Abstract

Visualising the process of running an algorithm, and its effect on the underlying input data structure, can be an awkward and clumsy process when not planned out in advance. A student's higher-level understanding of such algorithms can therefore suffer if a teacher's cannot effectively demonstrate the lower level operations on the data structure, especially in the context of a lesson or lecture, where an ad-hoc modification to a demo may be required to answer a question or demonstrate a concept.

This project analyses whether an integrated web-based platform for algorithm demonstration, using a JavaScript-based code editor with access to a built-in visualisation system for common data structures, can effectively streamline the process of presenting an algorithm and make it easier to both teach and learn algorithms.

Contents

Chapter 1

Introduction

Algorithms are a fundamental concept in Computer Science. They encapsulate every process for computational problem solving in the field, from lower-level manipulation of fundamental data structures such as the sorting of lists, to higher-level concepts which build upon these such as evolutionary algorithms, neural networks and cryptographic schemes. The description of an algorithm covers the type (and structure) of the input and output, as well as the operations performed on the input data in order to transform it into the output.

The process of teaching an algorithm in a lecture or lesson might typically include a visualisation of the data structure being manipulated. Slides are a common delivery format for presenting lectures, which allow ordered visual depictions of data structures in order to present the flow of an algorithm. They have the benefit of being portable and deliverable in most physical locations where a lesson would take place: a lecture theatre, classroom or a laboratory. They are also accessible on most personal devices without much setup; assuming the device offers functionality to view PDFs, which is a common feature of even the most basic modern smartphone, the slides can be downloaded and viewed with no additional setup, either at home or in the lesson as it takes place.

However, slides have the drawback of needing to be prepared ahead of time, and creating them can be a slow and time-consuming process. They are not an interactive medium; if you wish to amend the demonstration—perhaps to use different input data, or to demonstrate a variation of the algorithm or its parameters—you must run the algorithm manually (or by hand), and then transform the performed steps into the corresponding visual representation used in the slides. This process is generally manual and prone to user error, and is certainly not conducive to improvised modifications during a lesson, such as in response to a question posed by a student.

An alternative to this would be to make an implementation of the algorithm available to students, to run on their own device. This could be demonstrated live via a projector as part of the lecture, or provided for students to run on their own device—or perhaps some combination of the two. This would permit live changes to the demonstration in response to questions asked, and also permit students to experiment with the code to further their understanding of the implementation.

This, however, is burdened with numerous drawbacks. Such a setup depends on the demonstrator or students having access to devices capable of running the code in the lesson. It may not be practical to ensure everyone has the correct software ahead of time, such as the language runtime, and an IDE or suitable debugger. Any non-trivial setup required to get the code up and running wastes precious time in the lesson which could be used for teaching and discourse. A student who struggles to acquire and run the software may lose track of the lesson and fall behind.

Also, this method is not device-agnostic. A student who brings a portable tablet device, or one running a different operating system to the one for which the example code is written, may not be able to run the code at all. The student may then waste further time trying to find an alternate resource to use. The approach isn't very collaborative, and only works if everyone can run the same software and is already comfortable using the required tools or IDEs.

Finally, this misses the visual aspect that slides serve to provide. A debugger may be useful to step through an algorithm if the runtime provides this feature. However, the data structure being used may not be conceptually easy to visualise by looking at the code or debugger interface. For less elementary algorithms, a lack of an easy-to-digest visualisation may obscure and hinder a student's ability to understand both what the algorithm does to the data structure at a low level, and also why the algorithm works to solve the higher level problem at hand.

Specialised visualisation tools do exist for demonstrating algorithms. These may be presented as web apps, or as downloadable tools. These combine the benefits of both aforementioned delivery formats and may remove many of the drawbacks. However, there may not always be such a tool for a given algorithm. If there is, it may be specific to a single variant of an algorithm, or be inflexible in how the input data can be changed. It may not even be implemented correctly! If you wished to compare several sorting algorithms, for example, you may need to use several tools created by different authors, each with their own UI and style of visualisation. Switching between these to compare algorithms may not be seamless; if this is the case, some mental effort might be required by both the demonstrators and students in order to re-adjust to how each tool works, which may make it difficult to incorporate these effectively into a demonstration. Working with an unfamiliar teaching tool may be frustrating and break the flow of a presentation.

What if we could create a generic platform for demonstrating algorithms, specifically designed to be easy to use and adapt as required during a lesson or lecture? Could we combine the widespread availability and visual aspect of a slide-based algorithm demonstration with the interactivity of a running implementation, while also making it generic enough to allow any algorithm to be demonstrated using a particular data structure? Additionally, how would such a tool need to be designed so that it is effective as a teaching tool? This project analyses which criteria such a tool would need to meet, which technology and modes of interaction can be used to meet these requirements, and then implements a tool which attempts to meet these goals in order to create an integrated demonstration environment which streamlines the process of teaching and understanding algorithms.

1.1 Literature Review and Goals

The previous section identified the visual aspect of a slide-based demonstration. Specifically, slides permit a visual representation of the input data structure as it is manipulated to reach the output. You may traverse through the slides linearly, forwards or backwards. This kind of visual representation has already been identified as something which helps computer science students to understand algorithms more easily and in greater depth; specifically, greater student engagement with interactive *algorithm visualisations* (henceforth referred to as **AV**s) is correlated with a greater conceptual understanding of the algorithm (Grissom et al., 2003). This suggests that a tool which provides a greater level of interactivity with the visualisation will help us fulfil our goal more effectively. Prior research has described a hierarchy of different levels of engagement with a visualisation (Naps et al., 2002), which Grissom et al. also refer to. We should consider this hierarchy when deciding what AV functionality to offer. The levels of the hierarchy are:

1. **Viewing**. This involves simply watching a visualisation. The only control available at this level is the ability to move forwards and backwards. This is equivalent to a slide-based demonstration, as discussed earlier.

2. **Responding**. The research specifies this as answering questions posed about the demo, either integrated into the tool, or presented alongside it as a pen-and-paper exercise. In our case, in the context of a lesson or lecture, this might also include questions asked in real time by the demonstrator/teacher. this level is therefore equivalent to a slide-based demo used as a teaching aid during a lesson, in which the teacher poses questions to the students.
3. **Changing**. This involves altering what is visualised in some way. Grissom et al. suggest this includes changing the input data to explore how the algorithm behaves in different situations or edge cases. I would propose this also includes exploring variations of algorithms. For example, if quicksort was demonstrated, how would changing the choice of pivot alter the effectiveness and efficiency of the algorithm? The research identifies this level of engagement as being significantly better than simple passive observation of a visualisation.
4. **Constructing**. This involves taking a new algorithm and imploring the student to implement it and visualise it using the framework provided by the AV tool. Grissom et al. cite several other prior pieces of research, with conflicting results, as to whether this is more effective than the *Changing* engagement level from a student's perspective of learning an algorithm. However, the ability to implement an arbitrary algorithm for a given data structure would be important from the teacher's perspective, in the context of creating a generic AV platform—even if the students weren't doing this themselves—so I would consider exposing this functionality as still being an important consideration in the design requirements.
5. **Presenting**. This involves the students themselves using the AV tool to demonstrate an algorithm to their peers, which may or may not have been implemented themselves. The researchers indicated they hadn't identified any empirical studies into the effectiveness of this level of engagement. Given this project's intended goal of creating a teaching assistant tool, it may be possible to perform some initial research at this level if the tool is capable of being used in such a way.

Slide notes, as a supplement to an interactively-taught lesson, already provide basic AV functionality up to the *Responding* level. Therefore, in order to improve on this, the tool at a minimum should allow some way of changing the visualisation on the fly. Demonstrating an algorithm running in code, as part of a lecture, was also discussed in the introduction. This itself provides certain benefits. Research has explored the effectiveness of using a live code environment which is used to manipulate an AV, and found that minimising the time lag between the editing the code, and seeing an updated visualisation, resulted in students being stuck for less time, thus requiring less teacher intervention—and, ultimately, implementing algorithms that were correct more frequently compared to environments where code had to be written, compiled and then ran as part of a longer 'feedback cycle' (Hundhausen and Brown, 2007).

This research calls this a *what-you-see-is-what-you-code* (WYSIWYC) approach, and suggests that by blurring the distinction between implementing and visualising an algorithm, you can easily meet the criteria for the *Constructing* level of interactivity—and that the immediate edit-time feedback prevents so-called *gulfs of evaluation*, in which a disconnect between written code and the effect of its execution impedes understanding of what the code is actually doing. This is an important takeaway, in that the teaching assistant tool should facilitate changing the visualisation by allowing direct modification of the algorithm's implementation and therefore seeing the results immediately.

This describes effectiveness specifically as a visualisation tool. Research has shown that an AV tool on its own does not necessarily act as effective pedagogical support (Rößling and Naps, 2002). This literature has identified a set of requirements in order to tie AV functionality into a teaching context.

- **Rewind capability**. This feature is identified as being critically important to making

AVs effective as a teaching tool, as a student who gets lost or confused can backtrack step-by-step—restarting the animation isn't sufficient (Anderson and Naps, 2001). This is one area where a slide-based demonstration may be better than a live code demo, as debuggers may not necessarily offer functionality to arbitrarily rewind execution flow. This will have to be especially considered in the design process, as exposing control of the AV by executing code while also exposing rewind functionality may not be straightforward.

- **Reachability**. The platform should be accessible to the widest possible audience. Again, slides satisfy this criteria, as they can be viewed on almost any portable device. A downloadable program will take time to download and setup; the intoduction to this report describes why such interruptions in the lesson process are unsatisfactory. Java Applets (and other similar solutions such as ActiveX widgets) were a historical solution to this, but these are now plagued by security and permissions issues. At the time this research literature was written, these were the only available methods for interactive AVs. However, we now have portable devices capable of running full-featured web apps, and modern HTML5 and JavaScript–based applications can supplant the functionality of these two delivery methods while also being reachable by any web-capable device at an instant. As of February 2020, 85% of mobile and tablet users used WebKit, Blink or Firefox Quantum-based browsers with full HTML5 and JavaScript support (statcounter, 2020). Web apps do not require software to be downloaded, and require only the web page's resources to be fetched; provided an internet connection is available, this can be easily done on desktop, tablet and mobile.
- **General-purpose**. Rößling and Naps describe how systems exist which are optimised to visualise one specific topic but are rarely useful for anything else, meaning that users may have to adapt to the interface of several such systems within a single lesson or course, as mentioned in the introduction. The literature describes how integration of a multitude of animations in a common UI is vital to students becoming comfortable with using AV as a learning tool and makes it easier to integrate the tool into a computer science course.
- **Structural view of algorithm**. The research identifies the usefulness of being able to jump to key points or stages in an algorithm.
- **Interactive prediction**; that is, stopping the AV with pop-up questions prompting the student to predict what happens next, or the result of a subsequent step. Rößling and Naps describe how a confused student who is prompted with a question at each stage can benefit from such prompts, as they can 'reset' the student's perception of the algorithm with the correct answer, and an up-to-date mental context of the algorithm at the current state.
- **Annotations**. Alongside the animated AV and the source code of the algorithm, some annotations should be provided to explain what the AV is currently representing.
- **Smooth motion**. The research reports that a smooth animation of the algorithm's operation helps the majority of users detect the change between successive steps. If the animation is rewound, this smooth motion should apply to the reversed operation, too. There should also be pause functionality to allow the user to take breaks if desired.

This research therefore provides a useful set of requirements which make the AV more effective in a teaching context. Grissom et al. identify how the use of AV relates to the outcome of students' learning at different levels of the *Bloom taxonomy*; this is a hierarchy of increasingly sophisticated levels of conceptual understanding of a topic. In our case, this ranges from a surface-level understanding of the operations performed by an algorithm on the data structure at the lower levels, to a deeper understanding of why this combination of operations works to solve the problem at a higher level. The highest levels describe the ability to generalise this knowledge to (for example) develop new algorithms, or reason about the strengths and weaknesses of different algorithms. Grissom et al. find that AVs alone are effective at speeding up a student's

understanding at the lower levels of the Bloom hierarchy. While it finds they are not effective on their own at helping students to develop at higher levels of the Bloom scale, the research suggests that using the AV as a platform for a lesson means the teacher wastes less time covering the basics, such as visualisating the operations on the data structure. This frees up time for the teacher to discuss the more abstract concepts of the algorithm, and thus aid the students' conceptual understanding of an algorithm at higher levels of the Bloom scale.

This highlights that focus should be put on the intended model of use as a classroom assistant tool. As mentioned, new modes of interaction are made possible now that students have much greater access to powerful web-capable mobile devices than they did historically; indeed, many will now have their own smartphone. *Kahoot!* is an existing educational tool which capitalises on students' access to mobile devices in order to provide interactive quiz, survey and questionnaire functionality in a way that can be done on-the-fly in (or just before) a lesson without requiring much advanced preparation (Plump and LaRosa, 2017). *Kahoot!* unifies the quiz creation and student quiz completion interface into a single web app. When a teacher starts a quiz, it is given a unique short numeric code. Students can then instantly participate in the quiz on their own device by entering this code into the web app. The URL is short and memorable, and doing it this way means teachers don't have to worry about publishing a URL to students ahead-of-time, like a standard web survey would require, or wasting time getting students to enter a fiddly URL during a lesson. The web app also uses responsive design effectively so that it works well and looks the same on devices of all form factors.

Kahoot! also employs a small degree of gamification, whereby students can see which student is 'in the lead' in a given session of a quiz. Fun music and soft graphics are also used. The researchers find that these improve student engagement by providing a fast-paced learning platform with a degree of competition, which can be accessed during a lesson without frustration or interruption. The tool provides instant feedback to whether students are correct or incorrect; the researches note that this promotes class discussion of the correct answer, and helps to tie the discussion back to the lesson in which *Kahoot!* is used as an educational assistant. This instant feedback and interactive delivery mechanism are similar to the feedback and interactivity discussed by Hundhausen and Brown with their WYSIWYC model. Therefore I think this collaborative approach, in which the experience of students is synchronised across devices and ultimately to the lesson plan, may be employed in an algorithm demonstration tool to synchronise the demonstration of an algorithm across the entire classroom. The researchers report that this particular delivery model (unified web app) successfully achieves their goals. This additionally meets the Reachability criteria identified by Grissom et al., further suggesting it would be useful model to employ for an AV tool.

Chapter 2

Design

The requirements identified in the literature constrain the design of the visualisation tool; in particular, which technologies can be used. One such requirement, for the tool to be reachable to teachers and students without prior setup, constrains what the tool can be written in. The *Kahoot!* design study gives us a good model to work with. *Kahoot!* requires no additional browser plugins; it is presented as a *single-page application* wherein the UI is updated by dynamically rewriting the DOM using JavaScript, according to new information retrieved asynchronously from a server—that is, without refreshing the page. According to that study, this delivery model works effectively in the classroom, so I decided on using a combination of HTML/CSS and JavaScript to write the front end of the tool.

In the design phase, the key requirements are informally analysed, and broken down to determine which technologies and architectural patterns to use. By taking into account who the requirements are relevant to, they are then broken down into user stories, and prioritised to form a development plan.

2.1 Planning

There are four key functionality requirements identified in the literature:

- Ease of accessibility to teachers and students in a classroom or lecture situation, with minimal setup, as mentioned previously.
- Animated, interactive visualisations of data structures.
- Live and editable code demos, where the AV is representing the data being manipulated, and the animations represent the operations on the data structure by the algorithm.
- Group participation, perhaps with some degree of gamification.

The plan for the tool combines these requirements.

2.1.1 Accessibility

As mentioned, the first requirement constrains us to a HTML/CSS and JavaScript–based approach, as a web app is essentially the only setup-free delivery method available—especially in a lecture, where students will most likely be using their own personal devices, making it impractical to pre-load apps ahead of time. This must also take into account the form factors it must run on. While most mobile phones and tablet based web browsers are equipped with fully-fledged JavaScript engines, they do not have the processing power of a desktop. While JavaScript engines such as V8 (used in Chromium-based browsers, such as Chrome on desktop and mobile) are a lot

more efficient now in 2020 than they were some years ago, the implementation of this tool must still take into account the inherent limitations of mobile computing.

This rules out heavyweight JavaScript UI libraries such as React, as the extensive functionality offered by these frameworks may have an unacceptable performance impact on a tool whose effectiveness—per the analysed literature—partially depends on immediate and responsive UI feedback. At the time of writing, I also have little experience using these libraries, so for the sake of fast development I chose to instead write JavaScript that directly manipulates the DOM.

2.1.2 Animated AVs and Graphics

The second requirement for AV presentation in the UI requires graphics functionality. This is already offered in JavaScript, using `<canvas>` elements. This supports 2D vector graphics which will be ideal for visually representing data structures. Using this API directly would prove a challenge when it comes to animating operations on these data structures. *Two.js* is a library providing an API which wraps canvas graphics functionality and exposes it as a *scene graph* (two.js, 2020). Rather than having to deal with tracking the hierarchical relationships between graphical elements and drawing them directly, this will allow the tool to associate the data structures they represent with abstract representations of graphical objects on screen. Graphical elements are then constrained on a tree of parent-child relationships, in which a child element's position is computed with some offset from the parent element's position. The tool therefore only needs to deal with adding/removing and manipulating elements and their visual properties; the library draws the scene for us. This declarative approach is just as powerful and will save development time.

