

Palgo - Algorithm Animator

Andrei-Mihai Nicolae

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — April 4, 2017

Abstract

Understanding algorithms is both very common and hard for students and developers in general, regardless of their level of expertise. Even fundamental algorithms, such as Dijkstra's algorithm for finding the shortest path between two vertices in a graph, are quite complicated to grasp. Many studies show that visualizing an algorithm and its steps enhance understanding. In this report, we will present an Algorithm Animator built specifically for solving this problem in a modern, responsive and efficient manner. We will also show why certain design decisions, implementation choices and evaluation results make this tool a viable option for students and software engineers when it comes to learning different kinds of algorithms.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

1	Introduction	1
1.1	Aims	2
1.2	Motivation	2
1.3	Contributions	4
1.4	Report Content	4
2	Background	5
3	Requirements	10
3.1	Problem Analysis	11
3.2	Requirements Gathering	11
3.3	Functional Requirements	12
3.4	Non-Functional Requirements	12
4	Agile Software Development	13
4.1	Extreme Programming	13
4.2	Planning	15
4.3	Continuous Integration	16
4.4	Issues & Bug Tracking	17
5	Design	19
5.1	Architecture	19
5.2	Application Type	20
5.3	Prototyping	21

5.4	Electron	23
5.5	Drawing	24
5.6	Animation	25
5.7	Material Design	26
5.8	Compromises	27
6	Implementation	28
6.1	Algorithm Selection	28
6.2	Project Structure	28
6.3	Animation Pipeline	29
6.4	JavaScript Multi-Threading	31
6.5	Extra Features	33
6.6	Lessons Learned	34
7	Testing	36
7.1	Unit Testing	36
7.2	Integration Testing	37
7.3	Prototype Evaluation	38
7.4	Results	38
8	Conclusions	40
8.1	Open Source	40
8.2	Project Roadmap	40
8.3	Final Thoughts	41
8.4	Acknowledgements	41

Chapter 1

Introduction

The starting point of this report should be defining what an algorithm is.

NOUN

A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.
'a basic algorithm for division'

Figure 1.1: Algorithm definition taken from the Oxford English Dictionary.

As we can see in Figure 1.1, it is a “process or set of rules” followed when trying to solve a problem. Even if it sounds easy in theory, reading lines of code and then understanding what steps are necessary for a machine to perform to generate the expected output is rather complex and hard for a human being to comprehend especially if the algorithm presented is complex. In a study conducted by Christopher D. Hundhausen et al. [23], it was proven that even though algorithm visualisation is not the “perfect” solution, it is definitely effective in teaching students and experienced developers how algorithms work.

There are many tools out there that already provide animations of algorithms, however one thing was a key factor in deciding how to proceed with planning the whole software process: there are barely any standalone applications. Even though one can find many websites which let people visualise algorithms’ steps (e.g. a good example is VisuAlgo [21]), this is not such a viable solution as the user cannot use the application whenever they want to, as an Internet connection is required.

Therefore, the plan was to create an app that can run offline and would feel natural to the user regardless of the operating system of choice. There are many Java implementations (i.e. resulting in jar executables), but one of the main drawbacks of using Java is the lack of proper GUI tools that can build native-feeling applications. As such, the decision was made to use the Electron framework [14] originally built by the team behind the Atom text editor. This framework uses only web tools to generate executables for the three main OS families (i.e. macOS, Windows and Linux) that provide the user with a native, modern and responsive feel.

The planning was made using the best agile practices [7], eventually deciding on following a variation of XP programming. Designing an architectural plan was a major milestone to be reached as it took a considerable amount of time to be put in place. However, as discussed in future chapters, the implementation of the animation engine was a crucial and time consuming challenge that required the highest amount of effort.

The report also presents how the application was tested in various ways, as well as how it was evaluated using potential end users. Finally, we shall discuss the roadmap of the project, the importance of the open source community for Palgo as well as some final thoughts and lessons learned during the process.

1.1 Aims

The goals of the project were set and subsequently refined during many project meetings, as well as meeting with fellow classmates to get feedback along the way.

The main aims of the Algorithm Animator, however, have always been:

- Create an efficient and easy-to-use animator.
- Make the animator a cross-platform application that would run natively on the main operating system families: macOS, Windows and Linux.
- Provide a user-friendly interface that would make the user enjoy using the product.
- Create at least 5-6 fully functional algorithm animations.
- Make the application scalable and easy to maintain.
- Test all functionality and ensure quality above quantity.
- Evaluate the product throughout the software process to meet acceptance criteria.
- Build a roadmap that would allow the animator to grow even after the level 4 project has finished.

1.2 Motivation

Even if we have previously mentioned that visualising algorithms is one of the best ways of understanding how they perform, there are also other key motivational aspects behind the need of building an animator.

One such motivation is the increasing demand of software engineers throughout industry as well as the rise in level of expertise. Developers, and students in particular, will benefit tremendously from a good grasp on how to implement correct and optimal algorithms when applying for a new job. Thus, an algorithm animator would be a good component in one's tool belt.

Another major aspect was the opportunity to create a modern application that users would enjoy using. At the moment, one of the most popular animator toolkits is the one presented in John Morris' paper [27]. Figure 1.2 depicts a screenshot taken while using this toolkit.

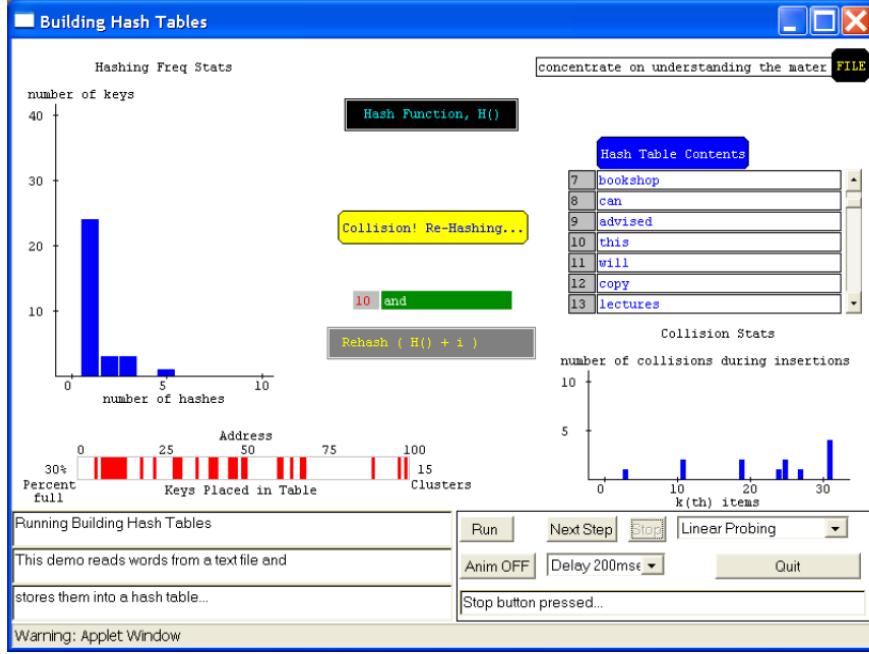


Figure 1.2: John Morris' Animator Toolkit.

This toolkit was written in Java, which comes with certain advantages: multi-threading support, enhanced familiarity due to university coursework and projects etc. However, the lack of proper graphical user interface components makes this a not so viable option for regular users when adopting it. On the other hand, Figure 1.3 shows our proposal to solve these shortcomings.

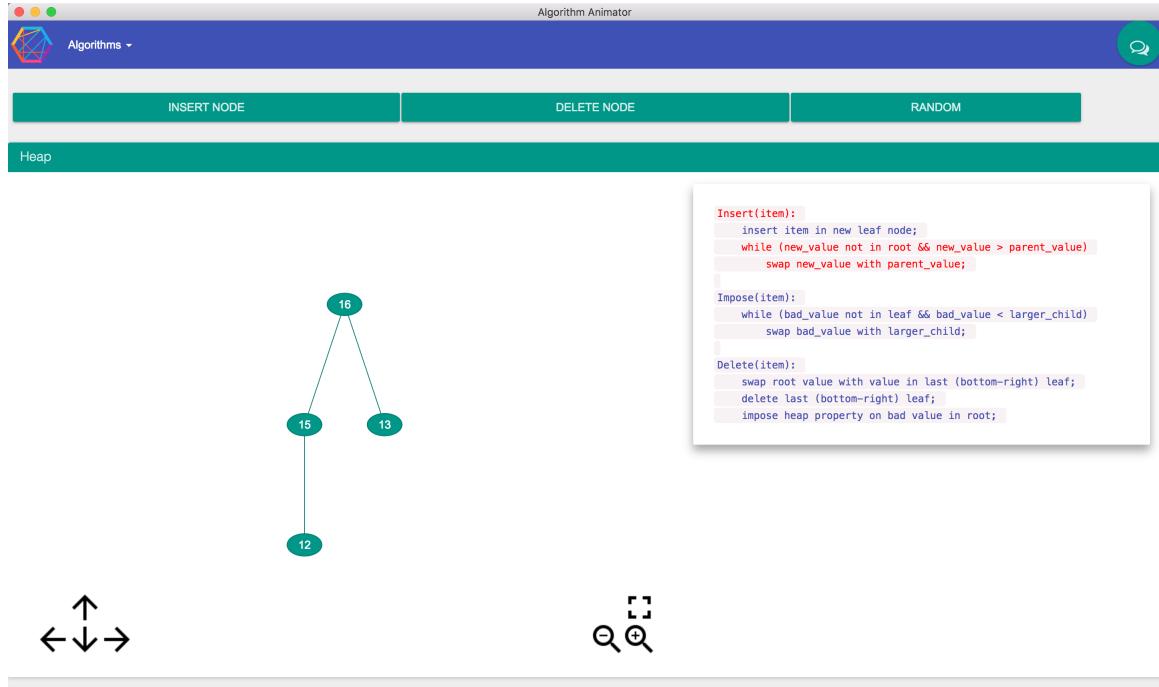


Figure 1.3: The animator presented in this report.

