the advancing surface algorithm [1]. See chapter 6 for an explanation of the algorithm.

7.1 Prerequisites

The existing graph class (5.2.1) was insufficient for the DAG structure required by the algorithm, so a new *SolverGraph* class was created alongside a *SolverNode* class. The new classes extend the original implementation with support for directed and unbreakable edges, separate inward and outward edge arrays for edge reversal, and a map index for constant time lookups.

7.2 Implementing the DAG

A function named *buildDAG* was written to construct the DAG. The function takes as input the grid size n and returns the corresponding DAG structure with all directional DAG edges.

7.2.1 Constructing Front, Cube, and Back Sets

Sets $Front$ and $Back$ were created as defined earlier, by using loops to iterate through 2 coordinates, and adding a node to the graph with said coordinates plus n or -1 respectively.

```
for (let y = 0; y < n; y++) {
    for (let z = 0; z < n; z++) {
        const cube: Cube3D = { id: nextId, x: -1, y, z};
        graph.addNode(nextId, cube);
        ...
    }
}

for (let x = 0; x < n; x++) {
    for (let z = 0; z < n; z++) {
        const cube: Cube3D = { id: nextId, x, y: -1, z};
        graph.addNode(nextId, cube);
        ...
    }
}

for (let x = 0; x < n; x++) {
    for (let y = 0; y < n; y++) {
        const cube: Cube3D = { id: nextId, x, y, z: -1};
        graph.addNode(nextId, cube);
        ...
    }
}
```

Listing 7: Simplified snippet from *buildDAG* used to construct the *Back* set. Construction of the *Front* set is near identical, but with n instead of -1.

The *Cube* set was created similarly, using 3 for loops to iterate through all possible coordinates within the 3D region, and adding the corresponding node to the graph. Nodes from all three sets are added to the index for efficient searching.

7.2.2 DAG Edges

Directed edges are constructed between nodes in the graph using an array of the three directions edges can be drawn along. We iterate through this array, and search the index for a node at the corresponding coordinate, if such a node exists, an edge is drawn between the source node and the target.

```
    for (const node of graph.nodes) {
        const { x, y, z } = node.value as Cube3D

        const neighbours = [
            [x + 1, y, z],
            [x, y + 1, z],
            [x, y, z + 1],
        ];

        for (const [nx, ny, nz] of neighbours) {
            const target = graph.index.get(`${nx},${ny},${nz}`);
            if (target) {
                graph.addDirectedEdge(node, target)
            }
        }
    }
```

Listing 8: Snippet from *buildDAG*. An array of possible neighbours is iterated through, and edges are added accordingly.

7.3 Implementing Unbreakable Edges

Translating the 2D puzzle edges to unbreakable edges posed a considerable challenge and resulted in a refactor to the original edge generation function. Edges can be drawn in three directions, with the cubes F_k, B_k, L_k, R_k requiring different calculations for each direction.

A function was implemented named *addUnbreakableEdges*, acting as a helper function that calls the appropriate function for each edge direction. These functions calculate the respective F_k, B_k, L_k, and R_k cubes, and construct the corresponding unbreakable edges.

```
    function addUnbreakableEdges(graph: SolverGraph, edges: Edge[], n: number) {
        for (const edge of edges) {
            if (edge.direction === "z") {
                addZAxisEdges(graph, edge.nodeA, n)
            }
            else if (edge.direction === "y") {
                addYAxisEdges(graph, edge.nodeA, n)
            }
            else if (edge.direction === "x") {
                addXAxisEdges(graph, edge.nodeA, n)
            }
        }
    }
```

Listing 9: Helper function *addUnbreakableEdges* which calls the function appropriate to the edge direction.

At this point, calling the functions we have defined in these sections would return a DAG complete with directional edges and unbreakable edges. This is all we need to produce a solution.

7.4 Generating Solvable Puzzles

To generate a guaranteed solvable puzzle, we need an algorithm that can determine if any given puzzle has a valid solution. We define solvability in subsection 6.4.2.

7.4.1 Solvability Check

To check solvability, we need to calculate all nodes reachable from the $Front$ set. This problem reduces to a reachability problem in the DAG. Connectivity to a given set can therefore be computed using a breadth-first search, which explores all reachable nodes in linear time $\mathcal{O}(V + E)$ [15].

A function was implemented named *frontReachesBack*. This function takes as input the DAG and the grid size n, and returns true if $Front$ connects to $Back$ (unsolvable). The breadth-first search implementation initializes the queue with all nodes from the $Front$ set, and computes connectivity using the DAG edges and unbreakable edges.

```
    while (queue.length > 0) {
        const current = queue.shift()!
        const cube = current.value as Cube3D

        if (isBack(cube, n)) {
            return true
        }

        for (const neighbour of current.outEdges) {
            if (!visited.has(neighbour)) {
                visited.add(neighbour)
                queue.push(neighbour)
            }
        }

        for (const neighbour of current.unbreakableEdges) {
            if (!visited.has(neighbour)) {
                visited.add(neighbour)
                queue.push(neighbour)
            }
        }
    }
```

Listing 10: Breadth-first search from *frontReachesBack*. Returns true if current node represents a cube from the $Back$ set and visits nodes from DAG edges and unbreakable edges.

7.4.2 Generating Solvable Edges

Using the *frontReachesBack* function, a new function named *generateSolvableEdges* was implemented. The function uses a simple do while loop to keep generating edges until the *frontReachesBack* function returns false, meaning the puzzle is solvable.

7.5 Generating Solution Tiling

Before computing a valid tiling, the algorithm finds the minimal solution as defined in subsection 6.4.3.

7.5.1 Computing Minimal Solution

To find the minimal solution we must compute the connective component of the $Back$ set, in a DAG where the directed edges are reversed. As mentioned before, *SolverNodes* contain an array of inward and outward edges, therefore we can use the inward edges array to essentially flip the DAG with little effort. A function *connectivityFromBack* computes nodes reachable from $Back$ using the same breadth-first search used in *frontReachesBack*. This function, unlike *frontReachesBack*, returns not a boolean but a set of all connected nodes. This set is the minimal solution.

7.5.2 Drawing the tiles

Once we have the minimal solution, the outward edges from the set of *Back* connected nodes to the rest of the DAG give us the corresponding tiling.