2.1.3 Editable Code

The third requirement means the tool should allow the user to create a demonstration of an algorithm by writing it within a code editor in the tool's interface. The tool should alternatively allow users to load the code of several built-in algorithms; both to speed up the process of starting a demo, and also to provide reference implementations of key algorithms. This code can then be executed, and the data structure (and operations on it) will then be visualised to represent the current state of the algorithm. The code editor should act as a debugger, allowing pause/resume functionality and stepping forward and backward.

This will require some way of entering and executing code within a web page. *JS-Interpreter* is a JavaScript parser and interpreter which is itself written in JavaScript (Fraser, 2020). This allows you to parse and evaluate a full instance of ECMAScript 5, which is completely sandboxed from the main JavaScript environment running the page, essentially functioning as a virtual machine. It also allows one to expose objects and functions inside the VM which, when called, will execute code in the 'outer' JS environment before returning to execution flow within the virtual machine. This allows us to create a *domain-specific language* based on JavaScript, in which we expose custom data structure objects inside the VM (such as a list) which, when modified, update the visualisation outside of the VM with the new state of the data structure.

For example, if the algorithm moves an element in the array, the custom VM object would trigger an animation in the UI of the tool to show the element moving, before returning control flow to the VM. Algorithms can then be written in JavaScript to run in the VM using these data structures. In theory, this allows us to implement an algorithm as one normally would in JavaScript, and we can then visualise its effects without having to specifically amend the implementation to do any kind of drawing. This is my take on the WYSIWYC approach—the VM's inner API provides data structures *which visualise themselves*, which completely blurs the line between implementing an algorithm and visualising it.

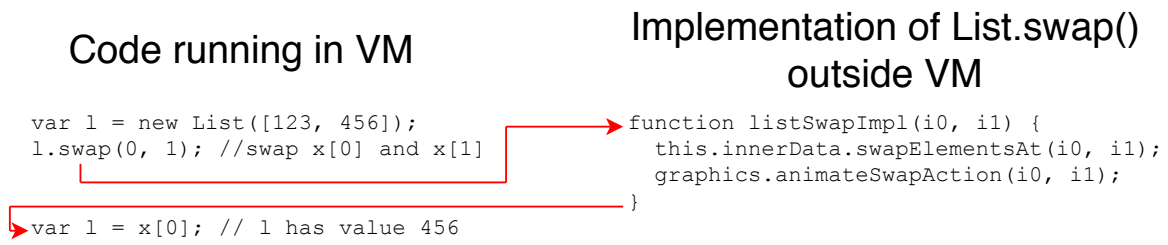


Figure 2.1:
A pseudo-JavaScript example of how control flow moves out of (and back into) the VM, so that algorithms running within the VM's sandbox can affect the state of the AV in the tool's interface.

JS-Interpreter also supports stepwise evaluation and pausing functionality, which is required for the tool's operation.

The choice of JavaScript as the basis for the DSL is partly a technological constraint. This library is the only JavaScript solution I could find which is powerful enough to support interaction with the VM in this way. Additionally, as there is a translation layer between native JavaScript objects and the VM's internal representation of the JavaScript objects used by the VM, it makes sense for the 'outer' and 'inner' languages to be the same. Luckily, there is a wealth of online resources for JavaScript; while *JS-Interpreter* does only support ECMAScript 5 (which lacks lambda expressions, among other features of modern ES6), the use of JavaScript should hopefully not impede the tool's accessibility to computer science students who have perhaps not used JavaScript itself before, as its syntax is similar to other languages like C and Java, which the student has hopefully been exposed to in a computer science teaching context.

A caveat of this software package is speed. The documentation suggests JavaScript running in the VM can be up to 200 times slower than native JavaScript running in the browser. This would mean executing an algorithm with a large input data set could be slow. However, the tool would already be limited by how much data can be visualised on screen. Also, the animation steps themselves need to be slow enough to appear smooth (as per the requirements), so a demonstration of a 1000-item list being sorted (for example) would already be affected by the ability to display that data on screen, *and* the time it would take to animate all the steps of a sorting algorithm with that much input data. The VM only serves to orchestrate the demo, not to practically solve large problems. For these reasons I don't believe the speed of the VM is a concern.

2.1.4 Group Participation and Networking

To satisfy the fourth requirement, demonstrating an algorithm must be a group activity. Students using the tool must be able to participate in a demonstration in some way. To facilitate this, an adaptation of the model used by *Kahoot!* is a viable approach. an instance of a running algorithm (a 'session') could be identified by some kind of code, visible in the tool UI (henceforth referred to as a *demo code*). A student or other participant could then enter this code, and watch the same demonstration on their own device. The same interface should be used for both presenting a demo and watching one, with some indicator that one is in 'participant' mode and not 'demonstrator' mode. A read-only copy of the content of the demonstrator's code editor should be mirrored in the participant's UI. Participants should be able to mirror the demonstrator's AV display on their own device, as well as step forward and backward independently of the demo if they wish to revisit or catch up to a concept they missed.

In a lecture, this would mean the demonstrator's UI could be projected onto a whiteboard or screen, along with the AV and code. Students can then mirror this display on their own

device (a smartphone, for example), allowing them to scroll through the code independently of the demonstrator, and rewind the demo on their own device—perhaps to recap a stage of an algorithm they did not understand—before seamlessly 'rejoining' the group demonstration, without requiring the demonstrator to interrupt the demo for the whole class if a single student wishes to recap a previous part of the demo.

Implementing this collaborative approach will require networking. There is no method to allow JavaScript running in a browser on one device to communicate directly with client side JS running on another device, so this project will require a client-server architecture. Ajax requests are useful for simple one-time request-response communications: for example, the client queries some data to present in a list in the UI, and the server responds with said data; the connection is then closed. However, we will need the demo to update in real time. The demonstrator will essentially be streaming changes to the current state of the demonstration. This cannot be effectively modelled using Ajax requests—a single request/response cycle cannot do this, and a constant stream of Ajax requests to poll for updates to the demo would be slow due to the overhead of a full network connection and HTTP session being established each time. Fortunately, *Websockets* are a JavaScript API feature which expose a bi-directional socket-like channel to client side code. This allows a connection to persist between the code running in the client, and the server software, through which messages (as strings) can be sent and received.

While most places will likely have available an available Wi-Fi connection, this cannot be assumed to be reliable or even present. Students may need to make use of cellular data to participate, especially in crowded lecture theatres. This constrains the volume of data which the tool can expect to reliably send and receive. The state (and changes to the state) of the demonstration must be represented in a way that isn't going to require a lot of bandwidth. This rules out using image or video–based representation to share the state of the AV over the network: sending a list of images would require a lot of bandwidth if we wanted smooth animation, even with JPEG compression, and video would require some kind of encoding/decoding which is unnecessary extra scope for the project. A text-based markup representation can be used to serialise the relevant parts of a data structure, combined with a frame-based approach, wherein each 'step' of the demonstration (such as an instruction like *"swap the 0th and 2nd elements in the visualised array"*) is represented and transmitted as JSON, which can then be visually reconstructed by the clients.

To implement this demonstrator-participant model, the demonstrator's client could send information about the demo, including these frames, to a server. The server then forwards these on to the clients. A client could identify which demo it wants to participate in by specifying the demo's code as part of the handshake. The server could then forward all information about that demo as it receives it onto the client. A client connecting to the server could alternatively specify that it wants to act as a demonstrator, upon which the server gives the client a demo code to use for itself and display in its UI, so that other users can participate in it.

Handshake for Demonstrator	Handshake for Participant
Demonstrator → Server: Start demo.	Participant → Server: Join demo.
S → D: Demo code is **2345**	S → P: Which demo?
(demonstrator displays 2345 in UI)	P → S: 2345
D → S: *(sends demo info to server)*	S → P: *(forwards demo info from demonstrator)*

Table 2.1:
How this demonstrator-server-participant setup could be established.

The 'interactive prediction' requirement described by Rossling and Naps covers the process of asking questions throughout the demonstration about what happens in the next step of the algorithm. The tool is intended to be used as a supplement to a lesson or lecture, so

the demonstrator could accomplish this requirement the old-fashioned way: by pausing the running demonstration, and asking a verbal question to the participants. However, as part of the gamification idea, a mini quiz-like functionality could be added, where the demonstrator can ask a question within the demo tool, which appears on the participant's devices. This would essentially be a subset of *Kahoot!*'s functionality, and their design study suggests this gamification aspect would improve student's engagement. Rossling and Naps' research would suggest that this would help to prevent confused students from falling behind, as described previously.

2.2 Functional Requirements

The section above describes the intended usage of the tool in a way which meets the overall requirements. We can then break this plan down into smaller formalised requirements, represented as user stories, based on who is going to be using the tool for each specific area of functionality. There are three separate user roles for this application:

- **Demonstrator**. This represents a user who is in control of an AV; this will typically be a teacher or lecturer. This could also be a student teaching a peer, referring back to the *Demonstrator* level of student AV engagement identified by Naps et al. These requirements describe how the demonstration can be controlled.
- **Participant**. This is someone who has entered the *demo code* on their own device, such that their device running the tool is mirroring the code and AV being displayed on the Demonstrator's screen. This also represents the Demonstrator themselves; consider a student using the tool in their own time to learn about an algorithm. They are effectively presenting the demonstration to themselves. These requirements describe how the info presented by the demonstration is to be consumed.
- **Creator**. This represents a user who wants to *create* a demonstration, rather than just run a pre-made one. The requirements of the Creator reflect the functionality required of the virtual machine's API exposed to the implementation of an algorithm.
  A Demonstrator may also be a Creator, but the roles encapsulate different goals: the requirements of a Creator describe which visualisation functionality is required to create an effective demo, whereas the Demonstrator role represents the requirements of the tool from a teacher's perspective when running a demonstration which is already implemented. The Creator's requirements represent primarily *what* can be demonstrated, whereas the requirements of the other two roles cover how the demonstration can be interacted with.

Some of the requirements are more important than others in terms of achieving our goal of improving upon existing methods of demonstrating algorithms. In order to prioritise these in the development of the tool, I have adapted the MoSCoW method, where each identified requirement is classified as *must have*, *should have*, *could have* or *won't have*. This is normally used to prioritise requirements in a business context as part of an agile development process; given a fixed delivery deadline, it allows a team with time constraints to defer requirements with lesser (or less immediate) business benefits first, preserving more important requirements. This method is applied to this project, too, with a fixed amount of time to create the demonstration tool. The set of requirements classified as must-have define the *minimum viable product* which must be met in order to achieve the goal. The should-have requirements can be dropped if they would take too much time, or be nonviable given the chosen technology, but should otherwise be considered important.

Note: the choice between *should* and *could* is made somewhat arbitrarily, but I have considered the effective gain from meeting such a requirement compared to the estimated amount of effort required to implement it. A drawback of the MoSCoW approach is that, in larger projects with multiple development iterations, there is a tendency to categorise easier-to-implement

requirements with more immediate business benefits as *must-haves*, and de-prioritise those which may be important, but have a less immediate tangible impact, such as refactoring (John McIntyre, 2016). However, due to the nature of this project working with a set amount of time over a single development cycle, this drawback may work slightly in our favour; as noted, the *must-haves* specify the minimum viable product, so priority is given to the "low-hanging fruits" which allow us to rapidly reach that product, over any other smaller refinements.

Requirements classed as *won't have* due to time or technology constraints are outside the scope of this project, but are identified later as being useful refinements for potential future expansion. For these requirements, as well as any of the *should-have* and *could-have* requirements which are not implemented in time, the architecture of the tool should be extensible enough so that these could be implemented without extensive re-writing of existing components.

2.2.1 Demonstrator

As a demonstrator, I want to...

1. (**must**) ... load a demonstration of a built-in algorithm.
2. (**must**) ... edit the implementation of a demo, so that I can affect the algorithm's behaviour, or change the input data.
3. (**must**) ... implement my own algorithm to create my own demonstration.
4. (**must**) ... start, pause, resume and stop the execution of the code.
5. (**must**) ... step through the algorithm line-by-line.
6. (**must**) ... control the display for participants using my own position in, and panned/zoomed view of, the demo.
7. (**must**) ... see an error description when the algorithm stops working or cannot be parsed.
8. (**should**) ... step *backward* through the algorithm line-by-line.
9. (**should**) ... jump to notable 'anchor points' in the demo, such as a specific step of an algorithm.
10. (**should**) ... see the execution point of the algorithm in the code editor.
11. (**could**) ... create a quiz question for participants to answer.
12. (**could**) ... edit the code of a running demonstration, as in the WYSIWYC approach described by Hundhausen et al.

2.2.2 Participant

As a participant, I want to...

1. (**must**) ... join a demo, including ones already in progress.
2. (**must**) ... see a copy of the code that the demonstrator is editing or running.
3. (**must**) ... see the AV as displayed on the demonstrator's machine.
4. (**must**) ... see annotations or hints about the current state or stage of the algorithm.
5. (**must**) ... pan and zoom the AV display.
6. (**should**) ... start my own demo instance, using the code from an existing one.
7. (**should**) ... 'detach' from, and step forward and backward independently of, the demonstrator.
8. (**should**) ... show or hide hints.
9. (**should**) ... show or hide the code display.
10. (**should**) ... change the speed of the demonstration.
11. (**should**) ... jump back to, and 'resynchronise' with, the demonstrator's demonstration state.
12. (**could**) ... replay the demonstration from a rewound state.
13. (**could**) ... see the execution point of the demo's local state on my device.

2.2.3 Creator

As a creator, I want to...

1. (**must**) ... implement an algorithm using a fully fledged programming language.
2. (**must**) ... visualise the data structure my algorithm works with.
3. (**must**) ... visualise the *changes* made to the data structure by the algorithm.
4. (**must**) ... annotate the visualised data structure from code, using colour codes and other visual cues, in order to identify how an algorithm is currently working with a data structure.
5. (**must**) ... add text hints/cues to the display in order to describe the algorithm, from code.
6. (**must**) ... write algorithms that operate on **lists**.
7. (**should**) ... write algorithms that operate on **graphs**.
8. (**should**) ... zoom and pan to a specific part of the data structure, from code.
9. (**could**) ... write algorithms that operate on **trees**.
10. (**could**) ... auto-generate randomised input data for my algorithms. *(rather than implementing this manually)*
11. (**could**) ... save my custom algorithm in the tool for later use.

At least one data structure is required to demonstrate the principle idea of the tool. The list was prioritised the highest here, because it is perhaps the easiest data structure to visualise technically, while allowing a range of algorithms to be demonstrated on it. Also, lists are closest to a pre-existing concept in JavaScript; one of the main goals of the tool is to be able to visualise an algorithm the same way you'd normally write code, and as JavaScript supports variable-length arrays (i.e. lists), this means the API for the data structure can be designed in a way that users will be familiar with.

Graphs were prioritised after lists as they will be slightly more complex to add, but also offer a wide range of algorithms to demonstrate. There is no existing language construct representing a graph in JavaScript, however, so this will require us to design our own API for a graph object. Trees were prioritised last. These are a subset of graphs, but while graphs typically represent objects with some spatial connection, trees typically serve a different purpose, such as representing array data in a different way in order to implement algorithms more efficiently (such as heaps). Supporting trees will also require further planning to develop an API for the algorithm to use, and a nice way to visualise them, so these were prioritised last in order to reach a minimum viable product quicker.

2.3 Non-functional Requirements

There are also a number of non-functional requirements which affect some of the choices made in the implementation.

- **The networking code should be able to deal with unstable connections**, to a degree. In a lecture, cellular data and Wi-Fi can both suffer due to crowding of devices in a small area. For example, if the demonstrator were to lose connection for a few seconds, the demonstration should not end; instead, the demonstrator should be able to seamlessly rejoin and carry on the demo when their device reconnects.
- **The number of interactions with the interface should be minimised to carry out the most common tasks**. For a teacher, this will likely be starting a built-in demo. For a student, this will be joining a running demo started by a teacher. Therefore the functionality to carry these out should be exposed in the top layers of the UI, without needing to drill down into too many other screens.
- **The tool should load quickly**. Ideally, the UI should start to render as quickly as possible once the page is loaded so users do not lose interest. This means the number of

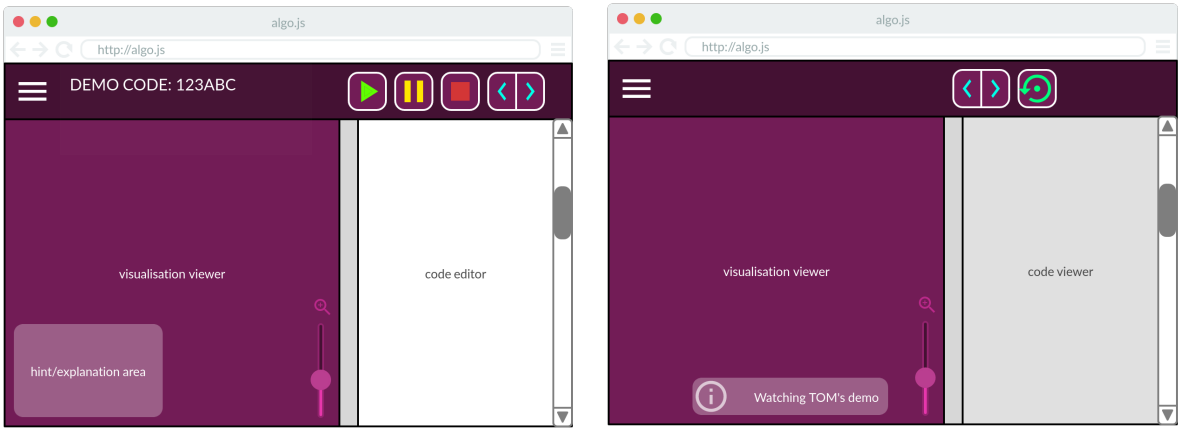
resources loaded by the page, including images and scripts, should be kept to a minimum.