We can see a clear difference between the two figures and why one might choose the latter option due to familiarity (native feel) and easy-to-use user interface.

1.3 Contributions

This report serves to deliver the following contributions.

- Describe the whole software development lifecycle of an algorithm animator.
- Present the various technologies used to build native and modern desktop apps [14] [43]
- Provide a brief overview of current animators available and background knowledge on their benefits and efficiency in teaching.

1.4 Report Content

The remainder of the report will analyse the background of animators and why they were proven useful, as well as cover all the steps in gathering requirements, designing, implementing, testing and evaluating the tool.

- Chapter 2 covers work related to the purpose of algorithm animators and why they are useful.
- Chapter 3 goes into how the problem was analysed and what requirements were gathered through project meetings and discussions with Algorithmics students.
- Chapter 4 shows the steps undertaken to follow the best agile methodology principles.
- Chapter 5 explains the design decisions behind the tool and illustrates various lessons learned and problems faced along the way.
- Chapter 6 goes into the implementation details of the animator.
- Chapter 7 show how extensive unit, integration and other types of testing (e.g. smoke, end-to-end) were undergone and why they were essential to the development of the application.
- Chapter 8 details the overall results of the project.

Chapter 2

Background

As stated in the introduction, the first step is to define what an algorithm is. We will use the above-mentioned definition taken from the Oxford dictionary [12]: “A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer”. In our case, we will only relate to set of rules followed solely by computers. Thus, put simply, an algorithm is a set of steps defined by someone in order to solve some problem. As an algorithm can be written by anyone, there is no known number or catalog of them. However, the most important algorithms (e.g. Huffman tree encoding/decoding) are taught in any Computer Science course at universities around the globe.

Having defined what an algorithm is, let’s take an actual example - Dijkstra’s algorithm for finding the shortest path between two vertices of a graph [13], first published almost 60 years ago (the pseudocode was formally checked by Dr. Norman). Listing 2.1 presents the pseudocode for the algorithm.

```
1 // S is set of vertices for which shortest path from u is known
2 // d(w) represents length of a shortest path from u to w
3 // passing only through vertices of S
4 S = {u}; // initialise S
5 for (each vertex w) d(w) = wt(u,w); // initialise distances
6 while (S != V) { // still vertices to add to S find v not in S with d(v) minimum;
7     add v to S;
8     for (each w not in S) // perform relaxation
9         d(w) = min{ d(w) , d(v)+wt(v,w) };
10 }
```

Listing 2.1: Pseudocode for Dijkstra’s shortest path algorithm.

As one might observe, this is not trivial to grasp. Therefore, many solutions have been created in order to help students and engineers in general understand them better. One such solution is a rather creative one, developed by students at Spatiota University, which tries to teach different sorting algorithms by dancing (Figure 2.1).

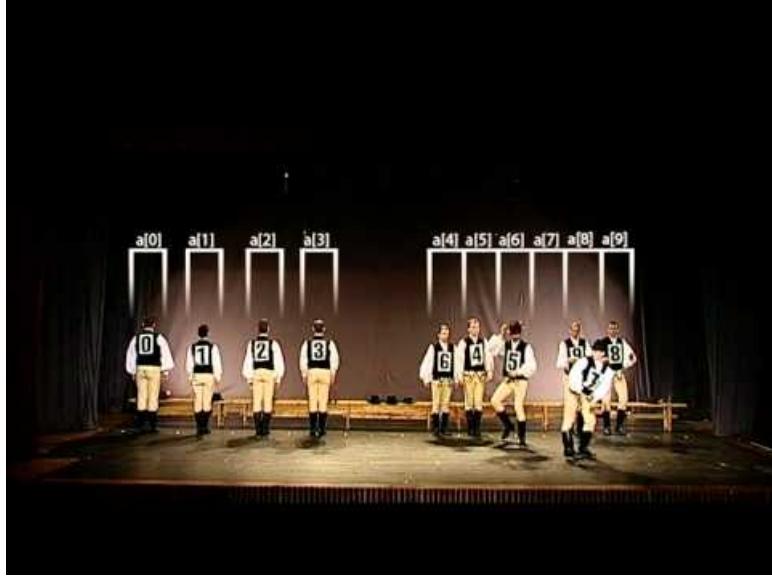


Figure 2.1: Sorting algorithms learning through Hungarian folk dancing.

This was featured on many websites and gained enormous traction among computer science enthusiasts. However, a more traditional approach that is adopted formally in universities as well is the use of animators and animation.

Firstly, let us define what an animator is. Again, taking the definition from the Oxford dictionary [12], it is “a person who makes animated films”. In our case, the person is the computer and the film is the sequence of steps required to produce an output after feeding input to an algorithm.

Usually, algorithm animators are divided into 2 sections: one section will contain the graphics in order to represent the data structures used (e.g. in graph traversal algorithms the vertices and edges comprising the graph will be drawn) while the other will have the code lines, the actual rules the computer needs to follow in order to produce the desired output. The current code line that is executed at a certain moment is highlighted, while the data structure visualized (e.g. some box that represents an array’s element) changes and the user can see the output visually and immediately, making a mental connection between the code line and the change it produces. Figure 2.2 will present the two animation sections we have used in Palgo.

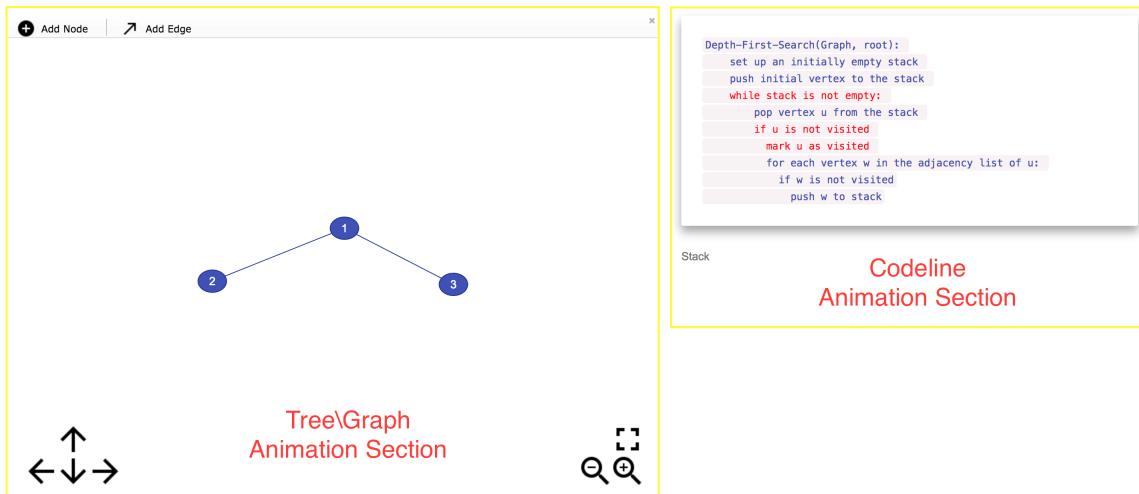


Figure 2.2: Sorting algorithms learning through Hungarian folk dancing.

But why an animator precisely? There are many studies that show visualization in any area can enhance

learning capabilities drastically. Brenda Parker and Ian Mitchell's work [33] shows clearly how seeing the steps can greatly improve a student's understanding of what is happening "behind the scenes". On the other hand, even if animations were proven to be generally useful and beneficial for the learner, the study shows the need for visualising algorithms animated declines over time.

Another study by Susan Palmiter and Jay Elkerton [32] tried to find out which method was the most effective in learning new computer-based tasks. Even though animations for complex mechanisms were found to be ineffective when compared to animations for easier algorithms (e.g. parallel programming vs. graph traversal), they were shown to be very helpful for understanding what is generally taught at university courses.

Having discussed methods of learning new algorithms better and why animators are useful, we need to take a look at previous solutions developed. Apart from John Morris' tool shown in Figure 1.2, there are many other applications available, including Galant [40] and the visualizer built by D. Galles [16].

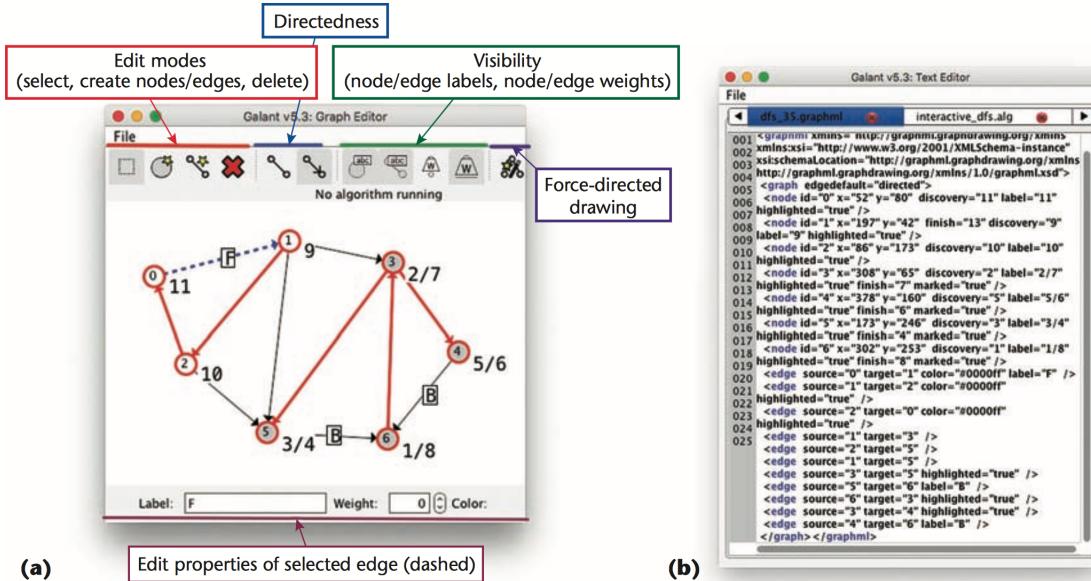


Figure 2.3: Galant Algorithm Animator.