The function *solvePuzzle* computes the minimal solution, and iterates through the neighbours of the nodes in the solution set, if a neighbour isn't in the solution set, then the edge between these nodes correlates to a tile in the solution. For each valid outward edge, the function calls a helper function, passing in the direction of the edge, to push the respective tile to the array of tiles. This tiles array is returned upon termination. Implementing this step proved incredibly difficult due to the complexity and overlap of the systems required to translate a DAG edge into a drawable calisson tile.

```
if (direction === "x") {
        const { q, r, s } = addCubeCoords(
                { q: nodeA.value.q, r: nodeA.value.r, s: nodeA.value.s },
                CUBE_DIRECTIONS[0],
        );
        nodeB = graph.index.get(`${q},${r},${s}`);

        if (!nodeB) return null;

        points = findTilePoints(nodeA, nodeB);
        nodes = findTileNodes(nodeA, nodeB);
        fill = CALISSON_BLUE;
}
```

Listing 11: Snippet from tile drawing function. Draws a blue tile if the edge direction is along the x-axis.

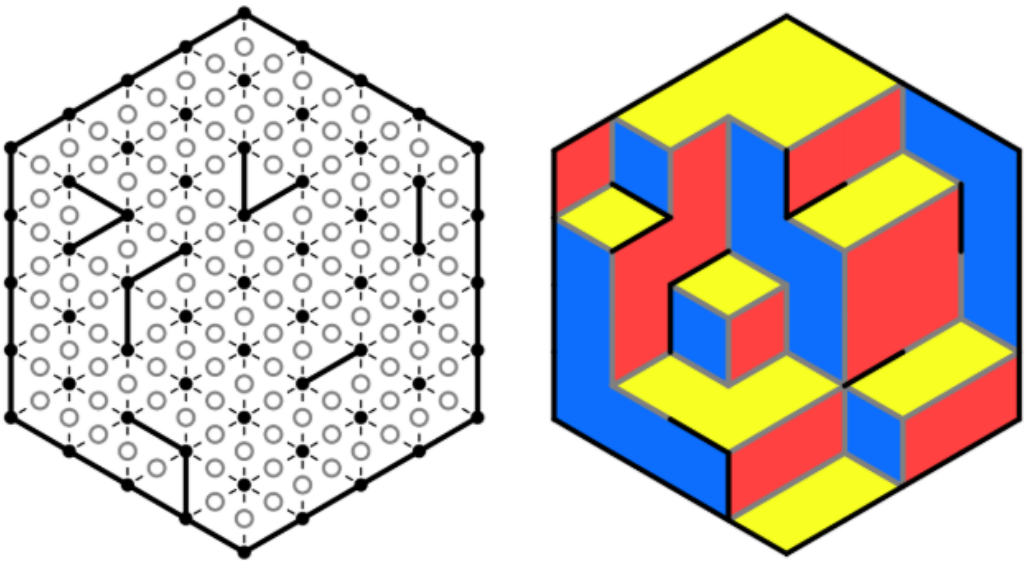


Figure 7.1: An example randomly generated solvable puzzle (left) and its generated solution tiling (right).

7.6 Time Complexity and Performance

The main operations of the algorithm are DAG construction and graph traversal. Let n denote the size of the grid. The number of cubes in the 3D region is n^3, and therefore the number of nodes in the DAG is n^3. Each node has at most three outward edges, giving a total of n^3 directed edges. Computing reachability using breadth-first search runs in $\mathcal{O}(V + E)$, and as both the number of nodes and edges are n^3, this results in $\mathcal{O}(n^3)$ time.

As both DAG construction and graph traversal cost $\mathcal{O}(n^3)$, this gives the overall algorithm a cubic complexity.

However, when generating solvable puzzles via repeated random generation, the total runtime becomes $\mathcal{O}(k \cdot n^3)$, where k is the number of attempts required to obtain a solvable instance. In practice, solvable puzzles are generated instantaneously, and this algorithm results in good performance.

8

Game Design and Implementation

This chapter outlines the design and implementation of the calissons puzzle game, as well as providing justification for design choices.

8.1 Difficulties

The difficulty system took inspiration from Pips [5]. The user can choose from three difficulties: easy, medium, and hard, accommodating users of varying skill levels.

Two factors control difficulty: grid size and number of edges. Larger grids increase the number of tiles required, raising the chance of errors. Edge count is more nuanced, too few edges produce trivial puzzles with many valid solutions, whilst too many essentially reveal the solution. The chosen parameters, shown in Listing 12, were determined through personal testing and could be refined with further user data.

```
useEffect(() => {
        switch (difficulty) {
                case "EASY": setGridSize(2); setNumberOfEdges(4); break;
                case "MEDIUM": setGridSize(3); setNumberOfEdges(8); break;
                case "HARD": setGridSize(4); setNumberOfEdges(10); break;
        }

...
}, [difficulty]);
```

Listing 12: Switch case inside a *useEffect* hook that sets the grid size and edge states upon a change to the difficulty state.

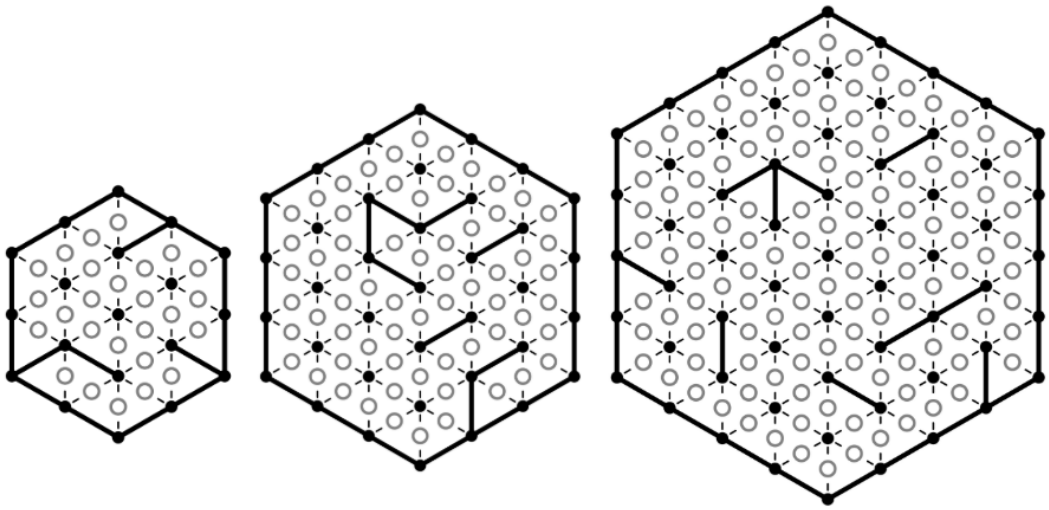


Figure 8.1: Example puzzles for each difficulty, from easy (left) to hard (right).