- **The user interface should be intuitive to use on all target form factors**, such as phones, tablets, laptops, etc.—altering the UI where necessary—*without* making the interface appear substantially different across devices. For this, responsive design principles can be applied. As the DOM will be used for the UI, CSS media queries allow alteration of the stylesheet for different display sizes, which can be used to adapt the layout of interface elements when being presented on devices with smaller screens. JavaScript also exposes specific APIs for touch interaction, which will allow us to support touch-and-drag or pinch-and-zoom in the UI.
- **The system for exposing different data structures to the VM should be extensible**, so that the base system does not have to be modified to allow further data structures to be added later.

2.4 User Interface

As mentioned in the introduction, there is some mental effort involved in switching between, and familiarising oneself with, different teaching tools. This may interrupt the flow of a lesson, or the 'train of thought', of a student. To make the use of the demonstration tool as natural as possible, the interface should re-use concepts and design cues which students will already be familiar with. The code editor, and the flow control options, should be modelled after the ones used by standard development IDEs, including play/pause and step controls. Additionally, syntax highlighting should be used in the editor as a further visual cue.

Controls for panning and zooming should also be intuitively obvious, but also not require excessive button clicks. Rather than having to enter a dedicated mode for panning/zooming—for example, having to select a pan tool in the UI—the user should be able to click and drag to move around the visualisation. The same applies to mobile devices, where users can reasonably expect to be able to drag with their finger to move a display around, and pinch their fingers together to zoom—this also frees up space in the UI on mobile for functionality which does have to be exposed using buttons or other UI elements. This is a mock-up of the primary user interface for the tool when being used from the demonstrator's perspective.



(a)
Demonstrator's UI.

(b)
Participant's UI.

Figure 2.2:
A mockup of the user interface for the demonstrator and participant.

The interface for the demonstrator and participant should be presented in the same way, but

with some indication to distinguish one from the other. Elements of this user interface include:

- The visualisation is presented alongside the code. In keeping with the idea of blurring the distinction between writing code and demonstrating an algorithm, this allows immediate feedback for a Creator to see how the effects of their code. The research by Hundhausen and Brown suggests this will make the process of creating algorithm demos more rapid, compared to processes with a longer feedback cycle, such as an interface with a separate code editor and visualisation mode.
- The play/pause/step toolbar is reminiscent of existing IDEs such as Microsoft Visual Studio, which themselves model the interaction with code on that of a video or tape player. For our demo tool, this should hopefully be especially intuitive; the code is itself driving the visualisation, so these buttons act like a debugger *and* a video player.
- The demo code is clearly displayed in the UI for the demonstrator. As the tool will be used in lectures and lessons, this will allow students to see the code on a projector or interactive whiteboard.
- A 'zoom bar' is used to represent the zoom level. A similar UI element is used in some web browsers and text editors.
- A semi-transparent hint box is overlayed on the AV area, allowing demo creators to provide a further description of what the demo is representing. This should be toggle-able, but should also re-appear when the algorithm has new information to provide, so that users don't miss any important information. This will be visible on the participant display, too, but was removed to make the diagram clearer.
- A *'hamburger menu'* icon is present in the top left. This is a standard icon indicating that a click or tap opens a menu with more options. By hiding infrequently used options in a menu, we can keep the rest of the UI visually tidy.

One of our nonfunctional requirements is the ability to gracefully handle unstable network connections. If the connection drops temporarily, the user should still be able to interact with the user interface, but it should be obvious that the demo will be interrupted until a connection is re-established. If a demonstrator loses connection, their UI should indicate that participants may not see their actions or changes reflected locally. The server will be able to tell if a demonstrator's connection closes unexpectedly, and it should inform participants that the demonstrator is currently trying to re-connect. Conversely, if a participant loses connection, the UI should indicate that their on-screen AV may not reflect the up-to-date state of the demonstration. The hamburger menu and network warning are shown in diagram C.1, found in the appendix.

The main menu should allow the user to immediately select what they intend to do with the web app. There are 3 potential goals for a user loading the tool, corresponding to our 3 user roles. The first 3 options in the welcome menu correspond to these modes, which will load the appropriate UI. These UI screens are both shown in diagram C.2, also found in the appendix.

Chapter 3

Architecture

The implementation of the tool is made up of the client side implementation, which consists of the HTML/CSS definition of the UI layout, and the JavaScript implementing the logic; and the server side implementation, which serves the client side scripts and HTML/CSS resources over HTTP(S), as well as the networking logic which communicates with the client side JavaScript via a websocket.

3.1 Server

The server is written in Python, and based on the lightweight HTTP micro-framework *Bottle*. The web aspect of the demo tool's server side component is mostly just serving static content rather than any heavy dynamic content generation. Bottle provides a simple syntax for serving static content from the server's file system which may be retrieved with GET requests.

3.1.1 Demo Storage

The demos built in to the tool are also stored on the file system of the server; they are categorised by the purpose of the algorithm, such as finding the minimum spanning tree, or sorting. There is a directory on the file system for each category. In each of these directories, a `category.json` file contains a human-readable name and description of the category, and also a `.js` source file for each demo. The content of each demo file is prefixed with a multi-line comment containing the full name, description and author of the algorithm in a JSON object. The demos are exposed to the client through a mini web API with two endpoints: one which serves a categorised list of demos, and another which provides the source code for a given demo, which are called with Ajax requests (using *jQuery*) from the client.

The first endpoint (`HTTP GET /demos`) traverses the category directories, including the `category.json` files and the JSON comments in the `.js` files, to produce a flat JSON representation of the entire list of demos that the server has, which is then sent as the response to the endpoint; this is used by the client in order to populate a list of demos in the UI. The second (`HTTP GET /demos/<demo-id>`) allows you to access the actual source code of a demo, given a category and demo name, with the JSON metadata removed; this is used by the client once the user has chosen a demo in order to fetch its source code. The server maps the demo ID such as `sorting.quicksort` to the demo's file path `/sorting/quicksort.js`.

3.1.2 Websockets and Demos

The stock Bottle framework doesn't support websockets. A third-party library *bottle-websocket* is used, which extends Bottle's functionality with websocket support (Kelling, 2020); this exposes a websocket connection with a standard socket-like API (with blocking read and write calls) allowing Python code to deal with multiple clients connected to the server's websocket.

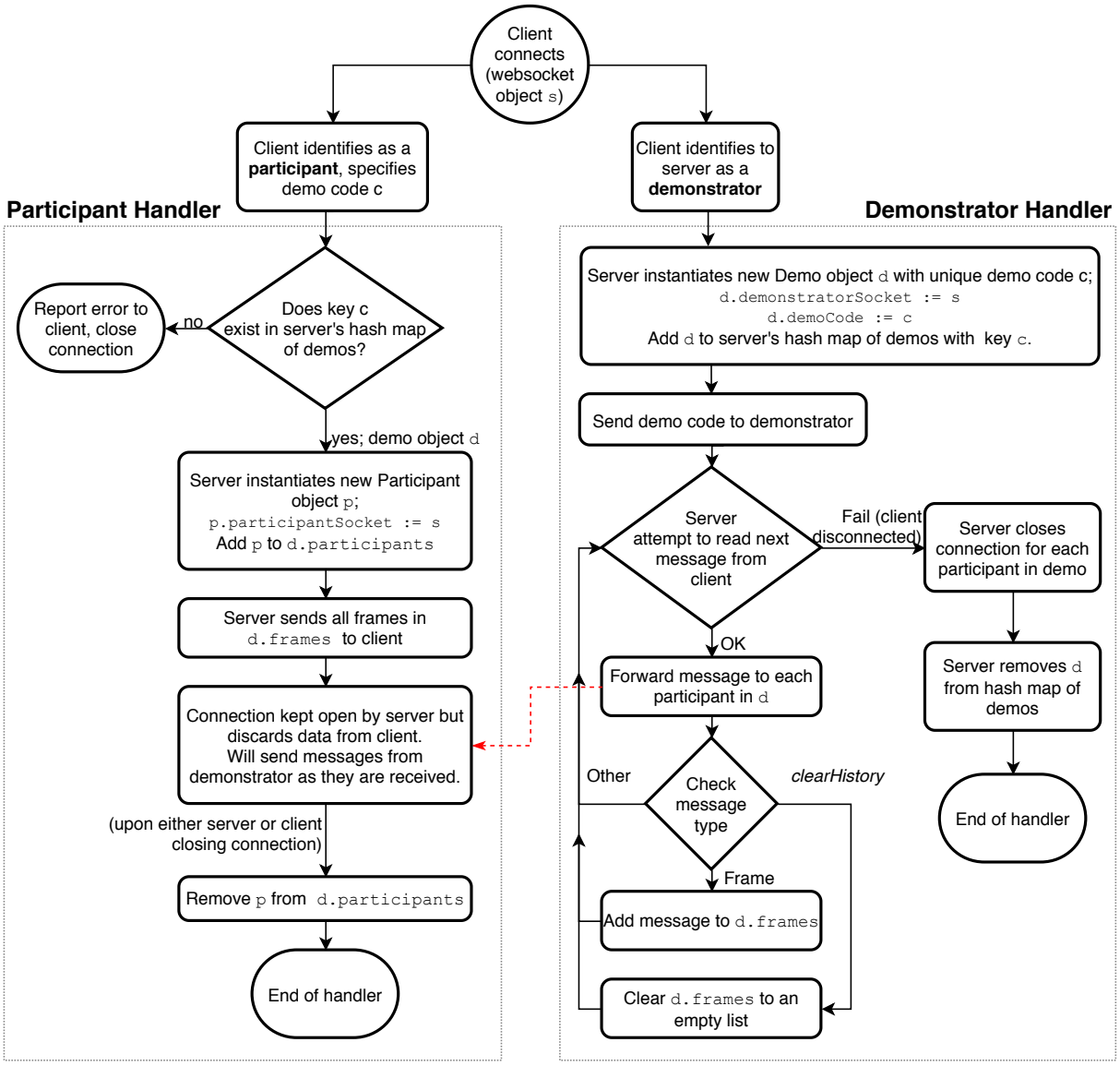


Figure 3.1:
The flow control of the server's websocket handler. The red dotted line represents data received from the demonstrator being forwarded by the server upon receiving them onto participants watching the same demo.

When a client connects and establishes a connection, the server associates the client's websocket to either a `Demo` or `Participant` object, depending on how the client identifies; these objects are just stored in memory. This allows code running in the handler for a demonstrator's connection to send data to the sockets of the participants watching the demonstration. Other than the demos stored on the file system, these objects are the only information stored by the server. An actual instance of a demonstration (such as a lesson) is transient, as it only lasts for the duration of a lesson, meaning this data does not need to be persisted to (for example) a database. The

operation of the websocket handler is described by figure 3.1. A *frame* represents an update to a demo; the details of how frames are used are described later.

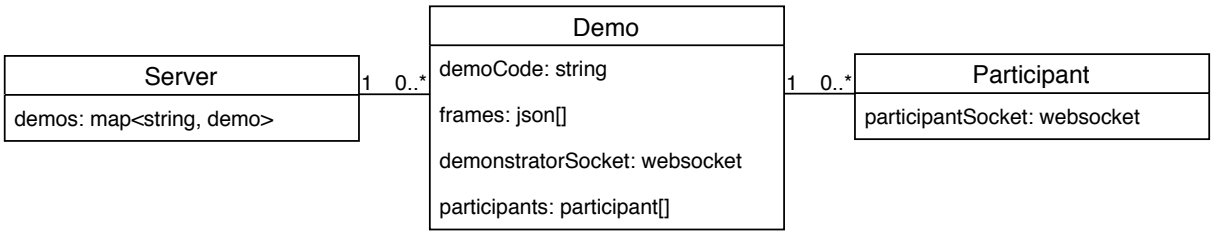


Figure 3.2:
A small entity relationship diagram of the structures used by the server to track ongoing demos.

Ajax requests to fetch the source code for an algorithm are made separately from (and can be made at the same time as) an ongoing websocket connection; separating this from the server logic for connected websockets keeps the implementation simple, as the server doesn't need to track as much state for each ongoing connection. This means that the server needs to perform minimal processing on received data once the connection is established (as either a demonstrator session or a participant session). The only processing performed on data received from the demonstrator is to check the *message type*, as shown in figure 3.1; any *frame*-type messages are stored on a per-demo basis by the server in the demo's `frames` list. When a new client connects, these messages are sent to the client, so that participants joining a demo in progress can still rewind to a point of the demo before they joined. The server doesn't perform any further processing on the frames—see the *Frames and Messages* section for elaboration of these terms.

The source code for an algorithm is static data; that is, data which is written once and read many times. Separating the process of acquiring a demo's source code from the server logic involved in actually *delivering* a demonstration over the internet means that, were this project to be implemented on a larger scale, storage of demos could be moved to a dedicated static storage platform such as Amazon S3 without many painful changes; the server would just query S3 rather than the file system. Alternatively, access to static resources via S3 could be mediated by an entirely different server, in which case the Ajax requests could be made to an API endpoint exposed by that server on a different subdomain.

If further expansion proved necessary, this design would allow for a decentralised approach where demos can be ran on different servers, as there is no interaction between different demos. Imagine a network of regional *algo.js* servers around the country. When starting a demo, a master server could inform the client of which *regional* server to connect to. The web app would then connect to the websocket of this regional server. If each regional server is given a 2-character identifier, then this could be prefixed to the alphanumeric demo code which students use to specify which demo to connect to. Provided the web app has a static list of all the regional servers, along with their 2-character code, then the demo code would contain enough info for the client to identify which server to connect to, along with which demo to join on that server.

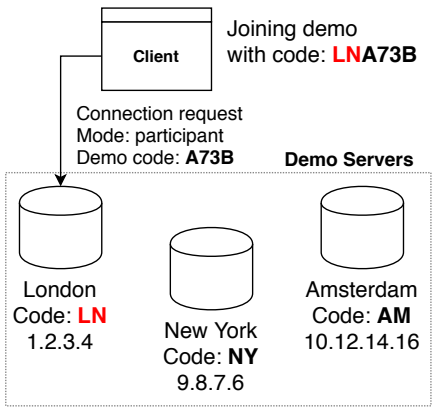


Figure 3.3:
A diagram of how demos could be decentralised across different servers as a form of load balancing.

3.2 Client Overview

In order to separate the various concerns of the demo tool's web app functionality, its implementation is split across several modules. Some functionality of each module is abstracted and its concerns are hidden from the other components of the implementation. In this way, some of the modules act as *context widgets* specific to this project, by hiding the complexity of their implementation behind a clean API. (Dey et al., 2001) This also allows us to better represent how the different modules communicate with each other, as none of them should be exposing details of their implementation in their interface.

One of our non-functional requirements is for the tool to load quickly. Our code is split across several modules, meaning the browser must fetch and load several JavaScript files referred to in `<script>` tags before we can start doing anything. By default however, a browser will fetch and load each script synchronously, meaning page loading (including of further scripts) is blocked until code referred to in each `<script>` tag is loaded and fully executed. The loading process may be prolonged if the internet connection is not stable or is crowded. To avoid this, the `defer` attribute can be used to force the browser to fetch script resources in parallel as the HTML is being parsed, and then execute them once this finished.

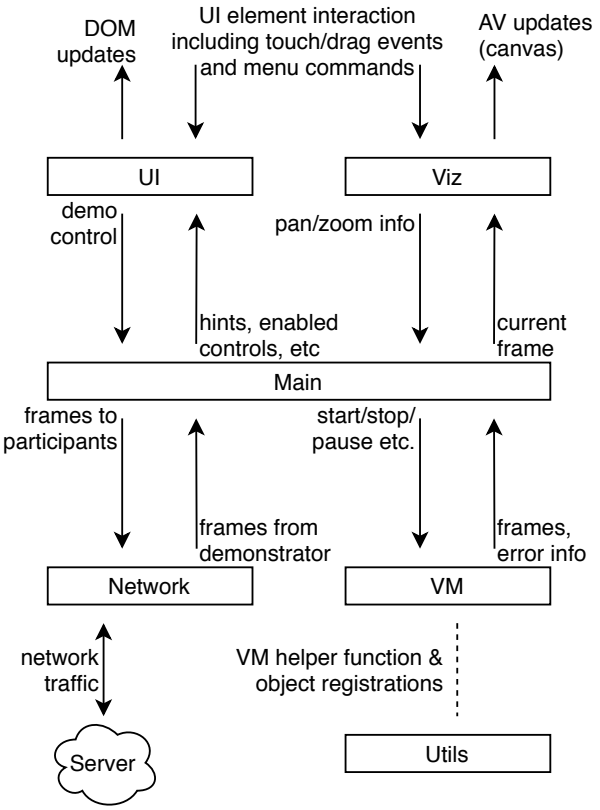


Figure 3.4:
An overview of the core modules making up the clientside implementation of the tool, including the flow of interaction between the various components.

The *module pattern* in JavaScript allows us to define a hierarchical namespace system across multiple files, which can be loaded independently of one another and in any order (Cherry, 2010). This is compatible with the deferred script loading approach and is what algo.js uses to structure its modules.