In Figure 2.3 we can see one of the two main approaches for building an animator, which is a standalone JAR application. Thus, it is written completely in Java and, as the language is pre-installed on most electronic devices, it can run on the major operating systems.

There are obvious advantages to this approach:

- the end product can run on a variety of machines with only one codebase;
- the large number of Java libraries facilitate an easy development workflow;
- there are many tutorials and resources available;
- Java is taught at most universities in introductory courses, making it easier for newcomers to create their own application rapidly; Figure 2.4 shows the most taught programming languages at the top 39 U.S. Computer Science departments.

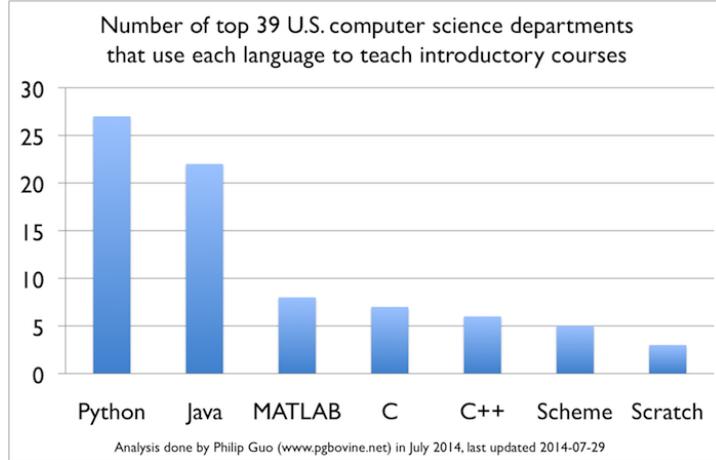


Figure 2.4: Languages used in introductory courses at top US universities.

However, there are also drawbacks to creating a standalone JAR application:

- the most popular Java libraries for producing standalone graphical JAR applications output old-fashioned user interface;
- the code for drawing on the canvas and animating in Java (using Swing or JavaFX) can become extremely verbose and unreadable.

The second main approach for building an animator is web-based, one of them being D. Galles' application [16]. Figure 2.5 shows a screenshot taken while using this application.

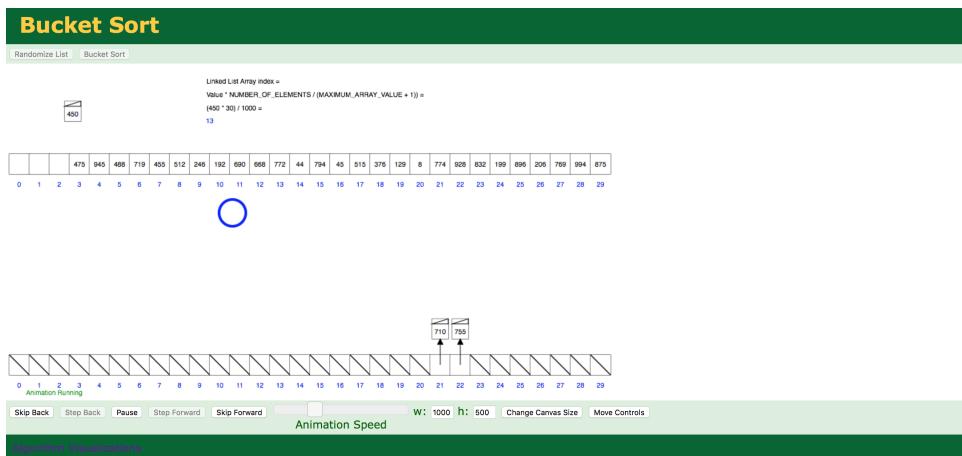


Figure 2.5: Algorithm animator built by D. Galles at University of San Francisco.

The main advantages of a web-based approach are that:

- the developer can use a wide variety of JavaScript tools (e.g. drawing directly on HTML5 canvas, VisJs and SigmaJs) to draw and animate objects;
- the application can have a very modern and responsive feel which can enhance users enjoyment the product;
- virtually all users have at least a browser installed on their machine, thus making the animator available to everyone.

However, the main drawback is that it is still a website, thus it need to be hosted (which implies additional costs) and it cannot be reached by everyone at all times due to the fact that not everyone has constant internet access.

We have covered all the necessary background and related work before diving into how Palgo was designed and implemented. We believe that our solution is a combination of the two main approaches listed above, merging the benefits of both into a single tool.

Chapter 3

Requirements

As for all other software built with best practices, requirements need to be gathered, defined, analyzed and then prioritized. As we tried to follow an Agile structure, the requirements changed every week through student-supervisor weekly meetings, as well as end user meet-ups.

We tried to keep the requirements SMART [25], which implies they are specific, measurable, attainable, realizable and time bounded. Making the requirements follow this well established format made the software development lifecycle much faster and cleaner.

Figure 3.1 shows a screenshot from a Trello [41] board used during one of the iterations (a separate board was used for tracking non-functional requirements as well).

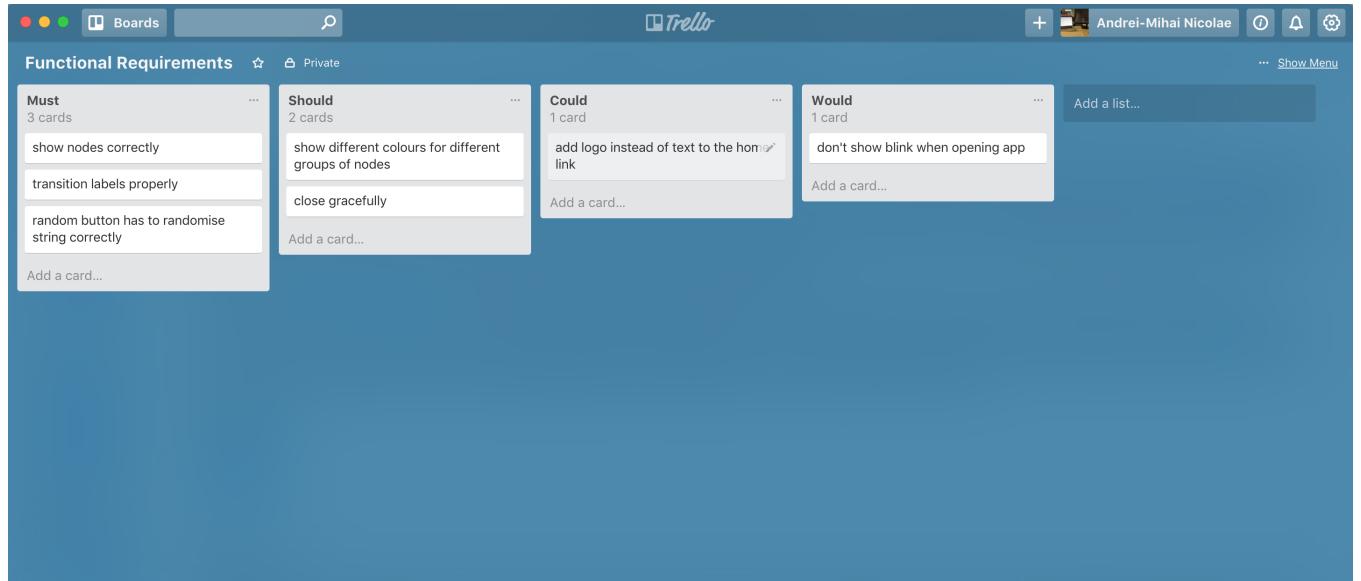


Figure 3.1: Trello board for functional requirements during one of the iterations.

The requirements followed a pipeline through which they, together with their rationale, would get broken down into pieces continuously until user stories could be produced. One example of such a requirement is: “Show me a correct and easy-to-understand animation for the insert function in a heap data structure.”. This would then be broken down in multiple user stories, such as “Colour the node that is inserted and add its label” and “Highlight code line corresponding with changing the label”.

We also tried to follow the MoSCoW [9] guideline for prioritizing the requirements. The technique has

proven to be very effective and it is adopted widely in industry, even though the approach is more than 20 years old. This facilitated a smooth workflow and concise goals to be targeted.

3.1 Problem Analysis

We have analyzed the problem systematically, dividing it into the following main questions:

- who are our target users?
- what type of format will the application have (i.e. web/desktop)?
- what algorithms should be animated?
- how would the main layout be structured?

During the analysis, we needed to take into account various factors, including how much experience with algorithms and computer science in general will our users have and what algorithms would need visualization the most. As previous animators built under the supervision of Dr. Norman are used for educational purposes in the level 3 Algorithmics I course, we believe that having mainly graph and tree algorithms would benefit the project the most. However, we tried to keep the structure simple and the codebase easy-to-maintain so that other families of algorithms will be straightforward to implement. Because we want to project to be continued in the open source space, users other than students can use the animator, thus we tried to keep the requirements more general.

Also, there was much discussion around whether to use a web or a desktop approach. For reasons explained above, we decided to implement a desktop application using the technologies mentioned in the subsequent chapters.