8.2 Implementing Win Conditions

Win conditions are checked after each tile placement (see Listing 3). The puzzle is solved when the tiles array reaches the maximum capacity of $(6n^2)/2$, and every edge is adjacent to tiles of different directions. If both conditions are satisfied, *isSolved* state is set to true.

8.3 User Metrics

Time was chosen as the primary user metric, calculated by subtracting the puzzle generation time from the time of completion. An incremental timer visible during play would be a natural improvement. In the endless mode, a streak counter tracks consecutive solves, resetting if the user skips a puzzle or generates a solution.

Figure 8.2: Time and streak counter from the endless mode.

8.4 Daily Mode

The daily game mode presents an easy, medium, and hard puzzle every day, with the puzzles being exactly the same for all users on the same date. This game mode took inspiration from Wordle [4] and Pips [5]. Two new systems were implemented to construct this game mode: seeded generation and local storage.

8.4.1 Seeded Generation

To guarantee the same puzzle is always generated for a specific date and difficulty, we apply seeded generation over the existing solvable puzzle generation system. Seeded generation replaces non-deterministic randomness with a pseudo-random number generator (PRNG). Given the same seed, a PRNG produces an identical sequence of values, ensuring that all random decisions made during puzzle generation are reproducible.

The pseudo-random number generator *mulberry32* [16] was used for this application. A 32-bit integer seed is generated by applying a simple hash function to a string of date and difficulty. This ensures all users playing on the same date and difficulty are returned the same puzzle.

```ts
export function getDailySeed(difficulty: Difficulties): number {
    const today = new Date();
    const str = `${today.getFullYear()}-${today.getMonth()}-${today.getDate()}-${difficulty}`;

    let hash = 0;
    for (let i = 0; i < str.length; i++) {
        hash = (hash << 5) - hash + str.charCodeAt(i);
        hash |= 0;
    }

    return hash;
}
```

Listing 13: Hash function used to convert the date and difficulty into a 32-bit integer seed.

8.4.2 Local Storage

The daily game mode saves progress and completion state in local storage. This proved tricky to implement as objects cannot be directly converted to a JSON file. Tiles are stored as string IDs and converted back to calisson tile objects upon loading.

```ts
export type DifficultyState = {
    tileIDs: string[];
    isSolved: boolean;
    autoSolved: boolean;
    startTime: number | null;
    elapsedTime: number;
};
```

Listing 14: *DifficultyState* type used to retain user progress in local storage.

A record of *DifficultyState* objects (see Listing 14) for each difficulty saves user progress in local storage and is loaded when opening the web application.

Local storage is also utilized to check if the user has used the application before, and if not, display the tutorial and about page (see section 8.6).

8.5 Endless Mode

The endless game mode provides users the ability to solve as many puzzles as they would like, and practice outside the daily mode. Implementation was simpler than the daily mode as no new systems needed implementation. Puzzles are generated for the selected difficulty, and the user can generate a new puzzle upon completion or by skipping. The streak counter keeps track of consecutive solved puzzles (see section 8.3).

8.6 Tutorial and About Pages

A tutorial page was implemented, taking design inspiration from Wordle [4]. The tutorial gives a brief overview of the grid and tiles to familiarize users with the puzzle space. The win conditions of the puzzle are displayed to the user alongside a visual representation of the edge adjacency condition. The controls of the puzzle, namely placing and removing tiles, are presented alongside GIFs to visually aid the user. The about page informs users about the project. See Appendix D for the full pages.

9

Front-end Systems and UI

This chapter is dedicated to the systems used in the front-end of the application, as well as the design of the user interface (UI).

9.1 Frameworks

React Bootstrap [17] was used for pre-built UI components and styling, and React Router [18] for client-side routing between the daily and endless game modes.

9.2 UI Design

9.2.1 Grid Layout

The UI of the application was built using Bootstrap's grid system. This means that the UI is split into rows and columns, which allow for the UI to scale accordingly for different devices.

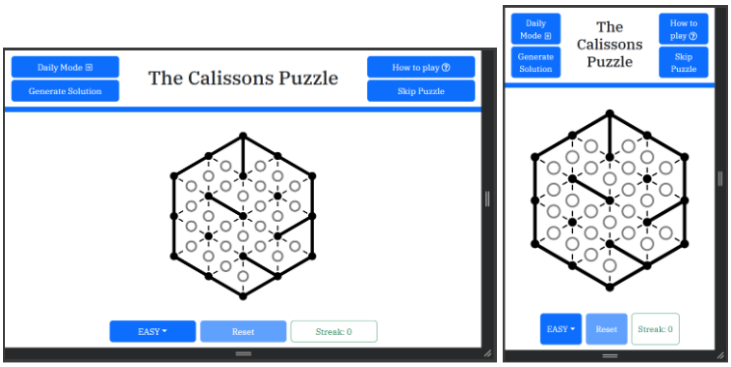


Figure 9.1: The grid system scaling the application for different resolutions.

The UI is split into three rows: a header row containing mode-dependent buttons such as the solving algorithm button, a middle row containing the puzzle's SVG, and a bottom row containing interaction buttons such as reset.

9.2.2 Modals

The React Bootstrap modal component was used for the tutorial and about pages (see section 8.6), as well as the puzzle results screen and confirmation pop-ups. A modal is a user interface element that appears as an overlay on top of the main content, temporarily disabling interaction with the background until the modal is dismissed.

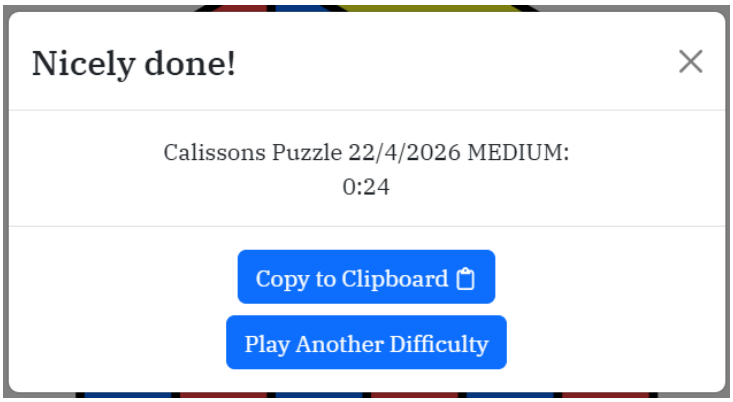


Figure 9.2: An example results modal from the daily game mode.