Another requirement is extensibility in terms of adding new data structures. The implementation of a data structure is split in two: the VM object implementation part which is called into by the VM module whenever a demo is loaded, defining the prototype of the object and the methods exposed to code running in the VM (which manipulate the underlying internal representation of the data structure); and the visualisation part, which registers handlers called into by the Viz module in order to render the data structure and animate changes made to it. A separate source file for each data structure comprising these two parts is written as a module. Each such module can be loaded independently.

3.3 User Interface and UI Module

The UI is split between the layout definition, written in standard HTML/CSS; and the code dealing with interaction with that layout, which uses jQuery to manage user interactions with

UI elements.

As mentioned in the requirements section, heavyweight UI frameworks like React have been avoided in this project for speed and simplicity. The UI has been designed to leverage CSS as much as possible; for mobile browsers especially, taking advantage of the CSS layout engine will be quicker than manually calculating layouts in JavaScript; newer features of CSS such as *flex boxes* allow UI elements to be scaled relative to one another without manual JavaScript intervention. The interface consists of the base display as shown in figure 2.2, including the toolbar at the top of the display, the visualisation viewer, and code editor. This is always shown on screen. Other elements, such as the welcome screen, the menu corner in the top left, message boxes (such as those used to display parsing or runtime JavaScript errors), and the algorithm picker menu are displayed as overlays on this content using CSS `absolute` positioning. If one of these overlays is present, it is shown over a semi-transparent overlay; this visually indicates that the dialog is modal, and prevents interaction with the underlying main UI.

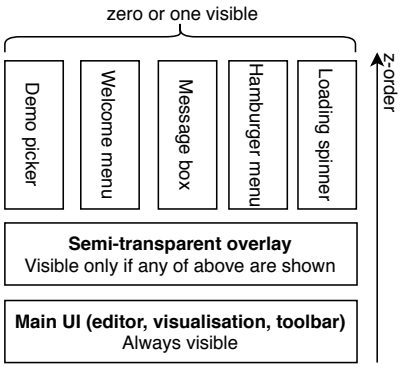


Figure 3.5:
The layered UI design.

Icons from the the free version of *Font Awesome* are used in the hamburger menu, and in the buttons to control code flow in the toolbar. The icons are embedded as glyphs in the font, meaning they scale correctly as vector images when resized. Another visual cue used is the *loading spinner*. This is another overlay used as an intermediate display when the web app is processing something. This is displayed when the user opens the algorithm picker menu, while the Ajax request to fetch the list of demos is still in progress; it is also shown while the main websocket connection is being negotiated. The animated cue should stop the user from losing interest in (and disengaging from) what would otherwise appear to be a non-responsive UI. The spinner is animated using pure CSS, derived from a template provided for free under a CC0 licence (loading.io, 2019). The *Ace* code editor widget is used to provide off-the-shelf syntax highlighting and code folding, as well as support for highlighting regions of code (ajax.org, 2020). It allows us to highlight the execution point in the code, like Visual Studio does with the yellow arrow indicator.

One of the non-functional requirements is for common tasks to require minimal UI interaction. From the welcome menu, you can start a demo with 3 clicks: start demo, pick category, pick demo. You can also join a demo with 1 click and some text input—to make this nicer on mobile, the keyboard appears automatically when this option is picked, and is hidden once the demo is joined. Besides the welcome menu, most common demo-related tasks can be done without drilling down; the UI presents most available actions on-screen at once. Less common actions are in the hamburger menu, which is only one extra click. Adhering to this rule also helped avoid feature bloat: "if I'm struggling to fit something in the UI, does the tool *really* need it?"

One of the other non-functional requirements is for the UI to be intuitive across different form factors. To allow the UI to display appropriately on smaller form factors such as phones, tablets and smaller laptops, CSS *media queries* are used to enable a modified set of layout rules when the device width is below 1250 px. This is the width below which the lack of horizontal space causes UI elements to render incorrectly. This media query enacts a number of changes:

- The font size for dialogs, including the welcome menu, is reduced considerably.
- The hint box annotation in the visualisation area is scaled down, and the negative space around it is narrowed, making more of the visualisation visible.
- The flex box scaling for code editor window is changed so that the editor takes up less of

the screen. The reasoning for making this *smaller* rather than larger is that users with a device which is too small to fully show the UI are more likely to be participants rather than demonstrators, who will likely be using a laptop or desktop with a fully-sized display. Participants are less likely to be interested in reading or editing in the code window, and having more screen space for the visualisation is probably more useful.
- The code control flow buttons are scaled down, and the labels ("play", "pause", etc.) are hidden. The icons are still visible, so users still have an indication of what the buttons represent.

The JavaScript code handling interaction with the layout is in the UI module (`algo.ui.js`). As mentioned, jQuery is used to manipulate the DOM. For the overlays mentioned above, displaying these (and hiding them afterwards) simply involves selecting the relevant `<div>` element—the different overlays have unique IDs with which to select them from the DOM—and then toggling the `display: none` CSS rule appropriately. Besides the standard `click` event handlers for buttons, this module also consumes the UI events for panning and zooming, including click-and-drag functionality. To correctly deal with touchscreens, this involves processing the `touch`-based events to provide equivalent functionality, as web browsers don't emulate the standard `mouse`-based events for touch interactions. These events are processed and used to produce a zoom level and pan offset, which is provided to the Visualisation module so that it can render the AV with the correct scale and translation. This separates the concerns of calculating the pan/zoom values, and actually using them to alter the AV rendering.

The UI module also updates the code control flow button icons and labels to describe the presently available state machine transitions, which are described in the Main module section. Finally, it also sets up the *Ace* code editor widget and deals with highlighting (such as indicating the currently executing line of code).

3.4 Frames and Messages

Information about (and updates to) a demonstration are all internally represented using *messages*. These are stored as JavaScript objects in memory, and transmitted as JSON when sent over the network. Updates are sent by the demonstrator to the server in the form of messages, and then forwarded to the participants, once a websocket connection is established. A message object has a `type` attribute, as well as any number of other attributes parameterising a message of a given type. For example, a `hint` message looks like this:

```
{
  "type": "hint",
  "title": "Hint Title",
  "text": "This is a hint.",
  "id": 12
}
```

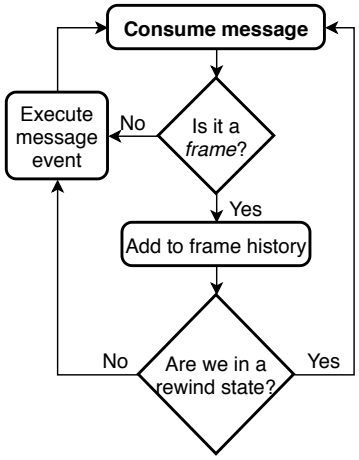


Figure 3.6:
An overview of the logic for consuming new frames.

Each message also has a *handler* - this is a function which consumes the message and performs whichever operation or state change it represents. Executing the handler for the hint frame above, for example, would make the hint box visible and update its text to that in the message data. A distinction is made between message types which are part of a demo's "timeline", and those which aren't. The former are referred to as *frames*. These are messages which represent changes

to the demo's state—that is, those which you would expect to be replayed in sequence were the demo to be rewound and played back—whereas non-frame messages represent immediate or one-off events. Events represented as frames include animations and hints, but not pan/zoom tracking, or the message indicating that a new demo is starting. Frames may not necessarily be handled when they are consumed; see figure 3.6. However, frames are stored in a *frame history* buffer so that they can be replayed as directed by the user.

The state of the demo consists of four things:

- The visualisation itself. This is affected by frames of type `setViz` and `animate`, which initialize and animate the AV respectively—see the Viz module section.
- The demo key (the colour key in the top left), affected by `setDemoState` frames.
- The hint annotation in the bottom left, used to describe what the algorithm is doing; updated with `hint` frames.
- The current execution point, shown in the code editor to represent what line the VM is executing. Affected by `step` frames—see the VM module section below.

3.5 VM Module

The VM module (`algo.vm.js`) essentially functions as a wrapper around the *JS-Interpreter* system. It encapsulates all of the logic involved in stepping through the VM and exposes it to the rest of the system as a simple three-state machine: *stopped*, indicating that the VM is not running; *running*, indicating that code is actively being executed; and *paused*, indicating that the demo program is still in progress but not being actively executed. The VM module also consumes parsing and runtime errors and calls into the UI module to display the error message and highlight the line with the error. Also, this module handles all data type and helper objects exposed to code running in the VM to interact with the demonstration.

JS-Interpreter executes code by stepping through the parsed AST. This sacrifices the efficiency of bytecode-based approaches for benefit of exposing exactly what is being executed at any given time. Its built-in parser also stores the position in the source code that each AST node was parsed from, making it trivial to highlight the current execution point in the code editor.

The library does not have a dedicated *play* function for continuous execution. That would be a blocking operation, as it would mean the algorithm would have to fully execute before control returned to our system. Instead, it exposes a `step()` method, which essentially performs one evaluation 'iteration'. In order to implement the 'play' function, `setTimeout()` calls are used to schedule repeated executions of `step()`-ing the interpreter with a short timeout, without blocking UI interaction. Pausing is just as simple as stopping this loop.

There is a trade-off, however: JavaScript timeouts are not guaranteed to execute precisely on time, as the browser may perform other tasks before executing a due timeout, which introduces unpredictable latency and slows down execution of the VM. This is especially so on mobile browsers, due to the more aggressive power saving measures on mobile platforms. The VM module works around this by `step()`-ing 1000 times for each scheduled `setTimeout` invocation. 1000 steps execute quickly enough to avoid blocking the UI thread for any noticeable amount of time, while being fast enough to practically execute an algorithm.

There is also some additional logic involved in pausing and manually stepping through the VM. One of our user stories is for the demonstrator to step through the code *line-by-line*. One `step()` call does not correspond to one line; in fact, many such calls may be needed to evaluate a single line. The module checks the type of the current AST node which the evaluator is focussing on, and the parent of that node, to allow evaluation to pause in 3 situations:

- When the current AST node is of type `ExpressionStatement`, so evaluation can pause on standard expressions such as assignments or function calls.
- When the current AST node is of type `VariableDeclaration`.
- When the current AST node is *not* a `BlockStatement` and the parent node is any `Statement`-type node *except* an `ExpressionStatement` or `BlockStatement`, so that evaluation can pause on the conditions of `if`, `while`, and other similar syntax constructs.

The 'pause' operation exposed by the VM module will carry on stepping the interpreter until one of the above is met. This behaviour was designed to emulate pre-existing IDEs as much as possible (specifically Visual Studio), so that the tool can be used naturally by the user without needing to adapt to unfamiliar pausing behaviour. **Note**: `ExpressionStatement` and the other AST node types are the type names internally used by JS-Interpreter, rather than anything defined in this project.

3.5.1 VM Registrations

JS-Interpreter offers a very powerful system to allow native JavaScript functions (including prototypes) to be registered so that they can be called from inside the VM. Objects and functions which the VM needs to use must first be converted to the VM's internal representation of these constructs; this is called a *pseudo* form. For example, the following snippet is used to register the native function `hintHook` so that it can be referred to as `Demo.hint` from within the VM:

```
// create pseudo form of an object
var demoVmObj = vm.createObjectProto(vm.OBJECT_PROTO);
// set the 'hint' attribute to the pseudo form of hintHook
vm.setProperty(demoVmObj, 'hint', vm.createNativeFunction(hintHook));
// set the 'Demo' attribute in the global scope to our new object
vm.setProperty(scope, 'Demo', demoVmObj);
```

This is done for essentially every bit of functionality exposed inside the VM. On the second line, `createNativeFunction` is used to create a pseudo wrapper around a native function. `createAsyncFunction` offers similar functionality, but allows us to expose asynchronous native functions to the VM—that is, ones which pass the result of an operation to a callback rather than via the function's return value, such as the `fetch()` API for Ajax requests. Inside the VM, they will appear to act as a blocking call. *JS-Interpreter* will pass a callback as an extra argument to the function which, when called, will resume execution of the VM. For example, you could implement a blocking *sleep* method as follows:

```
function sleepFunction(duration, callback) {
  setTimeout(function() {
    // when the timeout elapses, call the callback provided
    // by JS-Interpreter to return control flow to the VM
    callback();
  }, duration);
}
vm.setProperty(scope, 'sleep', vm.createAsyncFunction(sleepFunction));
```

Code written for the VM can then call the function: for example `sleep(200)`, which will block the VM's execution for 200ms but *not* the web app itself. The `step()` function provided by *JS-Interpreter* performs a no-op if it is called while waiting for an async function to return. By

implementing operations on the data structures with these asynchronous functions, we can trigger animations from these functions in the AV which halt the VM until they finish; however, as they are exposed to the VM as a standard synchronous function call, the Creator of the demo does not need to think about callbacks, or otherwise restructure their implementation of the algorithm to work around an animation system. Instead, they can write the algorithm synchronously (as it normally would be), as if they weren't trying to animate it at all. This separates the animation system from the algorithm being animated, which satisfies one of the goals of our project. This feature of *JS-Interpreter* proves very powerful in that regard.

The AV data structure types exposed within the VM have a hidden internal representation of the underlying data, stored in memory as JavaScript objects, and serialisable as JSON. For example, the internal representation for a `Graph` object describes the vertices of the graph, as well as an adjacency list for the edges. This internal representation, however, also stores the parameters needed to visualise the data structure, which aren't classically considered as part of the data structure itself; this includes Cartesian co-ordinates for each graph vertex to be displayed on screen, as well as the colour and vertical offset of each element in the `List` type. The VM module can therefore use this internal representation to store the data which will be accessed by an algorithm, and also pass it to the Visualisation module for rendering.

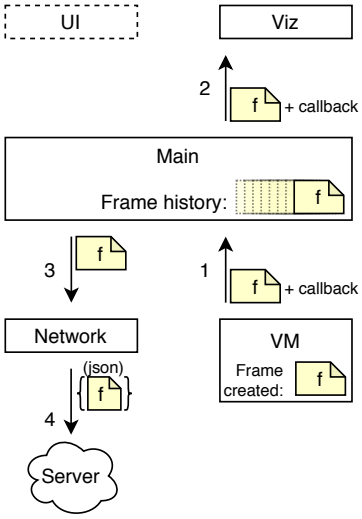


Figure 3.7:
How an animation frame generated by the VM flows through the system modules.

To demonstrate this, take the `List.move(a, b, n)` function. This is an instance method which moves n items starting from index a so that they start at index b after the method is called. Figure 3.8 demonstrates how this function would generate the animation frame message, given an initial list with 6 items and some visual annotations (such as colours and vertical offsets, which applied as needed by demos). This frame is passed from the VM module to the main module, consumed (as per figure 3.6), and then passed to the Net module, which then serializes it and sends it onto the server via the websocket, as represented in figure 3.7. The Viz module performs the animation described by the frame, and schedules the callback to be invoked once the animation is completed, which is picked up by *JS-Interpreter* and used to resume the virtual machine's execution.

Note: In step 2 in figure 3.8, a *deep copy* of the AV object's internal state is produced. This is an object which equal by *value*, but not by *reference*. If this deep copy was not performed, the recorded state in every frame in the frame history would refer to the same object! As we want to keep a record of the data structure's state, each frame contains a deep copy of the original working data used by the VM as it was when the frame was generated. JavaScript doesn't have an explicit deep-copy function, however as our internal representation is non-cyclical and contains only simple objects (no functions), we can achieve the same thing by serialising the AV object's internal data representation to JSON and deserialising it back again, and using this for `newState`.

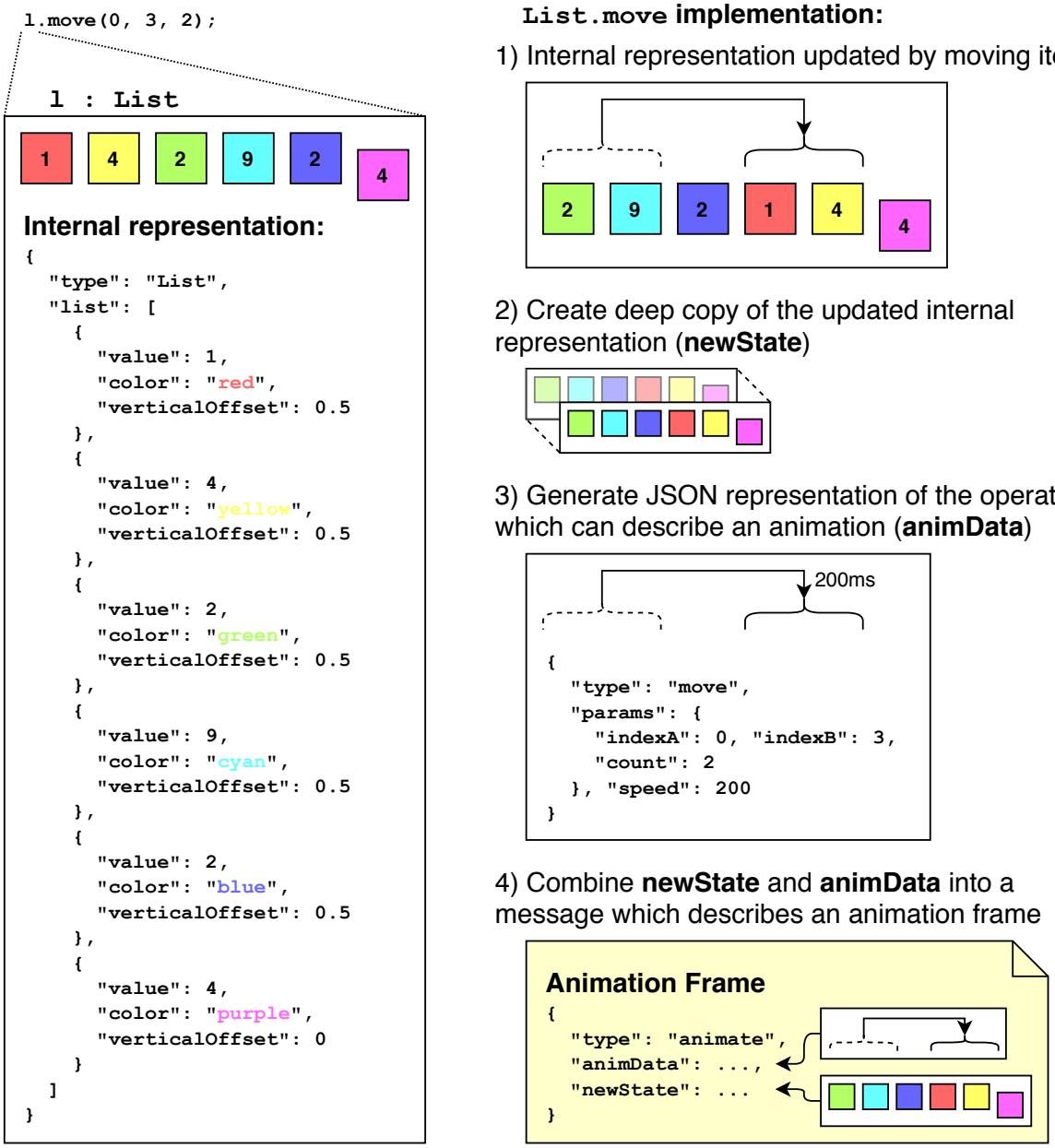


Figure 3.8:
An overview of how `List.move` generates an animation frame.