We have gathered the requirements and split them into functional and non-functional, then dividing them further into smaller tasks to be implemented. In the end, we were left with user stories that could be translated directly into work tasks.

3.2 Requirements Gathering

As mentioned above, the requirements gathering was a continuous process that lasted from the first week of 4th year up until the end of the project's development lifecycle. Not only were there the regular weekly student-supervisor meetings where we could add or refine requirements, but there were also meetings with possible end users. More precisely, every two or three weeks there were meetings scheduled with a few classmates from University of Glasgow's Computer Science course that would give continuous feedback after testing the newest version of the application. The meetings were held in one of the laboratories in the Boyd Orr building where the classmates were provided with an executable that could be run on their personal laptops. They were given around 15 minutes to try the features and, afterwards, they provided invaluable feedback without which the animator would not perform as efficiently as it does today.

3.3 Functional Requirements

Functional requirements target specific functionalities that the system should perform. They started with the algorithms that needed to be animated. Then, functionalities such as current code line highlighting and specific button behaviors began to stack up. Eventually, we can categorize them as follows:

- animate tree/graph depending on what the user chose to do (e.g. insert/delete node);
- highlight current code line for every algorithm;
- define specific behavior for every button displayed on the UI;
- each window interaction should perform as expected depending on the operating system the user is using the application on;

Many smaller functional requirements appeared along the software process, but the above mentioned are the main categories.

3.4 Non-Functional Requirements

A non-functional requirement is a type of requirements that judges the system as a whole and how it operates rather than checking for specific criteria. The main non-functional requirements were rather set from the beginning. We decided to have an algorithm animator that would:

- be easy to use and let the users familiarize rapidly;
- make users enjoy playing with the product;
- be installed/fetched with ease;
- allow end users to use it on any platform;
- be efficient and not consume excessive resources;
- be maintainable and scalable (i.e. let future possible contributors add algorithms easily).

Chapter 4

Agile Software Development

Good software practices are key to successful projects. We have followed Agile methodologies, in particular, the methodologies that were taught last year in the Professional Software Development class, as well as through Ian Sommerville’s “Software Engineering” book [37].

We eventually decided to follow an Extreme Programming path and we implemented the best advice taken from industry experts. Some of the key principles we implemented throughout the whole process are:

- good planning strategy that would allow us to follow a schedule; our cost estimation efficiency improved with every iteration;
- frequent commits and releases that would generate continuous feedback from our potential end users;
- customer involvement (i.e. student meetings every few weeks as well as weekly student-supervisor meetings);
- heavy refactoring as opportunity arose;
- emphasis on simplicity in design and implementation;
- tracking bugs and issues in general allowed us to be always informed and include necessary items for specific milestones so that the quality would always have high standards.

These principles ensured a good workflow with results that could be remarked immediately. We believe that a major part of the success of the project was embracing above Agile methodologies.

Gethin:
not sun
what yo
mean b
“remarked”

4.1 Extreme Programming

We have followed the approach first coined by Kent Beck called Extreme Programming (XP) [3]. It is an Agile technique through which good practices are pushed to “extreme” levels. XP has proven to be very effective as many companies use it in their development lifecycle. Principles it promotes that we have followed include:

- frequent releases based on requirements gathered incrementally in meetings (e.g. supervisor-student meetings)
- constant refactoring to promote simplicity;

- customer involvement throughout the whole process (i.e. getting feedback every 2-3 weeks from students that might use the application);
- people over processes, which means collective ownership of the code as well as avoiding exhausting long working hours.

Burndown charts are another great tool used by developers using XP in their schedule to track progress. You start off by defining the number of user stories you want to complete in an iteration and, then, divide that number by the number of days the sprint has (in our case it was 7 days). Therefore each day you should implement a certain number of user stories such that at the end of the iteration you are left with none. That is the most productive outcome. Also, because this was a student project, we will count weekend days as working days.

During the entire process we tried to monitor the progress and see how we are performing. Figure 4.1 shows a burndown chart from one of the November weekly iterations.

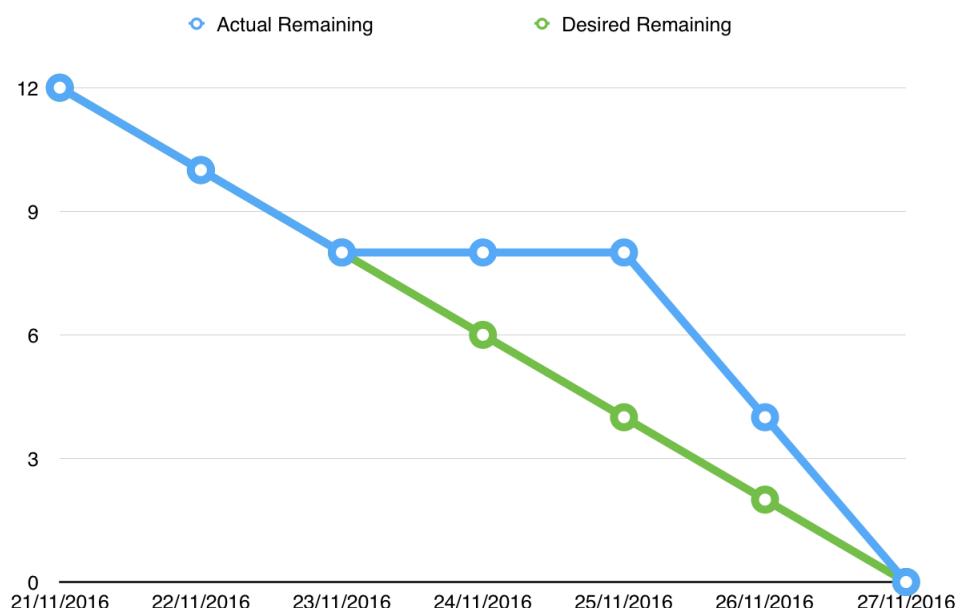


Figure 4.1: November iteration burndown chart.

As one can notice, there is quite a discrepancy between the desired number of user stories per day and what we actually completed. This was because of lack of consistency, shown in the graph between days 23 and 26, where there were no user stories finished. However, Figure 4.2 depicts a burndown chart corresponding to one of the last iterations of the project.

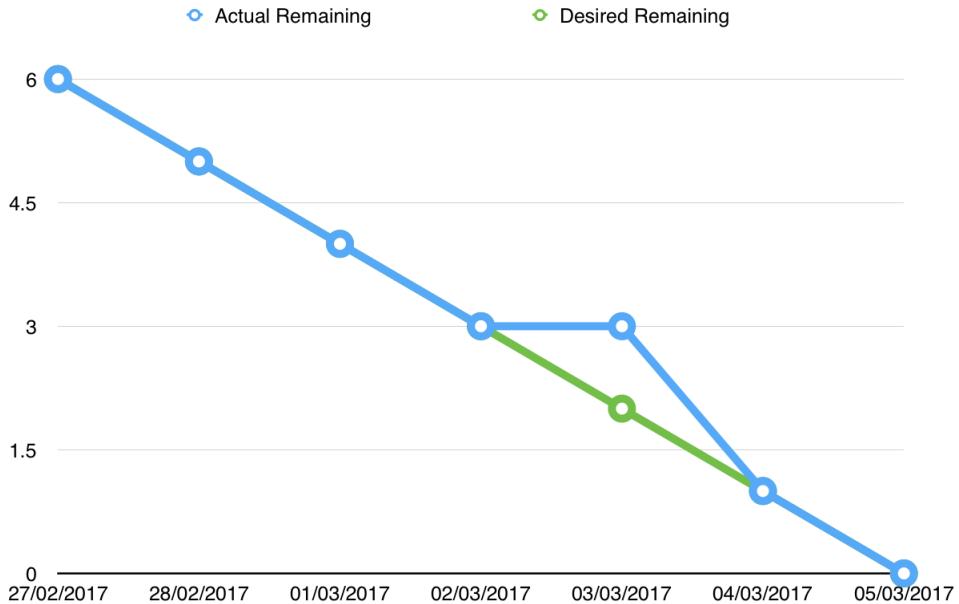


Figure 4.2: February-March iteration burndown chart.

There are noticeable difference between the 2 iterations: firstly, we started with fewer user stories to be completed in the latter sprint which was much more feasible to finish and it did not make us work extra hours (like in the first burndown chart example), thus violating one of the extreme programming principles. Moreover, there was only a slight discrepancy from the schedule on March 3rd where there were no user stories completed. However, this is relatively minor and we believe that for a two people project, this was an amazing achievement that could only be reached gradually.

Summing up, following all the good practices of XP has brought us great benefits and a working schedule that allowed us to implement all the requirements gathered in user and supervisor meetings.

4.2 Planning

Planning is a vital part of every Agile project - it involves setting deadlines for tasks needed to be implemented by developers. However, as some tasks are not trivial to estimate, multiple strategies have been invented so that team members can effectively set number of days (or other units of work) for every user story.

In our case we adapted the most effective Agile method, which is Planning Poker [19]. In this practice, all developers are given decks of cards and when a user story is brought up, every developer shows a card which has a value. That value will represent the number of units of work needed to complete the story. Then, after everyone shows their estimation, discussions emerge among the developers until a consensus is reached. In our situation, we did not use a deck of cards during the student-supervisor meetings, but we both estimated the number of days needed to implement a feature and, if our opinions diverged, we discussed until we had the same duration in mind.

It can be observed in Figure 4.3 how we kept track of all the user stories using Trello cards. We added the estimation in the description, as well as a description using the popular format “As a .../I want to .../So that ...” [11] to always know the rationale behind every implemented feature.