Confirmation modals are used for features like skipping puzzles and generating solutions, they add an extra step between pressing the button and the resulting action, this helps avoid accidental triggers.

9.3 Miscellaneous Features

9.3.1 Solving Algorithm Animation

To better convey the algorithm's steps, a delay was implemented so that solution tiles are added one-by-one starting from the outward edges of the *Back* set (see subsection 7.5.2). User interaction is disabled during the animation to prevent interference.

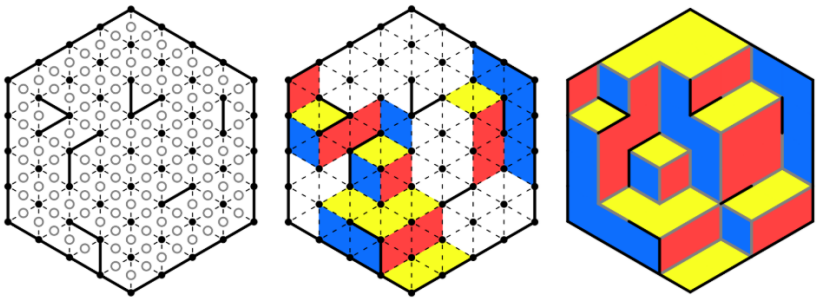


Figure 9.3: The solving algorithm mid-animation.

9.3.2 Viewing Original Edges

The puzzle, when solved, is displayed as the solution tiling with an edge between each change in direction. A feature was implemented to toggle between the complete set of edges, and the starting edges of the original puzzle.

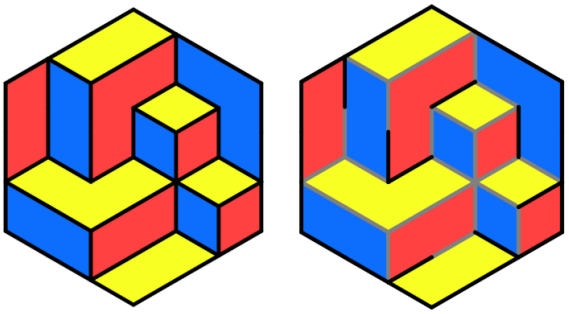


Figure 9.4: Toggle edges off (left) and on (right).

9.3.3 Copying to Clipboard

A “Copy to Clipboard” button was implemented for the results modal of the daily mode. This button allows users to easily share their results among other users, without having to manually copy their time (see Fig. 9.2).

9.3.4 Solving Confirmation

Upon manual puzzle completion, a confetti effect is utilized to confirm to the user that they successfully solved the puzzle. The puzzle will also switch to its solved display state, with a toggle option to view the original puzzle (9.3.2).

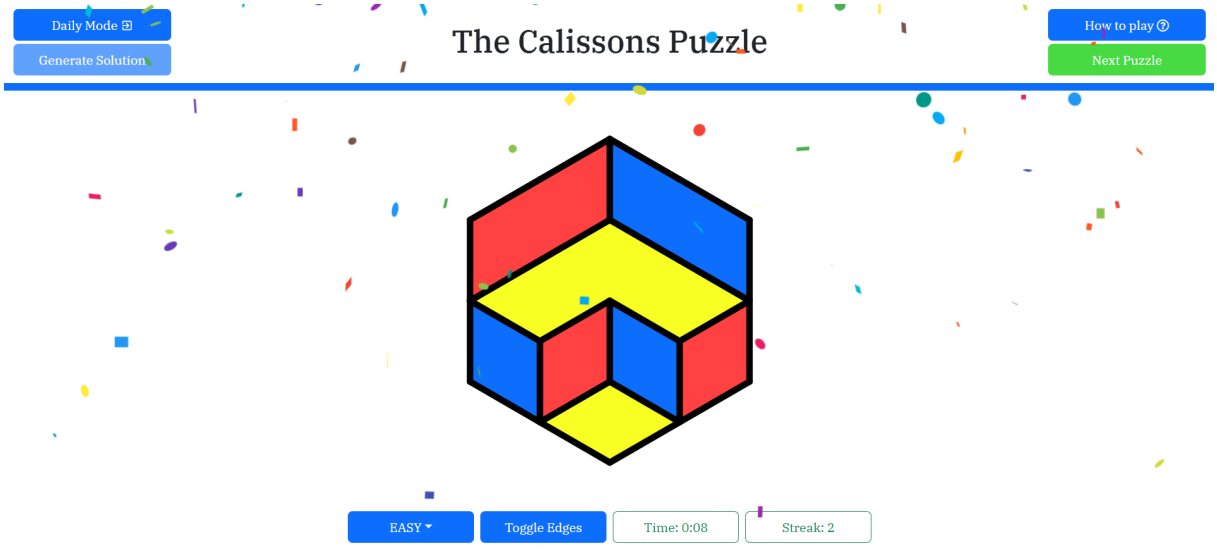


Figure 9.5: Confetti effect displayed after the user successfully solves a puzzle.

10

User-testing

See Appendix A for the professional considerations and ethics of this project's user-testing.

Overall, six participants were used for user-testing. All participants had no prior experience with the Calissons Puzzle.

As user-testing was the final phase of development, the application was treated as a finished product and feedback used only for evaluation.

10.1 Method

After reading the introduction script (section A.4) and receiving verbal consent, participants solved six medium difficulty puzzles followed by a brief questionnaire. For the first three puzzles, participants were told the rules verbally: "fill the grid with tiles such that each edge is adjacent to tiles of different directions/colours", with no access to the in-app tutorial. After the first three puzzles, participants read the tutorial before attempting the remaining three. Participants were encouraged to skip if stuck.

10.2 Data

10.2.1 Puzzle Solving

Participant	**Before Tutorial**			**After Tutorial**		
	P1	P2	P3	P4	P5	P6
1	SKIPPED	SKIPPED	SKIPPED	SKIPPED	SKIPPED	00:54
2	SKIPPED	02:24	03:08	00:44	00:40	05:02
3	SKIPPED	SKIPPED	SKIPPED	SKIPPED	03:08	SKIPPED
4	SKIPPED	SKIPPED	01:36	04:04	01:15	02:27
5	03:08	SKIPPED	SKIPPED	SKIPPED	00:53	00:37
6	SKIPPED	SKIPPED	SKIPPED	SKIPPED	02:42	03:09

Table 10.1: Table of puzzle solving times from the six participants.

A total of 14 puzzles were skipped before the tutorial, compared to 6 puzzles after the tutorial.