3.5.2 Tracking Line Numbers and Step Frames

In order to allow us to rewind the demo and then step back through it, we need to track the line numbers that the interpreter hits. Otherwise, if we rewound the demo, we wouldn't be able to step through it line-by-line, only animation-by-animation. Usefully, the VM's AST node that we use in the pausing logic also exposes the text range in the source code which it was parsed from. Every time the VM's interpreter is stepped, it checks if the current AST node meets the three criteria above. If it does, it records the start and end character index in the source code of that node in a message with the type `step`. These are sent out of the VM as frames, so that each step in the evaluation—along with its position in the code—is recorded as part of the demo's timeline. However, an algorithm will likely step through a lot of lines as part of its evaluation. In fact, if we were to send this data out of the VM naïvely step-by-step, there would be an inordinate number of `step` frames produced and sent over the network.

To mitigate this, the VM does not send out a `step` frame every time it reaches a new line. Rather, it batches them up, and only sends them out when it was going to send another message type anyway; for example, it will carry on recording the step data until it sends out an animation, at which point it sends out all of the `step` messages before the animation frame. Additionally, rather than sending out potentially hundreds of step frames at once, it merges them all into a single `stepData` frame, which contains a single array consisting of alternating start and end indexes of an arbitrary number of steps in the demo's source code (so that it will always have an even array length). This is a much more compact representation. This `stepData` frame is sent over the network as a single frame, and only expanded by the Main module when it is inserting it into the frame history.

3.5.3 Input Generation

One of the requirements is a Creator's ability to randomly generate input data structures for algorithm demonstrations. The API available to demos in *algo.js* provides some methods to assist in doing this. For Lists, some methods to generate arrays are included, including a random array function, an equivalent to Python's `xrange` function, and a shuffle function. Generating random Graph inputs isn't as easy, however, as there a lot more parameters to specify: how many vertices? How many edges? What degree of connectivity should there be? Also, a graph in the mathematical sense—that is, a set of vertices and a set of edges with numerical weights connecting those vertices—contains no information about how to visualise it. A graph generated with random edge weights may not necessarily be Euclidean; such a graph will not be intuitively visualisable on a flat surface like our AV tool's display, in such a way that the visual distance between vertices on screen corresponds to the weight of an edge between them.

There are two things to consider when generating nice-looking Euclidean graphs: how to spatially distribute vertices, and how to connect those vertices with edges. These amount to two separate problems. A naïve way to generate a set of vertices would be to randomly place a set of points in the plane, with a uniform distribution for the X and Y values in a given range; you would then assign edge weightings to pairs of vertices based on their Euclidean distance. This technically works. However, the generated graphs often do not look pretty, and may use the available screen space ineffectively. The random positioning means that the spatial distribution of vertices can be very uneven, with some vertices being placed in almost the same position as another, and others being very far away from the rest. This can make parts the AV hard to read, especially when text labels and edge lines start overlapping. While the actual algorithms work fine on randomly-generated graphs, the whole point of the project is to effectively visualise the algorithm, which is more difficult if you can't make sense of the data structure from its visualisation.

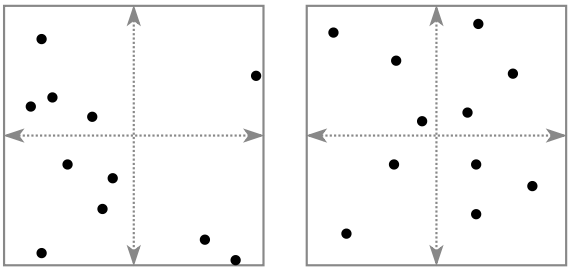


Figure 3.9:
Random point distribution on the left. Some areas have a high density of points, which can make it awkward for Participants to visually consume the annotated AV; other areas are essentially devoid of vertices, wasting screen space. A more even distribution is shown on the right.

Requiring the Creator to solve this problem for every created demo is a waste of their time. Generating random graphs manually is time-consuming, as would be writing your demo to do this automatically. This problem was identified mid-way through the project—this is elaborated upon in chapter 4. An ideal solution would be a two-step generation function in the API: a range of functions to generate a set of vertices in different ways, such as using different spatial distributions; and a range of functions to connect an input set of vertices, by adding edges in different ways. This would allow the user to generate different types of input graphs for their demo by using different combinations of functions—a function to add many edges to create a highly-connected graph, a function to add edges in a way that produces a tree, and so on—allowing a Creator to show how an algorithm's performance fares, and its output differs, on different types of graphs.

Adding this level of functionality is outside of the scope of this project. However, to demonstrate the point and partially achieve the goal, a test method is available to Graph-based algorithms, which can generate random graphs which are good enough to demonstrate algorithms such as Dijkstra's algorithm and Prim's minimum spanning tree algorithm. It isn't based on any rigorous mathematical technique, and some internal parameters have been tuned empirically, but it suffices. It accepts two parameters: $N_{vertices}$, the number of vertices; and N_{edges}, which is a measure of how connected the graph is. It works as follows:

1. Initialize an empty set of vertices $\mathbb{V}$.
2. Generate two values p, ϑ; where $0 \leq p < 1$ and $0 \leq \vartheta < 2\pi$, picked randomly using a uniform distribution with those ranges.
3. Calculate a radius value $r = (1 - p^3)\sqrt{5|\mathbb{V}| + 2}$. Convert (r, ϑ) to a Cartesian (x, y) and use this as a tentative vertex position V.
4. If the Euclidean distance between (x, y) and any other point $(x', y') \in \mathbb{V}$ is less than 2, go back to step 2. Otherwise, add V to $\mathbb{V}$.
5. If $|\mathbb{V}| < N_{vertices}$ then go back to step 2.
6. Initialize an empty set of edges $\mathbb{E}$.
7. For each vertex $V \in \mathbb{V}$:
    a. Create a list L containing every *other* vertex in $\mathbb{V}$ in increasing order of their Euclidean distance to V.
    b. Initialize a counter $e = N_{edges}$.
    c. Remove a vertex U from the start of L. Add (U, V) to $\mathbb{E}$ if it isn't already in it
    d. Decrement e; if $e > 0$ then go back to step 7c.
8. Return the list defined by the vertices $\mathbb{V}$ and the edges $\mathbb{E}$.

This generates a random graph from the origin outwards, where the minimum distance between any two vertices is 2, and whose edges are added using the nearest-neighbour method. Doing it this way (with polar co-ordinates) is much quicker than just randomly generating some Cartesian (x, y) in step 3, as doing so can make it incredibly slow to find co-ordinates satisfying Step 4's condition if the initial random positions of vertices is unfavourable. This algorithm generates vertices along (or just inside the perimeter of) a set of concentric circles which grow as more vertices are added. Picking candidate vertices in this manner ensures they are distributed with a roughly uniform density.

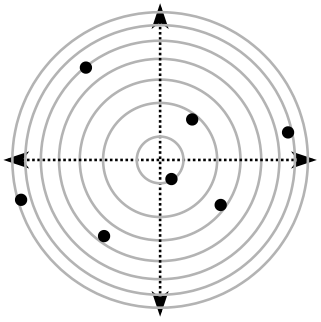


Figure 3.10:
Notice each vertex is on, or near, a circle's perimeter.

In step 3 of the algorithm, the $(1-p^3)$ factor ensures that candidate vertices are generated randomly, but more often so near the perimeter of the circle; p is uniformly chosen between 0 and 1, so $(1 - p^3)$ is more likely to be closer to 1. The $\sqrt{5|\mathbb{V}| + 2}$ factor ensures the circle grows as more vertices are added; the square-root is so that the area occupied by the circle grows linearly with

the number of vertices, meaning the density of vertices isn't all clustered around the origin as they would be if the radius grew as a linear function of $|\mathbb{V}|$.

3.5.4 Modifications to JS-Interpreter

JS-Interpreter suits the requirements of the project very well. However, in order to fix some usability issues, some changes were made to the library's source code. One such change allows a Creator to access elements in a `List` object using the array access syntax (`arr[i]`). Overriding the `[]` operator isn't possible in standard JavaScript. However, in order to allow Creators to implement algorithms as idiomatically as possible using a List in place of a standard JavaScript array, the interpreter has been modified to add a new internal flag to pseudo-objects, permitting object prototypes to define a `get()` function which will be called if you attempt to access such an object like an array.

Another change is to allow us to mark objects as "system objects" with another flag. The `Demo` object contains several helper methods, such as `Demo.visualize()`. However, an unhelpful error message is produced if a Creator misspells a function name (*'`undefined` is not a function'*). This is a caveat of JavaScript's type system rather than *JS-Interpreter*, and may make sense in some situations, but is not useful for the user. It also doesn't really make sense to change the `Demo` object in any way, so if a user attempts to edit any of this object's attributes, then an error is raised; and any attempts to access nonexistent attributes will also raise an error rather than returning `undefined`.

3.6 Visualisation module (Viz)

This module (`algo.viz.js`) implements the rendering and animation system for the AV. As mentioned previously, *two.js* is used to draw to a `<canvas>` using a scene graph approach. As described in the client overview section, the module for each AV data structure is split between the VM hooks which manipulate the data structure and generate animation frames (as described in the previous section), and the visualisation hooks which draw the data structure based on the internal representation.

There are two components drawn to the canvas: the AV itself, and the key; both have a JSON representation, where the format of the former is the same format as the VM's internal representation described in the previous section. The key can be used by Creators to describe what different colours applied to the data structure represent in terms of the algorithm; for example, the stock Quicksort demo uses the colour green to represent the pivot.

The appearance of both are fully described by a combination of their internal JSON representation, and the pan/zoom parameters. How are these drawn? This is where the scene graph comes in. An internal object called the *animation state* is used by the Viz module for two purposes: to associate parts of the data structure to the elements of the scene graph which represent them, and to track the progress of an ongoing animation of the data structure. The specific steps of a process for rendering a data structure naturally depends on which data structure it is. The Viz module calls into the data structure–specific module to do one of two things:

- *Initialise* the visualisation. This removes everything in the existing scene graph and recreates the AV from scratch. This is used when the demo algorithm wants to visualise a new data structure, such as at the start of a demo.
- *Animate* the visualisation. This keeps everything in the scene graph intact except the part of the data structure being modified, which may be moved, added or deleted.

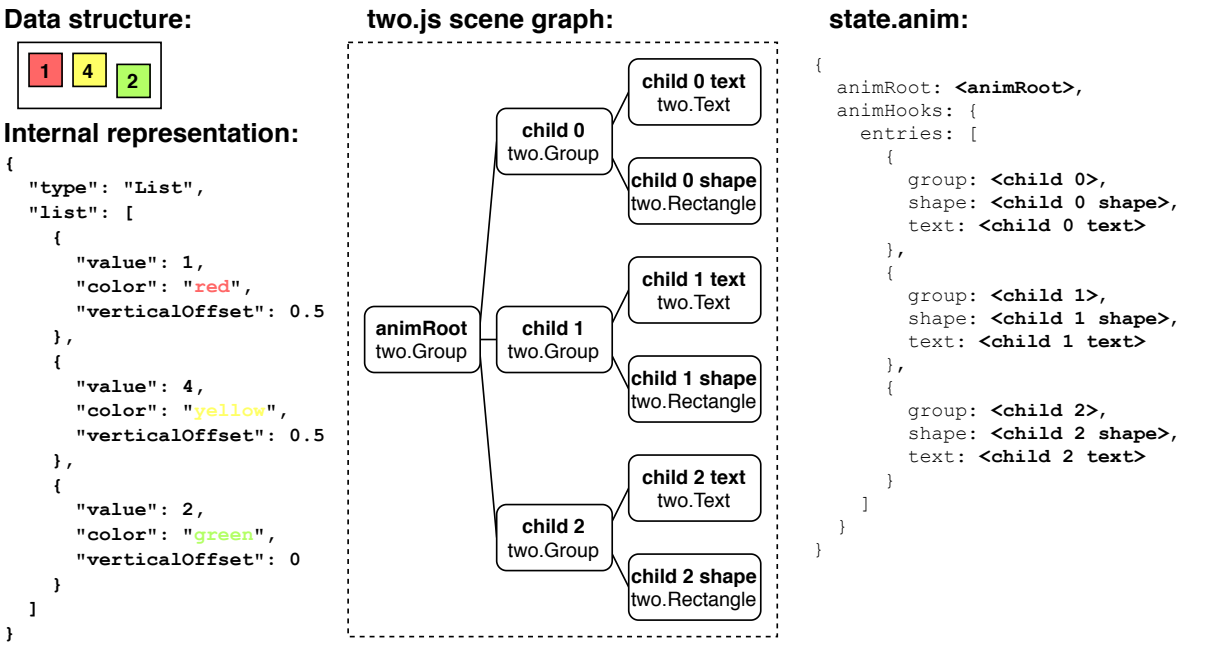


Figure 3.11:
How the data structure relates to the scene graph and the data in `state.anim`.

A few bits of terminology are needed here:

- There is an internal object called `state` in this module. This contains a `viz` object, which is the internal representation of the data structure (in the same format as used by the VM, as mentioned in the last section). `state` also contains `anim`, which is the *animation state* object.
- A *group* is an element of the scene graph which doesn't itself draw anything to the canvas, but can contain child elements which are positioned relative to the group object in terms of translation, rotation and scale. The *animation root*, stored in `anim.animRoot`, is the parent group for everything in the data structure. The pan and zoom parameters are applied to the root, which scales and translates all child elements in the AV.
- A *hook* is a reference to an element of the scene graph representing part of the data structure. These are stored in `anim.animHooks`, along with some scratch variables used for storing the distance offsets and spacing between items.
- An *AV handler* (AVH) is a data structure–specific function in a submodule, which is called into by the Viz module in order to either initialise or animate the AV for a data structure. An AVH has direct access to the scene graph in order to update the rendering, and is invoked with the data structure object's internal representation and (for animations) the animation data.

Figure 3.11 shows how references to relevant parts of the scene graph are stored as hooks in `state.anim` for a List AV data structure. *(The bold values in angle brackets are references to the scene graph objects with types defined by* `two.js`*.)* The initialisation AVH for a List uses its internal representation to create the scene graph objects for each element in the list: a `two.Text` object is created with the element's value, followed by a `two.Rectangle` of the appropriate colour and width. These are then added to a `two.Group` so they can be moved around together, which itself is then added to `animRoot`. The three scene objects created for each list element are added to an object, which is added to an array in `animHooks`. This means the List's animation AVHs can access (for example) the rectangle for element 2 in the list with `animHooks.entries[2].shape`, if that element were to change colour.

This scene graph approach means that the data structure's visual layout only needs to be calculated when it changes, not each frame. This might not make a huge difference on desktop devices with powerful CPUs, but this frees up important processing time for the VM (and the rest of the device) on mobile devices. Some scratch values for calculating the spacing of elements are also stored—both in `animHooks` and in the per-element objects, such as the calculated text width of an element—which means the width of each bit of text on screen need only be calculated once overall. **Note**: the colour key (at the top left of the AV) is independent of the data structure and its state is stored separately (as the *demo state*, which also keeps track of the speed at to run animations); this also has its own corresponding set of hooks, but is otherwise drawn in much the same way with its own dedicated AVH in the Viz module.

3.6.1 Animations

An animation represents any change to the visualisation of a data structure, whether the change take place immediately (such as a colour change) or with a smooth animation (such as motion). The Viz module calls into the AVHs to process the individual animations, but for those AV updates which require this smooth motion, it also exposes an animation scheduling system which an AVH can call back into. When the Viz module is invoked to render an animation, the animation data is passed in along with a *callback*. The Viz module invokes this callback once the animation is complete. Step 2 in figure 3.7 shows this: the callback in this instance is from the VM module, and *JS-Interpreter* will resume the demo once this callback is invoked when the animation completes.