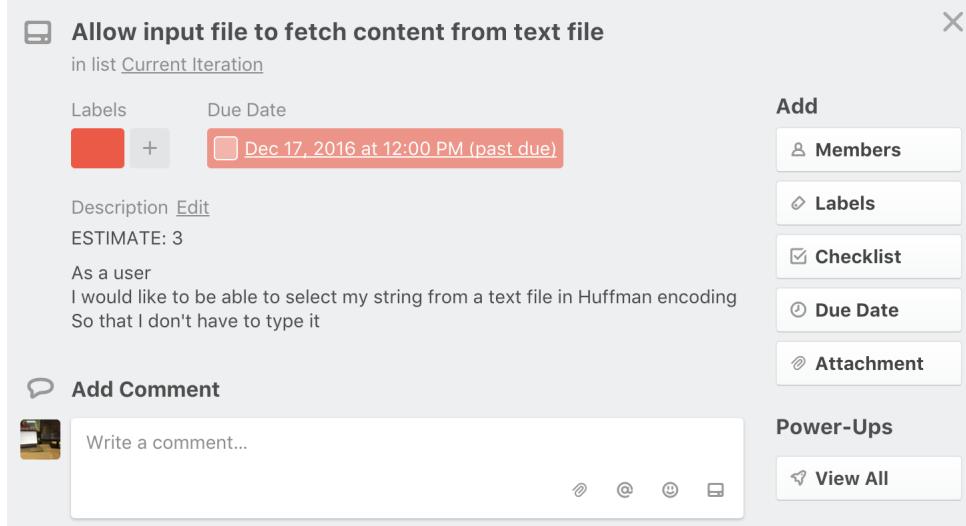


Figure 4.3: Trello card for estimating a user story.

This method was very effective and as we gained more experience after several iterations, we would come to a conclusion much faster and the user story cost estimations became more accurate.

4.3 Continuous Integration

We have used another tool heavily promoted by Agile methodologies which is Continuous Integration (CI). CI means merging all working copies of everyone who is developing on the project to a shared mainline multiple times a day [5]. This is a very useful tool for checking that the builds do not break regardless of the changes made and that quality is of primary importance.

In our case, we have used CircleCI [8] as the tool for CI. It was configured such that after every commit, the code would be merged into the main branch, tested to see if everything is working as expected and, only after, the code would be pushed.

 removed unnecessary edge popup for heap nclandrei committed 8 days ago ✓	View	aaefac6	Copy
 changed layout of heap algorithm page nclandrei committed 8 days ago ✓	View	1db77c1	Copy
 refactoring in prim jarnik versions nclandrei committed 8 days ago ✓	View	delf1d1	Copy
 solved bug that did not remove items from ntv table nclandrei committed 8 days ago ✓	View	a2aaa25	Copy

Figure 4.4: CircleCI status showing red/green depending if the build was successful on GitHub.

As it can be seen in Figure 4.4, there is a small green check on the GitHub repository of the project telling the developers that the build worked as expected (there would be a red cross in cases when the build did not work as expected). This was a great asset to the development lifecycle, informing us whether we needed to add bugs to our tracking system to be solved later or if everything performed exactly as it should have.

Not only did we applied continuous integration to our workflow, but we also used it properly: as mentioned above, the definition of CI means the developers should push their local changes to the main repository multiple

times a day. Every day of development meant multiple commits and multiple pushes so that we made sure the builds were all passing and the code was not broken.

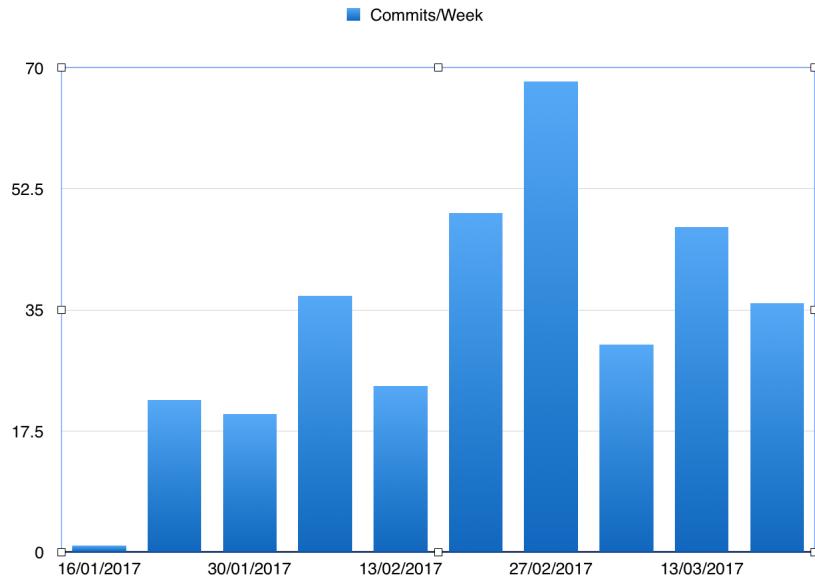


Figure 4.5: Commit frequency in the last 10 sprints.

Figure 4.5 shows the commit frequency per week for the last 10 iterations. As it can be observed, apart from first week which corresponds with the first week of the University semester, the number of commits was very high for every sprint, thus following the practice of committing and testing the quality of our product often.

4.4 Issues & Bug Tracking

Issues and bugs occur all the time in every piece of software. Because the userbase was rather limited, I updated the database manually from what we (i.e. Dr. Norman and myself) found not working as expected and what the students marked as functionality not working properly or wanted features. This was a very useful practice to follow because it also promoted continuous feedback on changes from potential end users, thus implementing the Agile manifesto properly.

Talking about actual implementation, we have kept a custom database on GitHub in order to be always up-to-date with what is going on and allow us to spot where we lack quality and efficiency. We have made use of the custom labels GitHub already provides, among which the most used ones have been:

- *bug* – here we marked flaws in the system (or the students who have been providing constant feedback);
- *enhancement* – we added here the features most wanted by our users or by ourselves; this was refined from the requirements gatherings previously mentioned that ended up as multiple user stories eventually;
- *question* - -these were marked for research purposes - one example might be: can we change the shape of vertices of a graph dynamically as the algorithm animation makes progress?

As an example, Figure 4.6 shows how issues were continuously tracked throughout the software process. This screenshot was taken during one of the iterations.

The screenshot shows the GitHub Issues page with the following details:

- Header navigation: Code, Issues (4), Pull requests (0), Projects (0), Wiki, Pulse, Graphs, Settings.
- Search bar: Filters (is:issue is:open), Labels, Milestones, New issue button.
- Issue summary: 4 Open, 1 Closed.
- Issues listed:
 - #5 Add play/pause/next/previous buttons to the algorithm animation. enhancement. opened 18 seconds ago by nclandrei.
 - #4 Get the animated nodes for the Prim Jarnik algorithm display proper colors. bug. opened 2 minutes ago by nclandrei.
 - #3 Add upload text button for the Huffman tree. enhancement. opened 3 minutes ago by nclandrei.
 - #2 Random button for Heap is not working properly. bug. opened 3 minutes ago by nclandrei.
- Filtering and sorting options: Author, Labels, Milestones, Assignee, Sort.

Figure 4.6: Example of issue and bug tracking in the GitHub provided Issues page.

As it can be seen in the figure, there is also a milestone tab assigned to each issue. Because the screenshot is from an earlier sprint, there were not milestones attached, but since January every issue had one so that we would plan effectively and reach all goals by the negotiated deadline.

Chapter 5

Design

This chapter will cover the design decisions taken throughout the software process. It will delve into the architectural approach we followed, why the main technologies were chosen (e.g. Electron, Vis.Js) and why they proved to be the most effective to achieving our purposes. In the end, we will also discuss compromises that needed to be taken and why.

5.1 Architecture

Because we would have a wide variety of users (students, experienced developers, university lecturers, enthusiasts) one needs to take into account a range of different perspectives. Therefore, architectural design and focus on simplicity is vital to the success of every software. We have used UML diagrams in order to visualize the system as well as possible before proceeding with the implementation.

Figure 5.1 shows our final candidate for the component diagram.

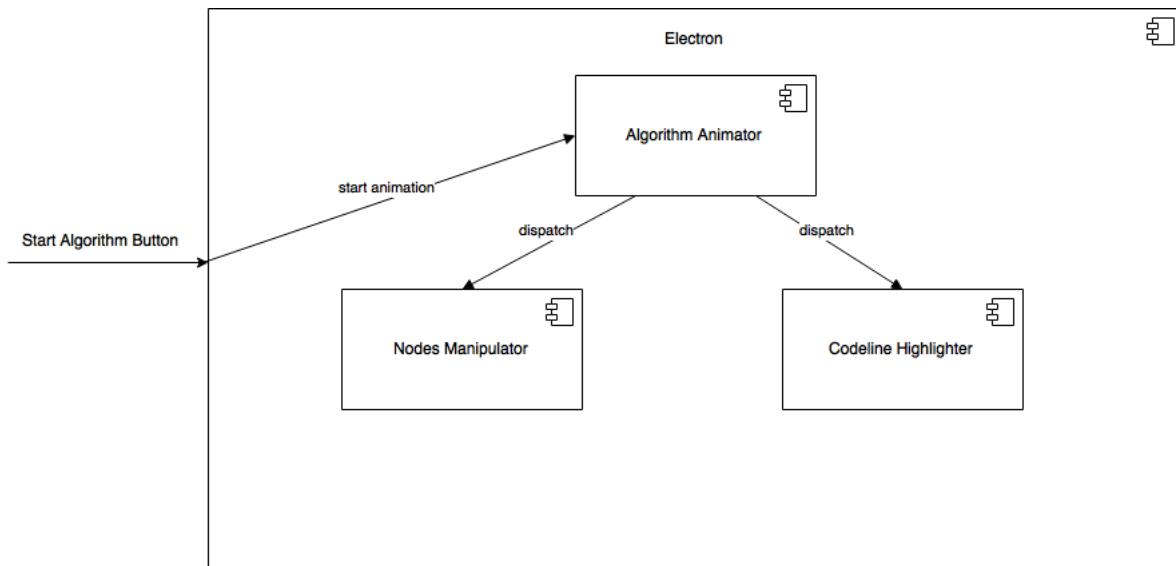


Figure 5.1: UML component diagram.