For solved puzzles, the average completion time before the tutorial was 02:34, compared to 02:07 after the tutorial.

10.2.2 Questionnaire

The following questions were asked at the end of the experiment. Responses were recorded on a scale from 1 to 5, where 1 represents the lowest rating and 5 represents the highest.

1. Did the user interface feel intuitive to use?

2. How confident did you feel in solving the puzzle before the tutorial?
3. How confident did you feel in solving the puzzle after the tutorial?
4. Did the tutorial help you understand the rules of the puzzle?
5. Did the tutorial help you understand how to interact with the puzzle?
6. Did you understand how to undo your actions?
7. Did the puzzle feel satisfying to solve?
8. Did the solving algorithm feel satisfying to watch?

Participant	**Questions**							
	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
1	4/5	2/5	3/5	3/5	1/5	5/5	5/5	5/5
2	3/5	2/5	3/5	4/5	3/5	4/5	4/5	4/5
3	2/5	1/5	3/5	4/5	5/5	5/5	5/5	5/5
4	4/5	2/5	4/5	5/5	5/5	5/5	5/5	5/5
5	4/5	3/5	3/5	2/5	2/5	5/5	4/5	5/5
6	3/5	1/5	3/5	4/5	1/5	3/5	5/5	4/5
Avg	**3.33**	**1.83**	**3.17**	**3.5**	**2.83**	**4.5**	**4.67**	**4.83**

Table 10.2: Table of question answers from the six participants on a scale from 1 to 5, with average scores at the bottom.

11

Evaluation

This chapter evaluates the project's objectives against the requirements defined in chapter 2, as well as evaluating extension objectives, areas of improvement, and future work. Evaluation is based on user testing results, performance metrics, and implemented functionalities.

11.1 Primary Objectives

Develop an interactive web-based visualization of the Calissons Puzzle

This objective was fully achieved. The web application successfully renders an interactive visualization of the Calissons Puzzle, with interaction being aided by a tutorial and clear visual feedback.

Interface

During user testing, observed participants faced no difficulties interacting with the puzzle interface, and rated the interface's intuitiveness an average of 3.33/5 (mode 4/5). This evidences that the system provided an intuitive interface for puzzle interaction.

Users can place and remove tiles, with clear visual indication for these actions. Participants rated their understanding on how to undo their actions a 4.5/5 on average.

TypeScript typing and event handlers ensures that the system appropriately handles invalid inputs, and no runtime errors were observed during testing.

Guidance

One of the main objectives of the user testing experiment was to evaluate the impact of the tutorial (8.6) on the user's understanding and performance.

Overall, participants skipped 8 fewer puzzles after interacting with the application's tutorial, and average completion times improved by 27 seconds. Participants rated, on average, a boost in puzzle solving confidence from 1.83/5 before the tutorial to 3.17/5 after the tutorial. This data suggests that the in-game guidance system provides users with a clearer understanding on how to interact with, and solve the Calissons Puzzle. Improvements may also be influenced by a learning effect, as participants became more familiar with the puzzle over time.

Implement procedurally generated puzzles

This objective was fully achieved. The application can automatically generate random solvable puzzles theoretically of any size, but grid sizes are capped at 4 to ensure compatibility with smaller devices, as well as maintaining good performance.

Solvability

Solvability of generated puzzles is ensured during the generation process. Puzzles will be generated until one passes the solvability check. See subsection 6.4.2 for more information.

Performance

Brief performance tests from a laptop (see section C.2) show that generating puzzles of any difficulty rarely exceed 1ms. The average time to generate a solvable hard puzzle (grid size of 4) was 2ms. This is effectively instantaneous for typical modern devices, ensuring a responsive user experience.

Visualize an automatic solving algorithm for any given puzzle instance

This objective was fully achieved. Solutions can be automatically generated by the application for any puzzle instance (see chapter 7). Solution tiles are pushed to the grid with a slight delay to create an animation for user experience.

User Experience

Participants of user testing got to watch the solving algorithm on any puzzles they skipped, as well as on an example of a hard puzzle. Participants rated the satisfaction of watching the solving algorithm a 4.83/5.

Performance

Performance tests on generating solutions for hard puzzles (grid size of 4) (see section C.2) give an average time of 0.75ms.

Perform a round of user testing to evaluate success criteria

This objective was achieved but could've been improved upon. A round of user testing was performed on six participants with no prior knowledge/experience with the puzzle. For ways the user testing could've been improved see section 11.3.

11.2 Extension Objectives

Automatically assign a difficulty rating to any generated puzzle

This objective was outlined at the beginning of the project, and was therefore partially reinterpreted during development. Puzzles are generated based on the selected difficulty (see Listing 12).

Implement a leaderboard for competitive player statistics (e.g. number of moves, time taken)

This objective was not met. A leaderboard system would require a database and server, which did not fit the scope of this project.

Ensure compatibility with most modern devices

This objective has been met. The web application can be accessed via most modern browsers, and the bootstrap grid system (9.1) ensures proper scaling for all reasonable screen sizes.

Puzzle save states and reset option

This objective has been met. All puzzles can be reset once the user has placed at least one tile, and the puzzle's tiles are saved in progress via local storage (only for the daily game mode).

A daily puzzle mode akin to Wordle

This objective has been met. The daily game mode is the default game mode of the application, and was implemented with features such as: save states, difficulties, results, and seeded generation.

11.3 Areas for Improvement

11.3.1 User Testing

User testing was performed on six participants, this provided good data to evaluate the project's criteria, but the number of participants was not enough to make meaningful statistical claims about the system.

Although this project targets a broad audience, all participants were young adults and didn't represent any other demographics. This means that I cannot evaluate if this project is accessible to broader demographics[1].

11.3.2 Difficulty System

The implemented difficulty system was created purely off personal testing. This system might not reflect the actual difficulty users experience with the puzzles and could've been based off user testing or via an automated difficulty detection system.

11.4 Development Methodology and Timeline

Originally, this project followed an Agile methodology, with development done in sprints (see B.3 for example sprint documentation). This shifted in development towards a more self-centred development approach, ditching the documentation as it went unused.

The project timeline partially followed the original Gantt chart (see section B.2), with at least 3 extra weeks spent researching the solving algorithm.

11.5 Development Tools

The chosen tools proved well-suited to the project. Whilst TypeScript's strict typing occasionally posed syntactic challenges, catching errors at compile-time was invaluable given the complexity of the coordinate systems involved. React's component-based architecture and hooks managed puzzle state and UI elements cleanly throughout development. In hindsight, I would choose the same tools for a project of this nature.