For the animations which do not require smooth motion, this is simple. The AVH updates the scene graph with the change, invokes the callback immediately and then returns. An example would be in a Quicksort demo, where the pivot's colour is changed to green. This animation isn't smoothed, meaning nothing has to be scheduled and the Viz module can immediately invoke the callback. For animations which do need to be smoothed, the Viz module's animation scheduling system makes it really straightforward. The AVH is invoked as usual, and may calculate things such as the spacing and item positioning for the list post-animation, which will be stored in `state.anim`. An animation is scheduled with three parameters: an *update* function, a *finish* function, and a duration; these are passed to a `Viz.startAnimation(updateFunc, finishFunc, duration)` function. When this function is called, a loop is started which calls `updateFunc` for every frame. **Note**: the update loop is provided by *two.js* itself, and handles it so that `updateFunc` will be called at the display's refresh rate where possible.

`updateFunc` is called with one parameter t. This is the amount of time passed since starting the animation as a fraction of the total time, between 0 and 1. For example, if `startAnimation` is called with a duration of 200ms, then 100ms later `updateFunc` will be called with $t = 0.5$. Assuming the function used for `updateFunc` is a closure inside the AVH which started the animation—which is the expected use case for `startAnimation`—then this can be used to interpolate the position of a scene graph element on each frame between the initial and final position.

`finishFunc` is called with no parameters, and is called when the animation time elapses. This is used to do any final processing, such as invoking the callback function. Alongside calling `finishFunc`, the animation system also stops the render loop as the AV is now static until the next animation, and also calls `updateFunc` one last time with a value of $t = 1$; this is to avoid the final positions of objects being slightly offset if the last call to `updateFunc` by the update loop is made slightly before the animation's running time elapses (ie. `updateFunc` is called with a value slightly less than 1).

As an example, we're animating the movement of an element of a list from one position to another. This may result in every other item in the list being shifted along, too. We already

know the existing X co-ordinate of each element on screen; assume the initialisation AVH stored the X co-ordinate of element n in `animHooks.entries[n].x`. Our animation AVH calculates what the new X co-ordinate of each list element will be *after* the animation, and stores this in `animHooks.entries[n].newX`, and then calls `startAnimation`, like so (pseudo-JS):

```
1   function listMove(animData, ..., callback) { // AVH
2     ... // initial setup
3     Viz.startAnimation(function(t) {
4       // update
5       for(var i = 0; i < animHooks.entries.length; i++) {
6         // move group for this element to the interpolated new position
7         animHooks.entries[i].group.setX(
8           animHooks.entries[i].x +
9             t * (animHooks.entries[i].newX - animHooks.entries[i].x)
10        );
11      }
12    }, function() {
13      // upon finish
14      for(var i = 0; i < animHooks.entries.length; i++) {
15        animHooks.entries[i].x = animHooks.entries[i].newX;
16      }
17
18      callback();
19    }, animData.duration);
20  }
```

On lines 8–9, the X co-ordinate of each element in the list is interpolated between it's pre- and post-animation value of X. On line 15, in our `finishFunc` implementation, the stored X co-ordinate for each element is set to the updated value, and on line 18, the callback is called, which will return control flow to whichever module passed the animation to the Viz module. The Viz module stores the `updateFunc` and `finishFunc` functions, as well as the duration and the *current* elapsed time of the animation, in `state.anim`; together these are used to call `updateFunc` with the appropriate t value on each frame update, and to call `finishFunc` when the duration has elapsed.

3.6.2 Reversed Animations and Smoothing

The Viz module also has functionality to run animations in reverse. This functionality has been designed so that only minimal changes have to be made to the AVHs themselves. If an animation is passed to the Viz module with a *negative* duration $-d$, this is interpreted as a request to run an animation with duration of d in reverse. The implementation of this is actually very simple: rather than calling `updateFunc` with an increasing value of t (ie. $t = 0$ at the start of the animation and $t = 1$ at the end), it is called with a *decreasing* value of t. This has the effect of visually running the animation in reverse. This is used for rewinding a demo—see the Main module for more information on this.

Regardless of whether the animation is ran forwards or reversed, `updateFunc` is called with a value of t which changes *linearly* with respect to elapsed time. Assuming an AV element moves

along a straight path, simply interpolating a position along this path results in a somewhat abrupt movement; while the animation is running, it will move with a constant "speed" on the screen. However, at the very start and end of the animation, it starts or stops moving instantly. Initial testing of the animation system found this to look a little jarring. To get around this, the Viz module exposes a speed function for AVHs to interpolate with:

$$\mathbf{st}(t) = \frac{1}{2}\sin\left[\pi\left(t - \frac{1}{2}\right)\right] + \frac{1}{2}$$

This function has a low rate of change around $t = 0$ and $t = 1$, and the greatest rate of change around $t = 0.5$ (at the midpoint of the animation). If $\mathbf{st}(t)$ is used to interpolate the positions of objects, the gentle "start" and "stop" produces an animation which (in my opinion) is easier to visually track on-screen, especially when quicker animations are used—and also simply looks more visually appealing.

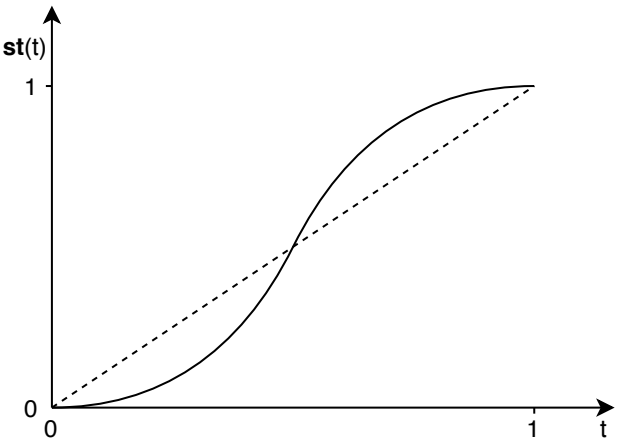


Figure 3.12:
The speed function (unsmoothed speed as dotted line).

3.6.3 Modifications to two.js

Like *JS-Interpreter*, *two.js* required a small modification for this project. The change to this library isn't as extensive as those for the VM module. The text rendering implementation in the stock *algo.js* version is slightly bugged: text with a fill (primary text colour) *and* stroke (outline colour) enabled will render incorrectly, with the outline being rendered over the fill. This looks OK with larger text; however, as the text stroke is simply a path drawn around each glyph's outline, it will "bleed into" the text with a smaller font, making it hard to see and slightly unpleasant to read. The modification simply fixes this so the outline is rendered *under* the fill.

3.7 Networking module (Net)

This module does the job of establishing the websocket connection, and sending and receiving data through it. JavaScript's websocket API is event-based, rather than the socket API style used by most non–web based languages (which use blocking read calls). A connection request is made by simply initialising the object. Four events are fired by the browser when the state of the websocket changes:

- `onopen`, when the websocket connection is established, some time after the object is initialised.
- `onmessage`, when data is received from the server.
- `onclose`, when the connection closes for any reason.
- `onerror`, directly before `onclose` if the connection is closed unexpectedly, such as a dropped Wi-Fi or cellular data connection.

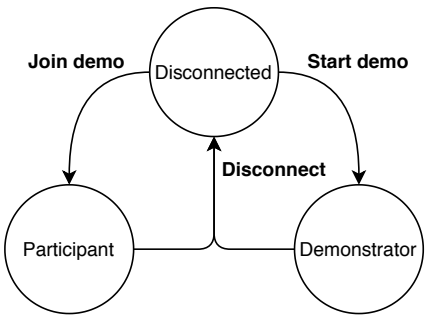


Figure 3.13:
The module's state machine.

Figure 3.13 shows the state machine which this module exposes. The module hides the workings of the websocket and exposes the functionality with 4 methods: `initDemo` to start a demo, `initParticipant` to join a demo, `send` to send data when connected, and `disconnect` to manually disconnect (ie. when the user wishes to end or leave a demo).

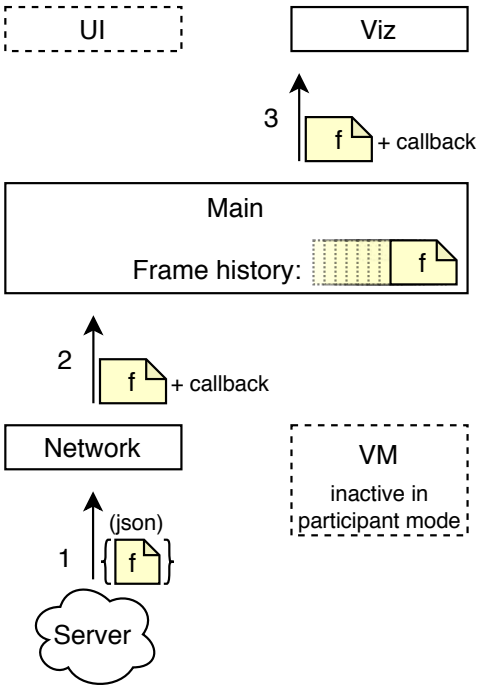


Figure 3.14:
How messages from the network, such as animation frames, flow through the system modules. Compare to figure 3.7.

`initDemo` accepts two parameters: a success callback function, which is invoked—with the newly created demo ID, as reported back by the server, as a parameter—when the connection is established and the demo is created; and an error callback, which is invoked when something goes wrong when connecting to the server. The latter just captures the `onerror` event data from the websocket API in a cleaner way (in one event, rather than two). The `initParticipant` method accepts three parameters: a demo code, which identifies the demo to join (and is sent to the server upon connection as part of the handshake—see figure 3.1); a success callback function, which is used as in `initDemo` but with no parameter; and an error callback function, which is also the same as `initDemo`, but is called in the additional case where the demo code provided does not correspond to a demo being hosted by the server.

When messages are received from the server, they are sent to the Main module to be handled in the exact same way as they would be if they were produced locally by the VM—see figure 3.14—which means including a callback function, which is used to notify the Net module that the message has been fully dealt with. This may not be an instant process if the message is an animation frame, as discussed in the previous section on the Viz module. Therefore, every time a message is received from the network, we need to wait for it to be fully consumed—that is, wait for the callback to be invoked—before we begin processing the next message.

When we're consuming frames produced locally by the virtual machine, this isn't an issue. Data structure operations which trigger animations are implemented using async functions, and *JS-Interpreter* pauses the VM until its callback is invoked by the Viz module when the animation terminates—meaning we can just consume messages from the VM one at a time. However, there is no guarantee that we won't receive a message over the network while the previous message is still being processed. Correctly buffering the messages received via the websocket is the main duty of the Net module.

Figure 3.15 shows how the event handler for `onmessage` stops messages from being processed concurrently. If a message is already being handled, the handler just queues the new message and terminates. Once a message has been handled, the Net module checks if any others have been received in the meantime, dequeuing and processing them if this is the case.

The websocket API deals with messages in string form by default. This means we do not need to deal with byte buffering when sending or receiving; we can just

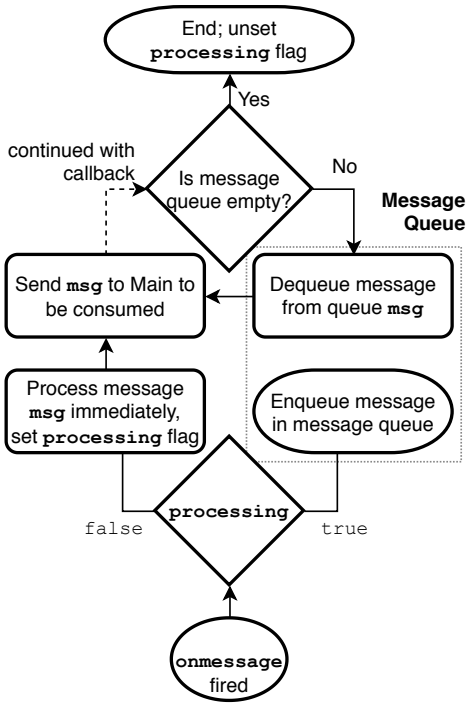


Figure 3.15:
The event handler for `onmessage`.

serialise our messages to JSON and send them on, so that they can be received by the server.

3.7.1 Ping Timer

In some situations, a websocket will be automatically closed if it is not used for a period of time. This may be done by the client, as is the default behaviour on some mobile devices when the phone is locked or the browser app loses focus, which means the connection could drop if (for example) a student's attention shifts momentarily from their phone. It may also be closed by the server. This project was hosted for testing purposes on *Heroku*, an online web hosting platform; to support HTTPS, Heroku acts as a gateway to the actual *algo.js* server so that all HTTP connections are "tunnelled" through it, including the websocket connection which is just a layer over HTTP. To get around this, whenever the Net module establishes a connection, it sets up a recurring call to a ping function using `setInterval`. This simply sends a ping packet to the server, containing some dummy data. When the server receives this, it responds with a dummy pong message. This prevents the websocket from being closed.

3.7.2 Handling of Unstable Connections

One of the requirements of the project was the ability to deal with unstable connections, such as crowded Wi-Fi networks or poor cellular data signals. Unfortunately, this wasn't able to be implemented in the timeframe for this project. This is discussed in further detail in section 4.1.

3.8 Main module

The main module is the entry point of the tool, and essentially orchestrates the other modules. Besides initialising the other modules upon loading the tool, it serves three main purposes:

- Consume messages and perform the appropriate action.
- Track the *state history*, allowing the user to skip forward and back through the demo.
- Acts as a state machine to track which demo actions are currently available to the user.

For each state, the actions (and labels) shown on the buttons above the code editor correspond to the state transitions that can be made in that state:

State	Action 1	Action 2	Action 3	Action 4
Demonstrator Stopped	**Run** Running	**Unstep**	**Step** Paused	**Stop** disabled
Demonstrator Running	**Pause** Paused	**Unstep**	**Step**	**Stop** Stopped
Demonstrator Paused	**Resume** Running	**Unstep** Rewound	**Step** Paused	**Stop** Stopped
Demonstrator Rewound	**Resume**	**Unstep** Rewound	**Step** Rewound Paused	**Stop** Stopped
Participant Watching	**Pause** Rewound	**Unstep** Rewound	**Step** Rewound	**Watch**
Participant Rewound	**Resume**	**Unstep** Rewound	**Step** Rewound	**Rejoin** Watching

Table 3.1:
The actions and states available. Bold text represents the label shown on the button, with the possible resultant states shown beneath that (no resultant state indicates the button is disabled).

Distinguishing the program states like this allow us to determine exactly what actions are possible at any given time, as well as prevent certain user interactions when they wouldn't make sense; for example, the code editor is only writable in the Demonstrator's Stopped state. It doesn't make sense to edit the code when the demo is running, or if you're a Participant. The first three Demonstrator states correspond to the states of the VM, and transitions between these correspond to calling the `play()`, `pause()`, and `stop()` methods in the VM module. Both *rewound* states are used when seeking through the frame history.

3.8.1 Frame Handling, History and Rewinding

The logic in figure 3.6 is a simplified representation of how frames are consumed and processed. This diagram is accurate for non-frame messages. For frames, however, the logic is slightly different. Before this is covered, the logic for the frame history must be described.

The frame history is just an array of frames. This is of variable size, like all JavaScript arrays. The main module tracks a variable `currentFrame`, which is the index of the frame we're currently "at" in the demo's timeline. When we want to move one frame forward, this is simple: `currentFrame` (represented as c in figure 3.16) is incremented, and the handler for f's frame type executed. For example, if f is an animation, it is sent to the Viz module to be played.

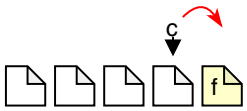


Figure 3.16:
Stepping forward in a demo.

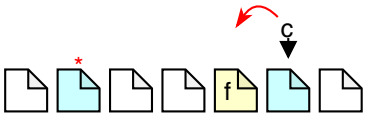


Figure 3.17:
Stepping backward in a demo.

Going back a frame isn't so simple. Stepping backward means we want to *undo* the current frame's action. For example, if the current frame is of type `step`, that means it updated the execution point shown in the code editor, and so to undo it we want to change the execution point to whatever it previously was. If the previous frame was also a `step`, that's OK—we can just execute that frame. However, if the current frame is of a different type to the previous one, this won't work. For example, if the previous frame was `setViz`, then executing this wouldn't "undo" the `step` frame at all.

To step back a frame, what the tool actually does is decrement the `currentFrame` pointer, and then *carry on searching backward* until we encounter a frame of the same type to the one we just "jumped off", and evaluates *that* (shown with a red asterisk in figure 3.17). Stepping backward must also consider animations. As mentioned in the section for the Viz module, animations can be ran in reverse. Look at figure 3.18. If the current frame is f_2 (so we're in we're in $state_2$), and we step back a frame to f_1, we expect to see $anim_2$ play backward and end up in $state_1$. However, simply evaluating f_1 would result in $anim_1$ play forward and ending up in $state_1$—but we don't want to play f_1's animation. The Main module checks whether the current frame is an animation when stepping backward and, if it is, will play its *own* animation in reverse (as described in the section on the Viz module) and then set the AV state to that of the previous `animate` or `setViz` frame, *without* playing its animation.