It can be noticed that our approach was rather simplistic - after the user hits the start algorithm button, the main component is the Electron window. This is, in fact, the native window generated by the framework which

Gethin:
do not go
therefore
at the start
of the sentence
Gethin: add
reference
to UML
diagrams
and say
something
about what
they were
chosen?

contains the application. As we will see in the subsequent sections, Electron uses web tools to generate the output, thus we start rendering on our HTML page the animation. This will be composed out of 2 components: the node manipulator which will colour, move, hide and change the vertices of the graph or tree, while the codeline highlighter will handle the animation of the algorithm's codelines. They will execute concomitantly and generate output immediately in the window.

Therefore, our architectural style of choice is component-based: every algorithm is a component itself and they all interact with the Electron core functionality such as windowing, copy-paste-cut, system calls etc. It emphasizes on reusability through loosely coupled independent components, thus adding simplicity to the project and making it scalable and maintainable.

Gethin:
previous sentence needs rewording and does not make sense

5.2 Application Type

This was a very important question right from the beginning. Having the experience from past years students who implemented algorithm animators, creating a Java application that could be packaged into a JAR seemed attractive. Java is the language used the most for university coursework, thus there was comfort in choosing this approach.

Also, there are many projects available that were of interest. The one we studied closely before deciding on whether to choose Java or not as the primary language behind our animator was ANIMAL [35], one of the most popular tools on the web. Even though the paper published by Rößling et al. is almost 20 years old, the animator is still actively under development, the latest release dating back to 5th of September 2016.

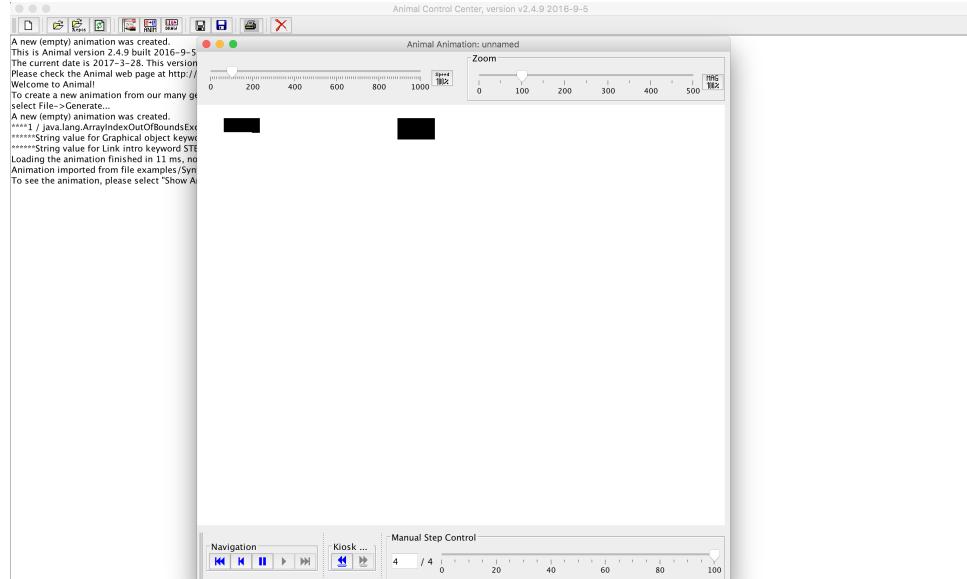


Figure 5.2: Screenshot taken while using the Animal algorithm animator.

The screenshot in Figure 5.2 shows how many options are available to the user and how powerful the application is, providing features such as the possibility to create your own algorithm, animate it the way you want to, set any colour for blocks and even draw manually.

On the other hand, the other option that we considered was making an animator based on web tools. The first and obvious approach was to create a website using the very large number of libraries for drawing on the canvas and manipulating the Document Object Model (DOM) in general. Firstly, we began by investigating how popular JavaScript was among developers and if this was feasible after all. Figure 5.3 presents a recent survey

from StackOverflow, one of the most comprehensive and useful resource repository for developers around the globe [38].

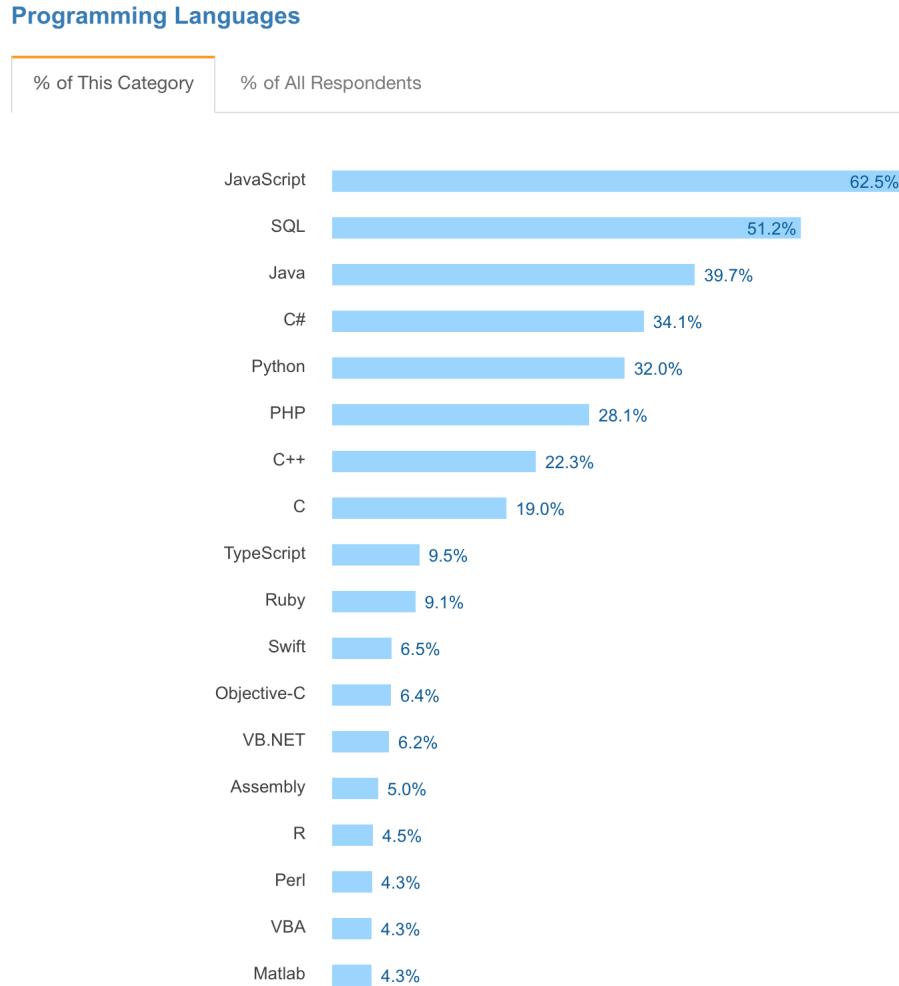


Figure 5.3: Most popular 18 programming languages in 2017.

As it can be seen, JavaScript is almost twice as popular as its competitor Java. Also, when talking about concrete JavaScript libraries, there is a considerable amount published as open source. Other advantages for this approach were:

- great reach to users as almost everyone has at least a browser installed on their machine;
- modern, state-of-the-art UI libraries that would make the experience pleasant for users.

Weighing both options, we eventually decided to proceed with the web approach because our main goal was to make users, especially students who want to learn new algorithms, enjoy using our animator. Web tools, as discussed previously, provided us with a variety of User Interface components that made our goal possible.

5.3 Prototyping

Following Agile methodologies implied that we would involve the potential customers during the whole process in order to get continuous feedback. Therefore, we developed low fidelity prototypes in the early iterations (i.e.

paper-based) and eventually high fidelity ones (mainly using Omnigraffle [30]).

Figure 5.4 presents one of the early paper drawings that could give our users an idea of how the animator might look like.

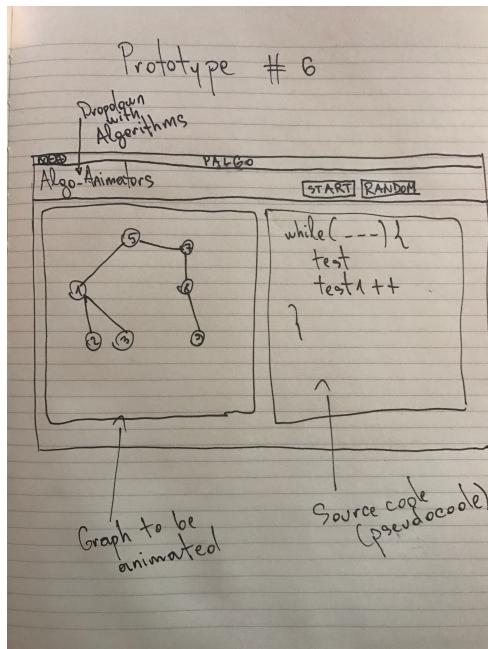


Figure 5.4: Low fidelity prototype drawn during one of the early sprints.

As you can notice, the prototype is very basic but we wanted to give the end users a rough outline of how the application will look like. Mostly, it resembles what Palgo looks like in the latest version, especially the layout of the two animation panes as well as the position of the logo.