[1] Despite not being official testing, friends and family have used the application, and my grandma is a big fan.

12

Conclusion

This project successfully delivered an interactive web application for generating and solving calissons puzzles, meeting all four primary objectives and the majority of extension objectives. The application is accessible via most modern devices and provides an intuitive interface for a puzzle that previously had little dedicated digital implementation beyond Longuet's blog [3].

The central technical contribution of this project is the implementation of the advancing surface algorithm [1]. DAG construction and traversal both run in $\mathcal{O}(n^3)$ time, with hard puzzles generating and solving in under 3ms in testing. The algorithm's visualization was particularly well received by user testing participants.

User testing demonstrated that the application successfully conveys the puzzle to new users. However, the small sample size of six participants, all young adults, limits the practicality of these findings, especially considering the intended broad demographic of the project.

Future work could address these limitations in several ways. A larger and more diverse user testing phase would provide more meaningful data for refining the guidance and difficulty systems. The solving algorithm visualization could be extended into a step-by-step interactive explanation of the advancing surface algorithm, which was originally intended but deemed out of scope. A leaderboard system, requiring a backend database, would add a competitive dimension to the daily mode. Finally, the duplicate detection in graph construction could be optimized as discussed in section 5.3, improving scalability for larger grid sizes.

Bibliography

[1] Jean-Marie Favreau and Yan Gerard and Pascal Lafourcade and Léo Robert, "The Calissons Puzzle." https://doi.org/10.48550/arXiv.2307.02475, 2023.

[2] Wikipedia, "Calisson." https://en.wikipedia.org/wiki/Calisson, 2025. Accessed: 7th November 2025.

[3] Olivier Longuet, "Le Jeu des calisson." https://mathix.org/calisson/blog/index.php?static2/regle, 2023. Accessed: 7th November 2025.

[4] Josh Wardle, "Wordle." https://www.nytimes.com/games/wordle/index.html, 2022. Accessed: 9th November 2025.

[5] New York Times, "Pips." https://www.nytimes.com/games/pips, 2025. Accessed: 9th November 2025.

[6] Gabriele Cirulli, "2048." https://play2048.co/, 2014. Accessed: 9th November 2025.

[7] Kori M. Inkpen, "Drag-and-drop versus point-and-click mouse interaction styles for children." https://doi.org/10.1145/371127.371146, 2001.

[8] Geeks for Geeks, "React Introduction." https://www.geeksforgeeks.org/reactjs/reactjs-introduction/, 2025. Accessed: 10th November 2025.

[9] Microsoft, "TypeScript Documentation." https://www.typescriptlang.org/, 2025. Accessed: 10th November 2025.

[10] How Dev, "What is Vite.js?." https://how.dev/answers/what-is-vitejs, 2025. Accessed: 10th November 2025.

[11] Tobias Koppers, "Webpack." https://webpack.js.org/, 2014. Accessed: 12th November 2025.

[12] Netlify, "Netlify Documentation." https://docs.netlify.com/, 2026. Accessed: 16th April 2026.

[13] Red Blob Games, "Hexagonal Grids." https://www.redblobgames.com/grids/hexagons/, 2026. Accessed: 17th April 2026.

[14] V. van Oostrom, "The problem of the calissons, by rewriting," in *IWC 2024*, p. 24, 2024. https://www.trs.css.i.nagoya-u.ac.jp/event/iwc2024/IWC2024_proceedings.pdf.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 3 ed., 2009.

[16] Cade Rosche, "Mulberry32 in JavaScript." https://github.com/cprosche/mulberry32. Accessed: 22nd April 2026.

[17] React Bootstrap Team, "React Bootstrap." https://github.com/react-bootstrap/react-bootstrap. Accessed: 22nd April 2026.

[18] React Router Team, "React Router." https://github.com/remix-run/react-router. Accessed: 22nd April 2026.

Appendix C

Testing Results

C.1 User Testing

		Before Tutorial	(min:sec)		After Tutorial	(min:sec)				Questions	(1 to 5)			
Participant	Puzzle 1	Puzzle 2	Puzzle 3	Puzzle 1	Puzzle 2	Puzzle 3	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
1	FAILED (02:12)	FAILED (02:06)	FAILED (01:57)	FAILED (04:17)	FAILED (03:47)	00:54	4	2	3	3	1	5	5	5
2	FAILED (07:23)	02:24	03:08	00:40	00:40	05:02	3	2	3	4	3	4	4	4
3	FAILED (04:30)	FAILED (03:32)	FAILED (05:36)	FAILED (04:57)	03:08	FAILED (06:35)	2	1	3	4	5	5	5	5
4	FAILED (10:41)	FAILED (06:11)	01:36	04:04	01:15	02:27	4	2	4	4	5	5	5	5
5	03:08	FAILED (06:06)	FAILED (04:05)	FAILED (06:07)	00:53	00:37	4	3	3	2	2	5	4	5
6	FAILED (03:27)	FAILED (03:36)	FAILED (03:30)	FAILED (05:24)	02:42	03:09	3	1	3	4	1	3	5	5
		Before Tutorial			After Tutorial		3.333333333	1.833333333	3.166666667	3.5	2.833333333	4.5	4.666666667	4.833333333
	Total Failed:	14		Total Failed:	6									
	Average Time:	02:34		Average Time:	02:07									