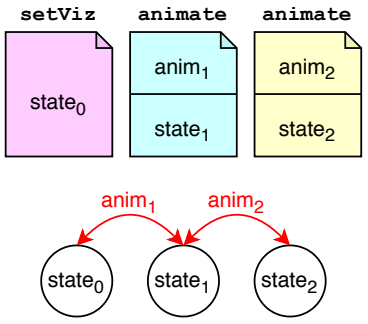


Figure 3.18:
How `animate` frames relate to AV states and state changes.

That describes how to move forward and backward in the frame history. To see how frames are added, two more variables are introduced: `demoFrame` and `detached`. `demoFrame` represents the current state of the *demo controller*. Exactly what/who the demo controller is depends on whether we're a Demonstrator or Participant. If we're a Demonstrator, the demo controller is

the VM. We're consuming frames directly from the VM, and `demoFrame` always points at the last frame it produced—which means it always points at the end of the frame history.

If we're a Participant, however, the demo controller is the Demonstrator. In this case, `demoFrame` always points at whichever frame the demonstrator is looking at on their own screen, and **`demoFrame` is equal to `currentFrame` on the Demonstrator's device**, as shown in figure 3.20. If the demonstrator is currently playing the demo, then `demoFrame` will also point at the most recent frame too. However, if the demonstator has stepped backward through the demo on their own device, then `demoFrame` will point to some frame in the middle of the demo. Whenever the Demonstrator steps forward or backward in the demo, the tool sends a message of type `jump` informing the Participants of where the Demonstrator is "at" in the demo's timeline—ie. where to 'jump' to in the frame history. *(This means a `jump` message can trigger an animation for an existing frame.)*

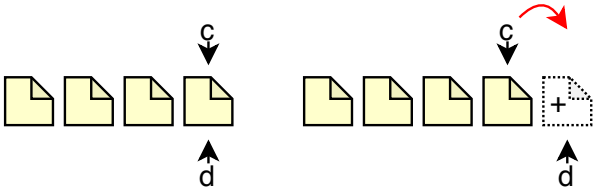


Figure 3.19:
When the user is attached to the demo, the current frame tracks the demo frame. When `demoFrame` changes—which occurs when the VM produces an animation frame, or a `jump` message is received—`currentFrame` will 'follow' it, and render the new animation.

If the `detached` flag is set, then `currentFrame` is allowed to move independently of `demoFrame`. As a Participant, this also means that the `jump` messages are ignored. `detached` is true whenever we start to "unstep" the demo. This is an important point to note: **the VM controls the demonstrator's display in the same way the demonstrator controls the participants' displays.** Designing the system this way means that parts of the implementation for Demonstrator and Participant mode are largely the same, which greatly simplified development of the tool—the only difference is whether frames are consumed from the VM or from the network.

The "Rejoin" action shown in table 3.1 simply sets `detached` to `false` and `currentFrame` equal to `demoFrame`, so that the AV will now continue to mirror the demo controller; this allows a participant to "re-synchronise" their AV to the Demonstrator's display once they are done back-tracking. Both "rewound" states shown in the table are almost identical in implementation, and only differ in the states they can transition into.

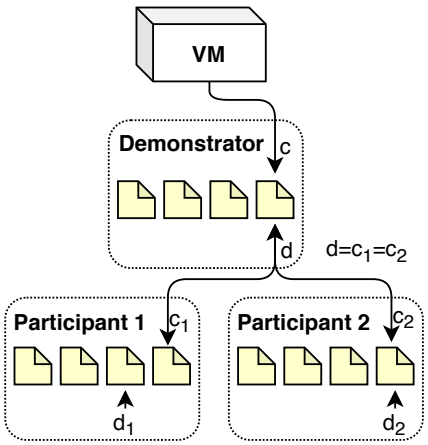


Figure 3.20:
The Participants' AV is controlled by the Demonstrator, and the Demonstrator's AV is controlled by the VM. Both can "detach" from their controller, like Participant 1 has in this diagram.

This system design also means that the "Step" button does one of two different things: step the VM, or replay a previous frame. Despite this, the design results in the two behaving the same, so that stepping through a rewound demo's frames is identical to stepping through the VM.

Chapter 4

Evaluation

The literature review for this project identified a set of general requirements described by prior attempts at creating effective AV tools for teaching. These were then refined into more formalised technical requirements which dictated the design and implementation of the tool. To evaluate the project, this chapter will recap those requirements and see how well *algo.js* meets them, from both a technical and a pedagogical point of view.

4.1 Technical

The functional requirements will be discussed first, followed by the non-functional requirements. **Note**: The functional requirements are identified with a letter followed by a number; the letter being D, P or C for Demonstrator, Participant or Creator respectively, and the number being the requirement's position in the corresponding list in sections 2.2.1–2.2.3.

The MoSCoW system ended up being quite useful to separate requirements which were necessary to meet the baseline goals from the requirements which would enhance the tool but otherwise not offer key new functionality. As mentioned in section 2.2, the choice between *should-* and *could-have* was a little arbitrary, but those relegated to *could-have* were those for which implementing them would require disproportionately more effort for the improvement they sought. By working first on the *must-haves*, I managed to reach a minimum viable product quite early on, which was enough to test the tool and explore how it behaved in practice—and, from there, prioritise the non-critical requirements based on:

- **Which features the tool obviously lacked.** As an example, after all the main functional requirements were done and I could start implementing and demoing algorithms, it quickly became apparent that I was writing the same code over and over to auto-generate things like random lists to demonstrate sorting algorithms, as described in 3.5.3. Additionally, due to the VM's inherent speed deficit, this was taking a noticeably long time to run at the start of the demo—this was particularly the case for generating nice-looking random graphs. I had already identified input generation as requirement C10, but at the time I hadn't considered that this may be slow when done from within the VM. For this reason, C10 was prioritised next despite being initially categorised as a *would-have* requirement.
- **Which features would be easiest to implement next.** At the start of the project, I had no experience using *JS-Interpreter*, *two.js* or the *Ace* editor. However, after starting the priority requirements of the project, I worked out that the design of these libraries lended themselves to implementing some of the other requirements more easily. As an example, Ace allows custom highlighted regions to be specified in the editor with a character range,

and *JS-Interpreter* exposes the character range of the interpreter's current position in the AST quite transparently. This made it requirements D10 and C13 extremely easy to fulfil by basically just wiring the range info from the VM to the editor with `step` frames.

As the majority of the functional requirements were met, this part discusses those which weren't.

- **D9** (should): demo "anchor points"/bookmarks. Internally, the system supports jumping to arbitrary points in the frame history just fine. However, adding the UI for this would have taken too long. I couldn't think of a way to support jumping to specific parts of an algorithm in a demo in a sane way, as most "parts" of an algorithm run many times in the course of its execution.
- **D11** (could): quiz questions, which the Demonstrator can start. It would have been nice, and it wouldn't have taken too long to build it upon the existing system. However, as it didn't really add anything to the tool which hasn't been done already, this requirement was skipped to fulfil other requirements.
- **D12** (could): edit a running demo. I didn't know how easy this would be when I was planning the project. *JS-Interpreter* turned out to be powerful, but it doesn't support this out of the box, and as it was a slightly optimistic requirement to begin with, it was skipped.
- **P6** (should): "fork" a demo instance with the same code. The initial idea was to be able to fork a *running* demo. This is technically possible, as you can serialise the VM's state and transmit it over the network—but it would have complicated the client/server model used. Currently, participant clients send no further data once connected. This would have required the client to request the VM state from the demonstrator, complicating the netcode. The "weak" interpretation of this requirement—that is, to just start a demo with the same code—can be accomplished with the user's system clipboard.
- **P10** (should): change demo speed. The Creator can change the demo's animation speed with `Demo.setSpeed`. However, you cannot override this in the UI. It wouldn't be impossible, but again, adding more features like this would require extra thought put into the UI's layout so that mobile users aren't inconvenienced by a crowded display.
- **P12** (could): replay a rewound demo. The user can step backward and then step back forward again, but the user cannot step backward and then click "play" to re-run through automatically. This could be done, but I just didn't have the time—it would require further additions to the program's state machine (see table 3.1) and, as the same process can technically be performed (albeit slower) by repeatedly stepping forward, this was deprioritised in favour of other tasks.
- **C8** (should): pan/zoom from code. This would allow the demo to autofocus on points of interest. Pan/zoom is implemented, and participant's displays will track the demonstrator, but doing it from code would require a rework of the Viz module to map a data structure location (e.g. an index in a list) to screen co-ordinates to pan to, and other considerations such as whether to prioritise the algorithm's pan/zoom actions over the user's, which made this requirement too much extra work.
- **C9** (could): `Tree` data structure. There was no technical hurdle to adding these—they were just deprioritised after Lists and Graphs were added, and didn't make the cut. Trees (and other structures) could be added as part of future development without touching any existing code, as a submodule which registers itself in the VM.
- **C11** (could): save custom algorithm demos for later use. This would be fairly simple: just add a `HTTP PUT` endpoint which accepted the code, along with a demo title and an author name, and write it to the file system. Demos aren't hard coded into the server so this wouldn't require too much work on the serverside, and the clientside UI wouldn't be complex. However, some way of moderating the system (to prevent offensive or garbage uploads) would be needed in practice which is a problem unto itself.

Overall, the majority of requirements have been satisfied, including all of the *must-have* requirements. Most of the others wouldn't be technically hard to do: as initially planned in section 2.2, the architecture of the tool is extensible enough so that a lot of these could be implemented without extensive re-writing of existing components. The rest were perhaps just too optimistic given the limited time available. Most of the non-functional requirements are met. The system for adding new AV data structures is extensible. The number of UI interactions to do basic tasks is minimised, and is intuitive on mobile. It also loads quickly on mobile: Mozilla Firefox supports emulation of various connection types in the developer tools. The page loads in less than two seconds over an emulated 4G connection, with the cache disabled; all loaded resources total to less than 800 kB. The bottleneck in the loading process is fetching all the page's scripts, which means the tool loads rapidly when these are cached after the first load.

The requirement for dealing with unstable network connections proved more difficult. As it stands, the tool works very well on a Wi-Fi connection, and well enough over 4G; once the web app is actually loaded, not a lot of bandwidth is required to run or watch a demo. The stock Quicksort demo takes roughly 40 seconds to complete, transmitting around 200 kB of JSON data over the websocket; the bandwidth required to run a demo is in the order of tens of kilobytes per second. Part of the requirement, however, was the ability to cope with dropped connections. If the network connection temporaraily failed, then rather than leaving the demo, outbound messages would just be queued until the connection can be re-established—this would be indicated in the UI as shown in figure **??**. This hasn't been implemented in the final product, as the logic to ensure this would work reliably would take too much planning, development time and verification. Both the client and server would have to be modified to ensure that:

- A dropped connection can be seamlessly reformed. This could be done by providing some kind of token to the client upon the initial connection, which can be used by the client to identify themselves upon re-connecting.
- If the Demonstrator disconnects, the Participants are made aware, so they don't think the demo has just ended.
- Once they reconnect, any queued changes to the demo since disconnecting are sent on to the server.
- If a Participant disconnects, they are sent all updates which they missed upon reconnection.

This would introduce a lot of complexity to the server and overall netcode. This requirement is somewhat orthogonal to the rest, in that it doesn't affect the ability to evaluate the effectiveness of the tool. Therefore, this is by no means a disaster. The networking performance could still be improved without this if the server compressed the JSON received from the Demonstrator before it broadcasts it. This would be useful for the Participants, who are likely to be using portable devices with tighter bandwidth constraints. Compressing data is more CPU-intensive than compression, which would have a larger impact for mobile devices, meaning that the bandwidth gained if the Demonstrator compressed data it sent out may not be worth the extra processing effort. However, as JavaScript has no built-in compression functionality and doing so would require further third-party code, this was not attempted.

4.2 Pedagogical

The technical requirements established in chapter 2 were based upon the functionality which an efficient pedagogical tool must offer. *algo.js* meets every level of the criteria for an effective AV tool, as identified by Naps et al. and listed in section 1.1:

1. **Viewing**. A demo can be played, and jumped forwards and backwards step-by-step.
2. **Responding**. As discussed throughout the report, the intended use case for the tool is as a supplement to a lecture or lesson–type scenario, in which the Demonstrator can ask the

Participants questions about what they see and what they think might happen next. Quiz questions would complement this , were they added as discussed in the previous section.

3. **Changing**. Complete control is offered over the algorithm and input data. If one wishes to change the input data or demonstrate a variation of the algorithm, this can be done by just editing the demo code; Participants can also see exactly what is changed.
4. **Constructing**. Evidently, you can demonstrate pretty much any algorithm if it works with one of the two implemented data structures.
5. **Presenting**. The tool tries to blur the line between Demonstrator and Participant mode by presenting the UI and behaviour of the two as similarly as possible. A student who wishes to divert from a demo to show something to a peer can simply copy the demo's code and start their own demo with it.

The WYSIWYC approach regarding immediate edit-time feedback is reached to a degree. Live code editing is not supported, but the code editor and AV are presented alongside one another and the 'feedback cycle', as it were, from editing the code to seeing the effects is reduced to a few mouse clicks. The requirements identified by Rößling and Naps for an AV tool's technical capabilities to aid in a teaching context have also been met: out of all the requirements they identified, only the '**structured view**' requirement hasn't been implemented, as discussed with technical requirement D9. '**Interactive prediction**' again isn't offered directly by the tool, but in a pedagogical context, this is part of the Demonstator's role as a teacher. Finally, the tool is of course presented as a web app, using a collaborative delivery model inspired by *Kahoot!*—sound effects and music haven't been added, but this is secondary to the reachability and collaboration aspects of this model.

4.3 Practical Study

In order to verify that the tool is effective as a teaching supplement in practice, an online trial lecture was held.[1] Three participants were asked to take part in the trial, which was co-ordinated with a Skype call. All participants were familiar with JavaScript's syntax. The participants were asked to use two devices: a computer (laptop or desktop) and a smartphone. The trial consisted of two parts:

- A demonstration of a simplified variant of *merge-sort* to the participants, which accepts input Lists with lengths which are powers of two. This was picked because merge sort is simple and efficient, but is also an algorithm that the participants are less likely to be immediately familiar with, such as Quicksort, so that we can get feedback from the participant about how the tool helped them to learn a new algorithm. Participants were asked to watch the demo on both their laptop and their smartphone.
  Two of the participants were asked to disable Wi-Fi on their smartphones so that they connect over 4G.
- The participants were next asked to start their own demo, and create a demo of bubble-sort themselves. Bubble-sort was chosen as it is extremely simple, and you can demo this with only two List operations (reading and swapping), which means the participants should hopefully not struggle to implement this themselves. They were asked to run the demo on their computers, and watch their own demonstration on their smartphone.

Participants were initially given no guidance of how to use the tool, in order to see whether the UI is intuitive enough for new users to operate the tool without instructions, but were allowed to ask questions if they got stuck. Participants were given a Google Forms questionnaire at the end of the presentation. A Likert scale is used to gauge whether specific elements of the tool assisted

[1] The Covid-19 pandemic, ongoing at the time of writing, prevented a physical classroom-style demonstration from taking place.

or hindered them in following the demo and learning the algorithm, and a free-form text input is used to allow the user to provide qualitative elaboration of these points. The participants were also questionned on whether they had performance or connectivity issues on either device. The questions given are in Appendix A.

4.3.1 Results

The results of the study, which are listed in Appendix B, showed positive feedback from the participants. They expressed a particular interest in the ability to modify the demo on-the-fly: for example, a participant asked what happens when the bottom-up merge sort implementation is given a list whose length is not a power of two. This was answered by changing the code, the participant was surprised that this could be done so easily, but that helped them to understand how the code related to the AV. The colour coding and vertical offsets used by the algorithms were also noted as being useful. The participants found that producing their own bubble-sort demo was quite satisfying. It took a moment for them to work out that you just need to write the algorithm as you would normally, but they quickly pointed out how one could take an algorithm's example code from the internet and very quickly turn it into a demonstration.

In terms of negative feedback, one participant indicated that it wasn't very clear how far ahead/behind they were in the demo when they stepped and unstepped. In the voice call, they suggested a progress bar like a video player, so you can tell where you are in relation to the demonstrator. Also, it wasn't clear to them why the text editor box was disabled with a paused algorithm, and suggested a better visual cue which tells the user that they have to stop the demo to edit it. The participants enjoyed how the different built-in algorithms looked as they were animated, but suggested the hint boxes at the bottom-left should be more frequent and have more explanation in them. During the demo, a participant suggested having two different "levels" of hint—a short hint is displayed by default, which could be expanded into a bigger and more verbose hint. Also, they said that the code for the demos could do with more comments to explain what each part did.

One major suggestion which all of the users verbally commented on when creating a bubble-sort demo was to add some form of code completion, akin to Visual Studio's *IntelliSense*, to help users see what functions are available to read/write to (and annotate) the data structure. They suggested that this would be a lot more useful than a separate documentation page. The participants eventually found out how to use the annotation features like `setColor` and `setVerticalOffset` by inspecting the built-in demos, but stressed that this would be a very useful feature which would make it a lot more natural to write their own demo.