However, as iterations passed, we started building high fidelity prototypes and we used more mature tools to wireframe such as the above-mentioned Omnigraffle. Figure 5.5 shows one of our prototypes from late January.

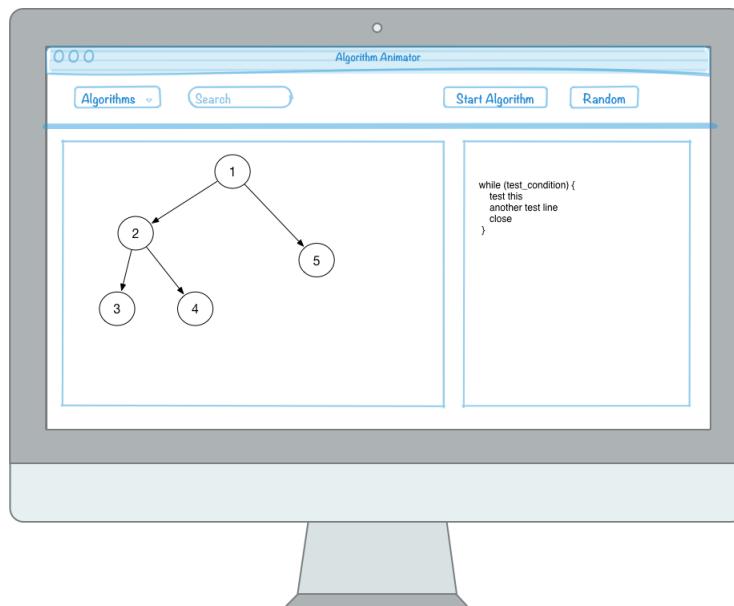


Figure 5.5: High fidelity wireframe built with Omnigraffle.

We soon afterwards started demonstrating the application itself without the need of wireframing, as the

animator began to be mature enough and provide a minimum of functionalities that could give the testers a better idea of how everything works compared to a simple image that we showed to them.

As it can be seen from the first paper prototype up to the higher fidelity ones the design we came up with was very similar to the one the application has nowadays. However, the buttons were moved from the right hand side to the left because of the users' feedback - they requested that they should be below the bar with the logo so that the UI would be cleaner with the elements more easily distinguished. Also they requested to remove the search bar as they found it useless except if the number of algorithms grew considerably and a dropdown would have ruined best UI/UX practices.

5.4 Electron

Thus, we ended up with using a web approach. There are many options available, ranging from university-specific algorithm animators to more general ones such as the previously mentioned VisuAlgo [21].

However, one very important aspect that we were concerned about was the core underlying principle of the internet, which is that it requires a permanent internet connection. What if our users would not have access to internet a couple of days before an algorithmics exam? What if our potential user would want to play around with algorithms at his/her own pace without needing to have a wired or wireless connection?

Also, another thing we had to think about was UI and UX engineering. Having maybe too many resources available, we could have ended up with choosing one that would not fit a part of our users. How can we make sure that the framework we would decide on will make all users feel familiar?

Therefore, the best approach would be to have an animator that is implemented using web tools but can function as a standalone desktop application. Thus, by being native, it would make all users feel familiar in any their operating system of choice. At this point, we had to research different alternatives to create cross-platform native applications with web tools. The one that was most popular and had the largest library of resources was Electron, which we chose in the end.

Electron is a framework built by engineers at GitHub and then released as an open source project on their platform. It is in fact a Chromium backend that serves webpages inside windows which act as native regardless of the platform of choice. Therefore, the developer can use any web tools available to develop his/her applications benefiting at the same time from features such as system calls.

Because it uses Chromium under the hood, we are automatically taking advantage of the V8 engine [18] which powers the Google Chrome browser. Efficiency was primordial to us and such we wanted to check if such a candidate would produce expected output in a reasonable time, so we checked benchmarks available for various JavaScript engines.

Figure 5.6 shows benchmarks for various JavaScript engines including jsc, Ion and V8. As we can see in the figure, V8 ranks extremely highly, being at the lowest points of execution times starting almost from its inception.

The other very important aspect was to see if Electron had enough resources, tutorials or demo applications to allow the development workflow to be smooth. We ran a query on Google Trends among other platforms to check how popular it was. Figure 5.7 presents the results.

We can clearly observe that apart from the big spike when the framework became the underlying engine behind many important desktop apps such as Slack towards the end of 2015, Electron has had a constant ascension throughout the years. Also, researching the internet for resources, we found many tutorials and demo applications from which we could draw inspiration.

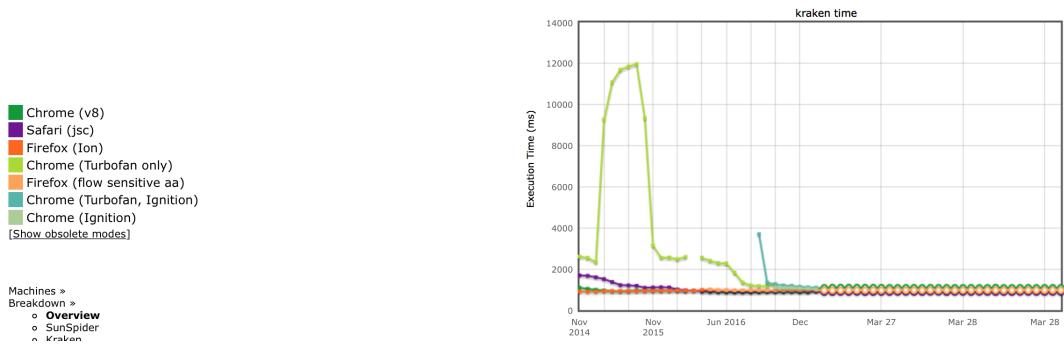


Figure 5.6: Benchmarks for various JavaScript libraries as of 28/03/2017.

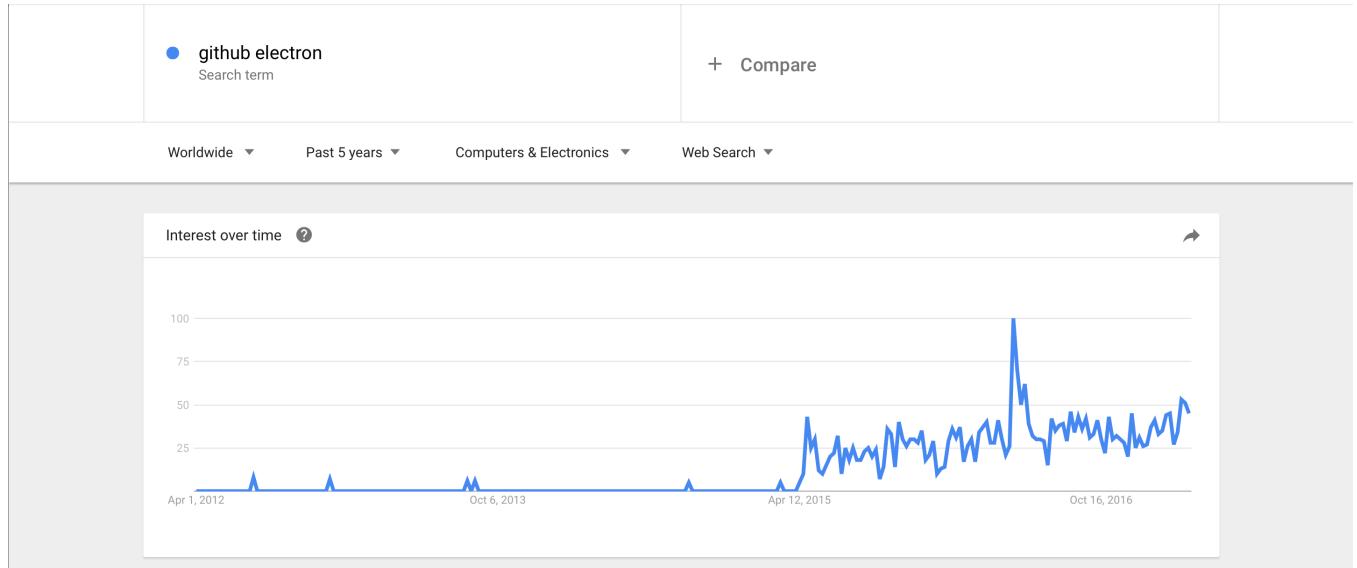


Figure 5.7: Electron trending on Google in the past 5 years.

Summing up, Electron was a natural choice after careful thinking about advantages and caveats and it proved to be a very productive framework that allowed an easy and simple workflow for the rest of the software development lifecycle.

5.5 Drawing

The component that had the same importance as Electron was the engine behind manipulating the vertices and edges of trees and graphs as trees and graphs were the focus of the algorithms to animate. Because of the large variety of JavaScript libraries available for playing with the canvas, we had a couple of choices to select from. This was a very lengthy process because many provided some features while others gave us other unique ones.

Table 5.1 presents the libraries together with their pros and cons.

As shown in the figure, the main candidates we have chosen for the drawing libraries are VisJS, Treant, D3, HTML5 Canvas and SigmaJS. Let us start with Treant. This was a very compelling option because it had a very robust and easy to use Tree API: vertices and edges could be easily manipulated, as well as the layout was very flexible to suit the developer needs. However, as it can be seen in the table above, we did not have an out-of-the-box API for graphs. Therefore, this option became unfeasible.

	Vertex Manipulation	Edges Manipulation	Tree API	Graph API	Actively Maintained	Easy Pick-Up
Treant	✓	✓	✓	x	x	✓
Vis JS	✓	✓	✓	✓	✓	✓
Sigma Js	✓	✓	✓	✓	✓	x
HTML5 Canvas	x	x	x	x	✓	x
D3	✓	✓	x	x	✓	x

Table 5.1: Table showing features provided by the main drawing libraries for trees/graphs.