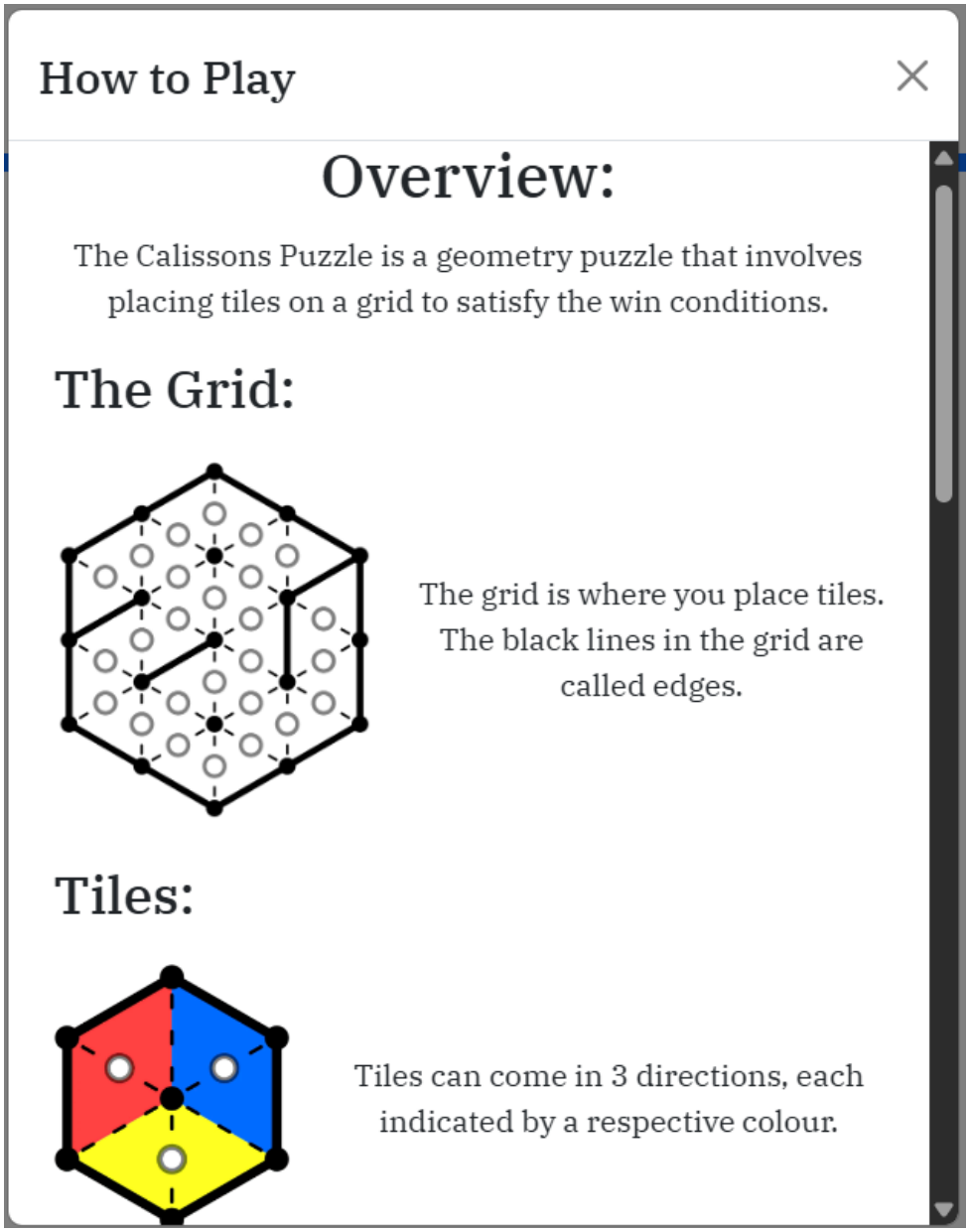
C.2 Performance Test

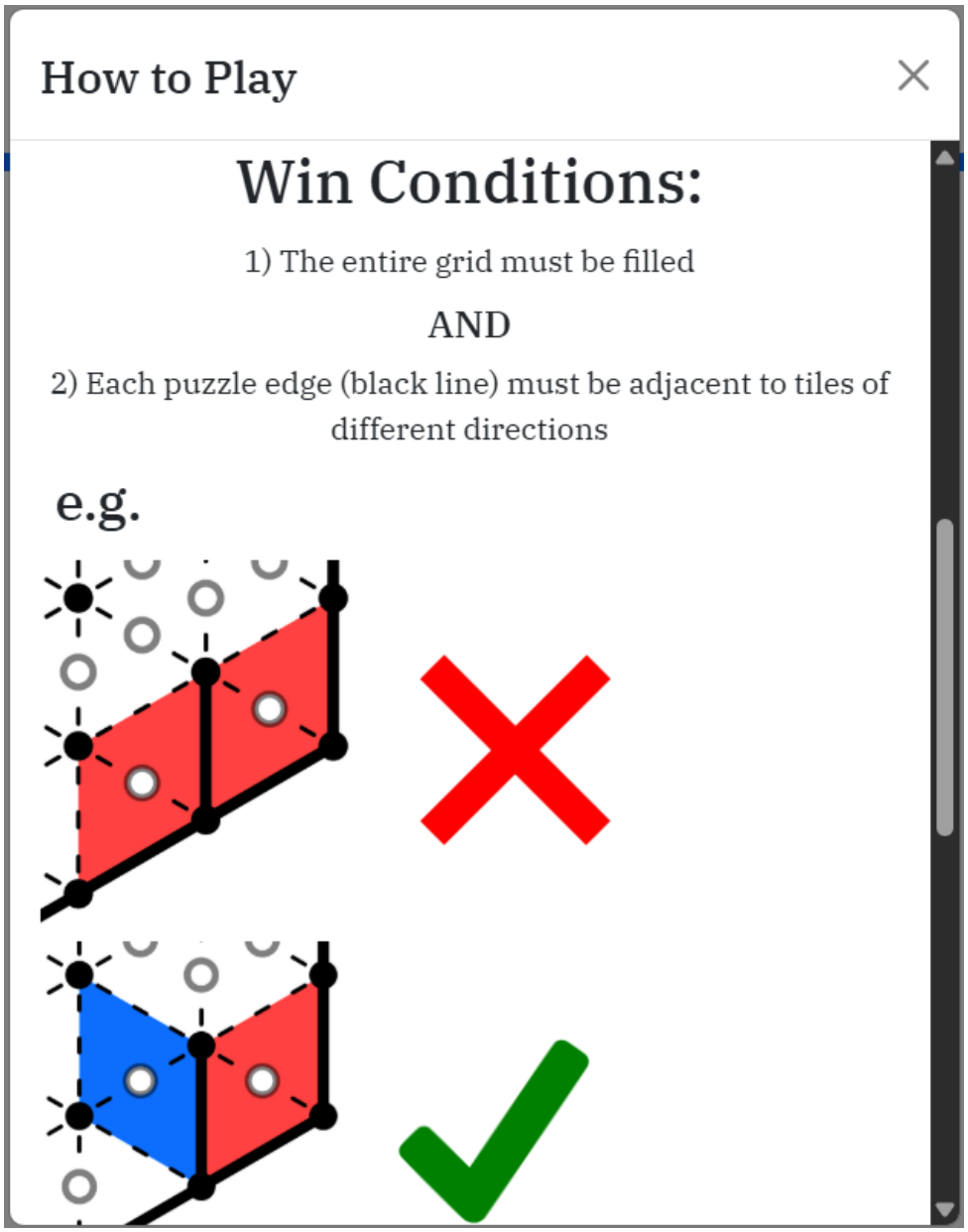
Grid Size		Generating Solvable Edges		Times (ms)					Average
2	1.66	0.99	0.22	0.13	0.3	0.56	0.75	0.19	0.6
3	2.63	1.18	2.87	2.03	0.36	2.27	0.53	1.17	1.63
4	2.76	1.16	1.08	0.38	0.36	4.36	5.92	0.69	2.08875
Grid Size		Generating Solutions		Times (ms)					Average
4	1.18	0.74	0.6	0.6	0.86	0.59	0.76	0.72	0.75625