Due to the circumstances preventing a physical trial lecture from taking place, one user suggested some kind of webcam feed in the UI, to show the demonstrator's face. This raises a point that *algo.js* could be adapted into a remote teaching tool with other similar features such as text chat. This is particularly relevant in the current Covid-19 situation; students may struggle with focus and motivation when their normal classroom setting is disrupted, and integrating this kind of feature would streamline remote teaching as students wouldn't have to focus on the two tools separately (learning app and chat app). One feature that would be particularly useful in a pedagogical sense is the ability for the demonstrators and participants to see the location of each other's mouse cursor. This quickly became apparent during the demo, due to a lack of any physical whiteboard or screen to point to. If a participant wants to ask a question about a specific part of the AV, they could point at the AV with their cursor, and the same cursor would appear on the demonstrator's display. This would make the process of asking questions a bit more natural for participants, and would be useful to demonstrators in the same way that a laser pointer can be used in a presentation: to draw attention to specific parts of the screen, and link what you're saying with what they're seeing. This would further integrate the tool with

the teaching process, and being able to just point at something on screen when making a point (and everyone else being able to see) would make the tool a more natural extension of the lesson, versus having to awkwardly explain in words which part of the AV you refer to.

In general, feedback about the UI on all kinds of devices was positive. Some minor performance issues on iOS were noted in the questionnaire by one of the participants running on 4G but they didn't record this as being significant. Overall, the trial lecture provided useful insight on how it felt to use the tool in a collaborative situation. The participants really enjoyed being able to join each other's demos, and remark on each other's code as they were writing it. In fact, they started doing this without any prompting, and it was quite rewarding to see the participants actually explore & enjoy using the tool, and experimenting with modifying the stock demos. This corresponds to the 'Presenting' level of the engagement hierarchy described by Naps et al. in the literature, and the participants in this trial lecture certainly seemed to benefit from this, even if this was only due to the novelty factor of exploring coding in a new way.

4.4 Conclusion

The goal of the project has been to create a web-based AV tool which streamlines the teaching and learning process into an 'integrated demo environment', by making demonstrations easier to perform for the teacher, and easier to visually and mentally consume for the student. While the tool cannot substitute for a teacher, it has certainly shown to help as the basis for a lesson by making the process less clumsy and more fun and interactive. Seeing the participants organically begin to teach one another by exploiting the tool's collaborative aspect shows how it could feasibly serve as a prototype for a new model of classroom interaction.

algo.js is only made possible by the standardisation of the websocket API, powerful smartphones with full-featured web browsers, and the widespread deployment of reliable Wi-Fi and 4G networks—this project would have been completely impossible to do with the technology available at the time most of the prior literature discussed in this project was written. There is a growing realm of potential new models of teaching which exploit what is made possible by advancements in consumer hardware and software capabilities, but it seems like teaching tool design has been slow to catch up to this. *Kahoot!* was one of the first to exploit what can be done with smartphones; its fun music and competitive nature took classrooms by storm, and it went on to become incredibly successful in a short space of time. However, there are so many more ideas that can be done by combining new technologies like this, some of which are surely better and more interactive than PowerPoints, even if they aren't as general. It seems to me like there is a paradigm shift waiting to happen, as soon as someone finds a novel way of refining the teaching process *and* executes the idea well.

For an educational tool to be successful, it needs to be painless to use, and feel like a natural extension of the teaching process. The design and presentation of the tool is just as important as the technical functionality. Students and teachers must be able to use the tool almost without thinking about it, which is only possible if the tool's model of interaction is intuitive enough that they can focus on *what they want to do* for the lesson, and not interrupt their train of thought trying to figure out *how they're going to do it* with the tool. *algo.js* has met the technical goals to blur the line between writing an algorithm normally and visually demonstrating it. On top of this, the trial participants began to explore the collaborative ways of demonstrating their code to each other; the fact they could *use* the tool rather than just play with it like a toy indicates its suitability as a teaching platform. In conclusion, *algo.js* has streamlined the process of demonstrating algorithms in a way that is effective in a teaching context, which is the goal of the project.

References

ajax.org (2020) *Ace - the high performance code editor for the web.* Available at: https://ace.c9.io/ (Accessed: 1 April 2020).

Anderson, J.M. and Naps, T.L. (2001) "A context for the assessment of algorithm visualization system as pedagogical tools." In *First international program visualization workshop, porvoo, finland. University of joensuu press.* 2001. pp. 121–130.

Cherry, B. (2010) *JavaScript module pattern: In-depth. Adequately good: Decent programming advice.* Available at: http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html (Accessed: 16 March 2020).

Dey, A.K., Abowd, G.D. and Salber, D. (2001) A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human–Computer Interaction*, 16 (2): 97–166. doi:10.1207/S15327051HCI16234_02.

Fraser, N. (2020) *JS-interpreter documentation. JS-interpreter.* Available at: https://neil.fraser.name/software/JS-Interpreter/docs.html (Accessed: 10 March 2020).

Grissom, S., McNally, M.F. and Naps, T. (2003) "Algorithm visualization in CS education: Comparing levels of student engagement." In *Proceedings of the 2003 ACM symposium on software visualization.* 2003. pp. 87–94.

Hundhausen, C.D. and Brown, J.L. (2007) What you see is what you code: A "live" algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing*, 18 (1): 22–47.

John McIntyre (2016) *MoSCoW or kano models - how do you prioritize? HotPMO - PMO that delivers faster.* Available at: https://www.hotpmo.com/management-models/moscow-kano-prioritize (Accessed: 1 April 2020).

Kelling, Z. (2020) *Zeekay/bottle-websocket.* (original-date: 2011-09-26T00:55:04Z). Available at: https://github.com/zeekay/bottle-websocket (Downloaded: 18 March 2020).

loading.io (2019) *Pure CSS loader - optimized spinners for web · loading.io.* Available at: https://loading.io/css/ (Accessed: 24 March 2020).

Naps, T.L., Rößling, G., Almstrum, V., et al. (2002) "Exploring the role of visualization and engagement in computer science education." In *Working group reports from ITiCSE on innovation and technology in computer science education.* pp. 131–152.

Plump, C.M. and LaRosa, J. (2017) Using kahoot! In the classroom to create engagement and active learning: A game-based technology solution for eLearning novices. *Management Teaching Review*, 2 (2): 151–158.

Rößling, G. and Naps, T.L. (2002) "A testbed for pedagogical requirements in algorithm visualizations." In *Proceedings of the 7th annual conference on innovation and technology in computer science education.* 2002. pp. 96–100.

statcounter (2020) *Top 9 mobile & tablet browsers, feb 2020. Statcounter.* Available at: https://gs.statcounter.com/#mobile+tablet-browser-ww-monthly-202002-202002-bar (Accessed: 9 March 2020).

two.js (2020) *Two.js documentation. Two.js.* Available at: https://two.js.org/ (Accessed: 10 March 2020).

Appendix A

Questionnaire

1. These questions are about the merge-sort demonstration. *(**Note**: The questions are numbered from 1 to 4: 1 means "It didn't help me at all/I didn't notice this", 4 means "This was extremely useful".)* For the text input areas, a couple of sentences would be great.
    a. How helpful was the animated visualisation for improving your ability to understand merge-sort? (1-4)
    b. Is there any any particular aspect of the demonstration which helped you to understand the algorithm better? (text)
    c. Which aspects of the visualisation did you not like/found unhelpful, or what do you think this feature is missing? (text)
    d. How well did the view of the running code aid your understanding of the demo? (1-4)
    e. Is there any particular aspect of the code view which helped you to understand the algorithm better? (text)
    f. Which aspects of the code view did you not find useful, or what do you think this feature is missing? (text)
    g. How did the ability to step forward and backward through the merge-sort demo affect your experience? (1-4)
    h. Is there any particular aspect of being able to step/unstep which helped you to understand the algorithm better? (text)
    i. Is there an aspect of the step/unstep functionality that you didn't like, or something you thought is missing? (text)
2. These questions are about your experience creating and running a bubble-sort demo. In your code, you sorted a List object. How how intuitive did you find the relationship between your code's usage of this object and the animated visualisation?
    a. In your code, you sorted a `List` object. How well did you understand the relationship between your code's usage of this object and the animated visualisation? (1-4; 1 means "I struggled to understand how my code affected the visualisation", 4 means "It made sense immediately")
    b. What did you find useful about this model of creating demonstrations? (text)
    c. What didn't you like about this model?—or, if you think it would be improved with

some other functionality, describe what could be added. (text)

3. These questions are about the tool as a whole.
    a. How intuitive did you find the user interface? (1-4, 1 means "I struggled to navigate the tool's interface", 4 means "The user interface felt natural and intuitive")
    b. If there was a point during the lecture where you wanted to do something, but the UI didn't make this clear, please describe what. (text)
    c. Did you notice any graphical stuttering on your laptop? This is where the demo updates, but the animation looks choppy. (1-4, 1 means "Too much for the tool to be useful", 4 means "none at all")
    d. Did you notice any graphical stuttering on your phone? (1-4)
    e. Did you notice any network lag on your laptop? This is separate from the animation - an example would be the demo not updating and ruining the timing of the animation. (1-4)
    f. Did you notice any network lag on your phone? (1-4)
    g. What operating system and web browser are you using on your computer?
    h. What manufacturer and web browser are you using on your phone?
    i. Was your phone using Wi-Fi or 4G? (option between Wi-Fi and 4G)

Appendix B

Results

The answers by each of the 3 participants for each question are listed below. The data was exported from Google Forms to Google Sheets, and then processed into a table.

	Response A	Response B	Response C
1a	4	4	4
1b	By changing the colour of each cell to identify which cells were in which groups and which cells were already sorted really helped. It focused the viewer's attention on specific parts of the algorithm.	Be able to move the specific couple of numbers you are looking at away from the rest made it really easy to see the different steps as individual problems that can be solved separately from everything else.	The colour-coding made things much clearer. Being able to see the real-time impact on the visualisation of changes in the code helped relate the code to the algorithm.
1c	I felt that the small text boxes that appeared in the bottom corner were very useful at reinforcing what the demonstrator was explaining. I would have liked to see more of them at each iteration of the algorithm.	Would be super useful for visualising more complex image processing algorithms for example, and i don't believe it should just be aimed at kids. For this something like a visualisation of a 2D array would be useful.	A way to stream a video alongside the demo to be viewed by those watching the demo. This would allow for remote teaching.
1d	3	4	4
1e	The ability to see the for-loop in the code showed me how this process was an iterative process and which variables were being checked.	Highlighting and font size was good and made it clear to understand.	Highlights in the code to show what lines have just been run in addition to the comments on the demo about what has just been done.
1f	The code view was great and the only improvements that could be suggested would be ensuring that the code itself is commented enough to help explain what each line of code did.	Would be nice to have auto completion for the functions that List and Graph provide, as this would allow you to see what sort of things you can do to the visualisation.	Maybe some more comments about what each part of the code is doing for people learning to code
1g	4	4	4

	Response A	Response B	Response C
1h	Being able to go back helped in case you either clicked too far ahead by accident or failed to understand what had changed between this step and the previous step.	When there was a bit that was not understandable it was useful to be able to replay this part again and again until understood	Being able to pause the demo and see previous steps only on my screen rather than the whole group having to go back will be great if someone needs a bit more clarification but don't want to hold the others back.
1i	No, I thought the step/unstep feature is great as it is.	Nope - thought it was as good as is required.	Maybe an indicator on the demonstrators screen to indicate that someone(s) is not at the same point of the demo to ensure they don't get left alone.
2a	4	4	4
2b	The ability to add a new element to the list easily by putting it in the constructor, rather than having to write .add() multiple times helped me keep track of which elements were in the list.	You didn't have to change much of the code from a normal bubble sort to support the visualisation meaning it would be easy to copy some example code from online and make it work quickly with the demonstration.	A nice starting point of the code was given to help understanding the object. Changes in the code having instant effect on the model.
2c	I felt that List model was great at its job; I can't think of anything I would change.	A swap function was required for the visualisation to show a swap. Possibly some sort of look ahead to work out that two values have been swapped manually and playing an animation would be useful - but may be too complex / not possible to do well.	Maybe a way to save the code so that it wasn't lost when moving between screens.
3a	4	3	4
3b	No, I couldn't find a feature, clicked the hamburger menu and it was there. The UI is very user-friendly and I'm sure that any secondary school pupil familiar with computers would be able to use this in a lesson.	Got confused about the fact that I couldn't edit the code while it was running but In a paused state. Maybe setting the cursor to a X or something would allow users to quickly see they couldn't edit it, and a message showing how to make it editable again.	N/A
3c	4	4	4
3d	3	4	4
3e	4	4	4
3f	3	4	4
3g	Safari 13.1 on MacOS 10.15.4	Mac OS with Google Chrome	Windows and Google Chrome
3h	Safari 13.1 on iPhone 8 running iOS 13.3.1	Apple iPhone 7 with Safari	Moto and Ecosia *(Android and Chrome)*
3i	4G	4G	Wi-Fi

Appendix C

Additional UI screens

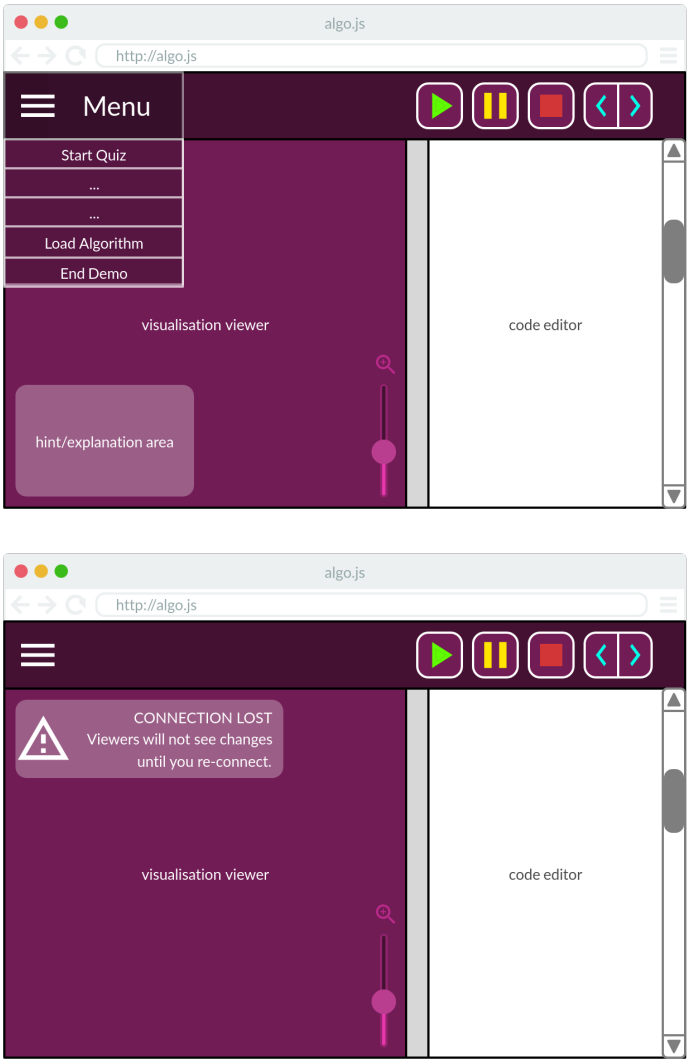


Figure C.1:
The hamburger menu, accessible by clicking/tapping the icon in the top left, above; and a warning label indicating that the network connection has been interrupted, below.

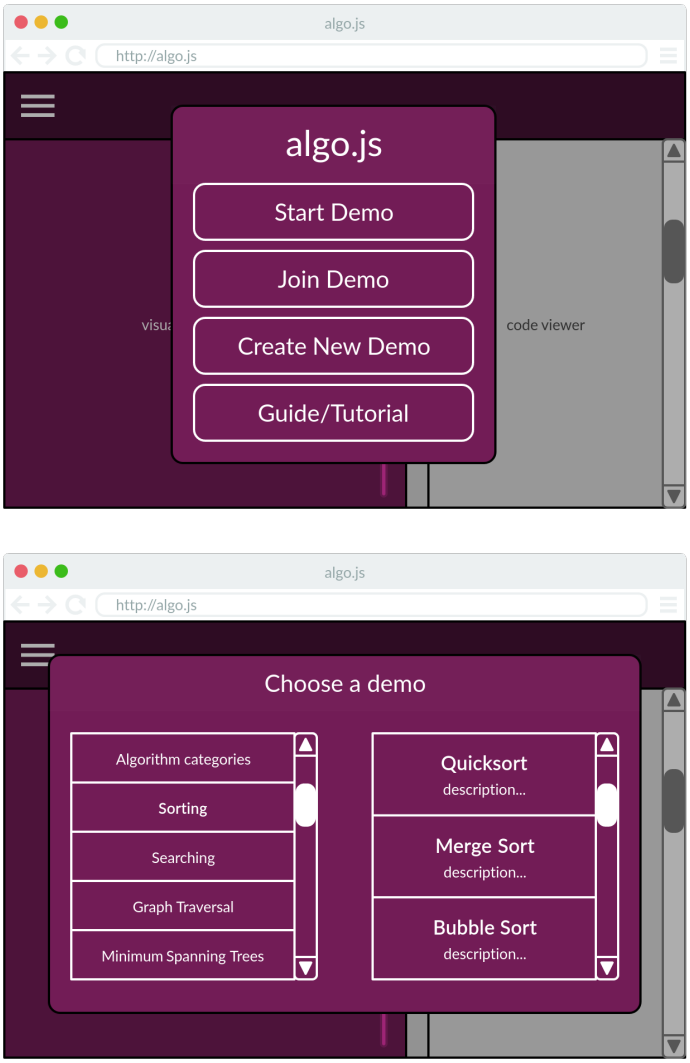


Figure C.2:
The welcome menu presented to the user when the web app is loaded, above; and the screen allowing selection of a built-in algorithm with which to start a demo, below.