Appendix D

Game Assets

D.1 Tutorial

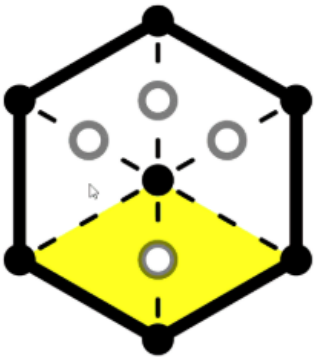



How to Play
Win Conditions:
1) The entire grid must be filled
AND
2) Each puzzle edge (black line) must be adjacent to tiles of different directions
e.g.

How to Play

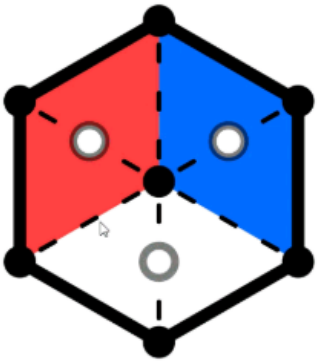
How to place tiles:

1. Each node correlates to one tile of a specific direction
2. Hover a node to see what direction (colour) of tile it represents
3. Click on a node to place the corresponding tile



How to remove tiles:

1. Each tile will always have a node in its centre
2. If a tile is present, click on its node to remove it

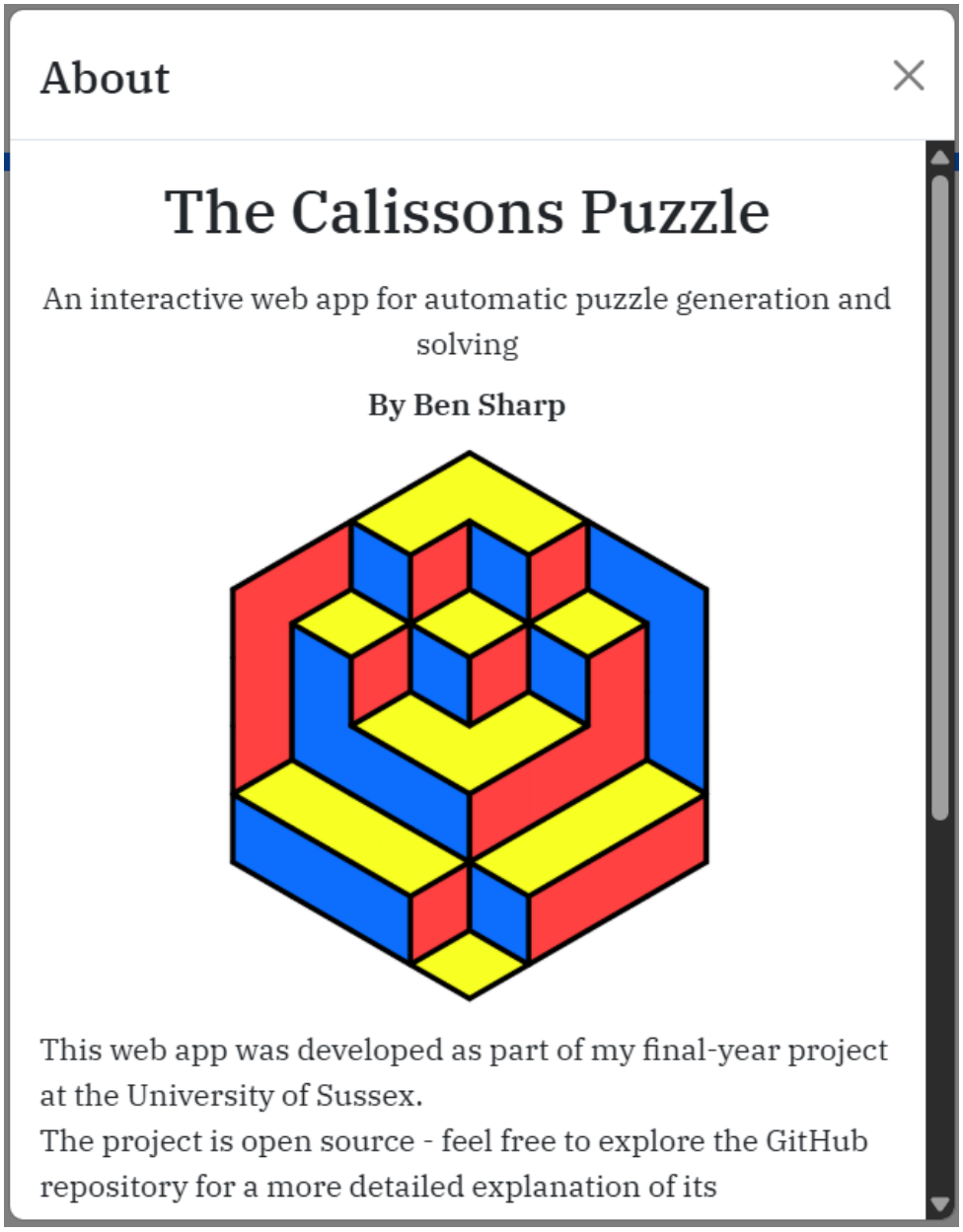


Good Luck and Have Fun!

[1]

[1] GIFs are used to visualize tile placement.

D.2 About


About
The Calissons Puzzle
An interactive web app for automatic puzzle generation and solving
By Ben Sharp
This web app was developed as part of my final-year project at the University of Sussex.
The project is open source - feel free to explore the GitHub repository for a more detailed explanation of its

About

This web app was developed as part of my final-year project at the University of Sussex.
The project is open source - feel free to explore the GitHub repository for a more detailed explanation of its implementation.
(GitHub repo will be made public after submission)

About the Puzzle

The Calissons Puzzle (le jeu des calissons) was created in 2022 by Olivier Longuet. All credit for the puzzle's design goes to Olivier. Please check out his blog here.

How Does It Work?

This app implements the *advancing surface algorithm* outlined in this paper. All credit for the algorithm goes to its authors. Feel free to check out this project's GitHub repository for a deep dive.