HTML5 Canvas is a component defined by W3 Consortium themselves and thus it's actively under development and heavily standardized. However, apart from continuous maintenance and new regular features, it is hard to pick up and it does not provide APIs specifically designed for graphs or trees which meant we needed to put too much effort in learning a new platform rather than focusing on actual implementation of the animations.

D3 and SigmaJS are both very mature platforms that have a very large number of features for animating and drawing on the canvas. They were both viable candidates up until the end. However, the reason why we eventually chose VisJS is because it was the easiest to pick up and had out-of-the-box APIs for both graphs and trees. Thus, choosing VisJS, we saw a rapid development workflow and an emphasis on simplicity due to the fact that the tools we were using were not standing in our way, but rather helped us to quickly implement the user stories.

5.6 Animation

After drawing, the next major issue we had to discuss was the actual animation part. We have researched many tools available and we ended up with basic set timeouts. Listing 5.1 depicts a code snippet used for animating the codelines followed by a reset of the whole graph to its initial state.

```

1 setTimeout(
2   function() {
3     unHighlightAllCodeLines();
4     for (var k = 0; k < nodesArrayLength; k++) {
5       unHighlightTableCell(nodes[k].label);
6     }
7     resetWholeNetwork(network, container, options);
8   },
9   2000 + 3000 * nodesArrayLength + 13000 * nodesArrayLength - 1
10 );

```

Listing 5.1: JavaScript snippet for creating an animation.

Even though the implementation details will be covered in the following chapter, we shall discuss here the importance and the reason why we decided to use this approach instead of others. Some libraries were already built to facilitate developers an easy path to animation, so why use plain JavaScript for accomplishing this? We have reviewed and discussed advantages and disadvantages for the following JS libraries for animation:

- Velocity JS;
- DOM JS Animation;
- JQuery Animation.

Starting with Velocity JS as it is an actively developed library with more than 39 contributors on Github and almost 700 commits [42], what it primarily offers is transforms, loops, easings and colour animation. This sounds

promising, but the main problem in our case is that we are working with Vis Nodes in a tree or a graph, thus our components are JavaScript objects rendered in the canvas. Even more than that, we have circular dependencies, as the root of a tree can have children which in turn are Nodes themselves, so mixing the structure Vis JS imposes with the functionalities of Velocity JS was rather impossible.

Also, an issue with using any of the 3 options was that they are manipulating elements which are part of the DOM (Document Object Model), while in our project the elements are part of the canvas. Therefore only the canvas could be animated as a whole without being able to manipulate the individual elements inside.

Having discussed all these caveats, we were left with no other choice than devise algorithms to animate our nodes and codelines using plain setTimeout calls throughout the project. Even though it was a compromise and it slowed drastically the workflow in the beginning, once we passed the steep learning curve, we were able to animate all the elements in short periods of time.

Gethin:
the circular
dependency
is not clear
from what
you said

5.7 Material Design

We had a couple of main possibilities to choose from when designing the application's User Interface:

- Material Design by Google;
- Bootstrap by Twitter;
- Pure CSS.

There were others as well, such as Base [2] or Simple Grid [20], but these were not specifically suitable for our purposes. Therefore, we shall analyze different benefits given by the 3 above mentioned platforms and why we eventually ended up with Material Design by Google.

In Figures 5.8, 5.9 and 5.10 we shall present different design approaches for one of the most used and noticeable components to a user when they interact with an application, which is the navigation bar.



Figure 5.8: Bootstrap default navigation bar.



Figure 5.9: Material Design navigation bar.

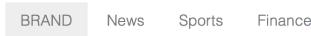


Figure 5.10: Pure CSS default navigation bar.

We can clearly notice a more colourful, natural and playful approach in the Material Design navigation bar. We have presented all 3 options to the level 4 students and they were all in favor of the Material Design UI for all components (more details on the results of these evaluations in Chapter 7, section Prototype Evaluation).

Even though the Material Design components were invented by Google (which are also available on GitHub), we have opted for the ones open sourced by Federico Zivolo [44] because they are much more popular and there are more resources with which one can get accustomed to faster.

We have decided to use this approach for several reasons:

- Material Design has gained much more traction than the other frameworks since its inception;
- even though Pure CSS does not require any JavaScript dependencies (hence the name), it is very raw and it requires a lot of tweaking before going into production;
- Bootstrap is the foundation of virtually every framework available but it is exactly that, a foundation, thus it does not give the developer advanced features out-of-the-box.

5.8 Compromises

One of the compromises mentioned previously as well was that we had to use setTimeouts to animate our objects instead of mature, easy-to-grasp libraries. What the setTimeout function does is it sets the execution of the callback function provided at a specific time in the future. Because we had only a canvas and we could not manipulate vertices as regular DOM objects, all the vertex colouring and label changing was done using a chain of setTimeouts that, at the beginning of the algorithm (i.e. when the user presses the “Start Algorithm” button), they would have their exact time calculated as to when they got executed. One example is: colour this vertex at millisecond 2600, change its left children’s label to 2 at millisecond 3500 and finally highlight the codeline “set label to 2” at the same time, 3500. This was extremely hard to understand and master and the first iterations were quite slow. However, as time passed and I gained more experience with this technique, development became much easier.

Another compromise that we needed to take was the lack of play/pause, previous and next buttons for the animator. Figure 5.11 presents a screenshot taken while using VisuAlgo.

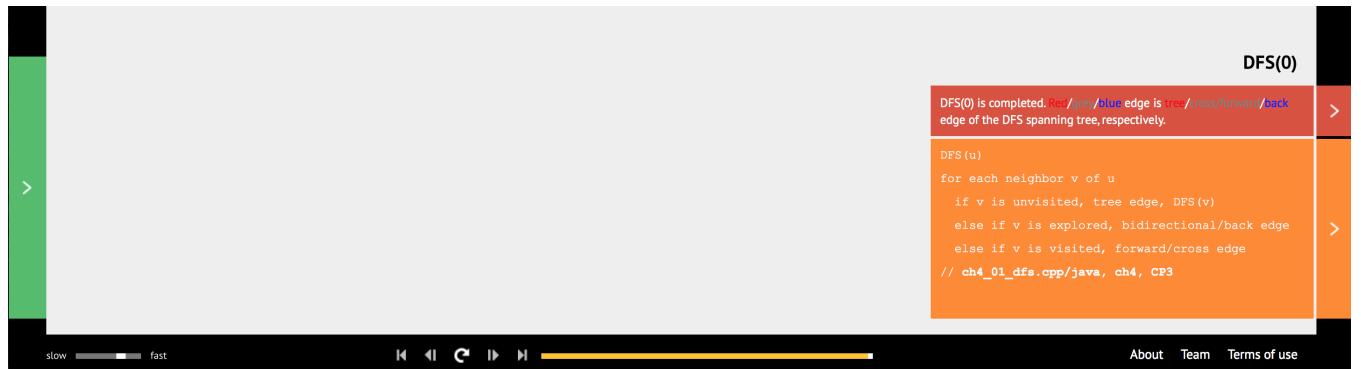


Figure 5.11: VisuAlgo play/pause/previous/next buttons for the animator.

This is a functionality desired by most of our users as well as Dr. Norman. However, due to the previous compromise we have stated (i.e. using setTimeouts for animating the algorithms), it was virtually impossible to mimmick these buttons.

Chapter 6

Implementation

In this chapter we will discuss how the project was actually implemented, how we overcame the JavaScript shortcoming of not having multi-threading support, as well as how other various functionalities were put into place. We will conclude by talking about the main issues faced and what lessons were learned.

6.1 Algorithm Selection

The algorithm selection was solely done during the supervisor meetings. We have come up with the following list of algorithms:

- Huffman tree construction;
- Heap insertion and deletion;
- Depth-First Search in a graph;
- Breadth-First Search in a graph;
- Dijkstra's Shortest Path;
- Prim-Jarnik's Minimum Spanning Tree;
- Dijkstra's Refinement of Prim-Jarnik's Minimum Spanning Tree.

These candidates were chosen because they are all among the most common and used algorithms and they are also very popular among university computer science courses. We have taken the codelines which would be animated from Dr. Norman's slides from the Algorithms I class at University of Glasgow. Moreover, we also double checked them for correctness. They were translated from either pseudocode or Java to JavaScript which implied rather simple transformations as the languages do not have such divergent syntaxes.

6.2 Project Structure

We have split our application into 3 main components, as in traditional website development: stylesheets, JavaScripts and HTML5 pages. They were all coordinated by npm which is the node package manager [29].

Essentially, npm is the equivalent of Maven or Gradle for Java projects and is both a dependency manager and a build automation tool. Therefore, in npm we defined the name of the project and all sorts of metadata as well as the dependencies needed for building the app. As soon as all these were set, we started working on the three main categories of files mentioned above. Figure 6.1 presents the simple structure we opted for.

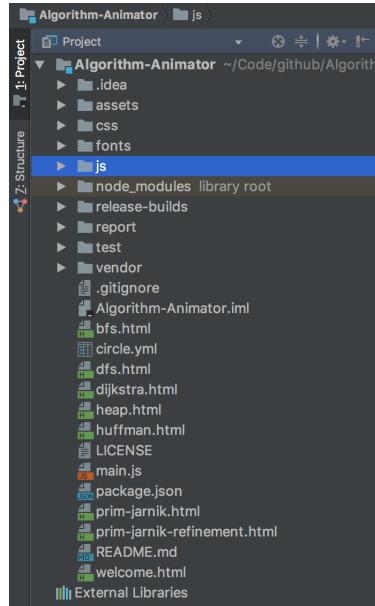


Figure 6.1: Project Structure as shown in IntelliJ IDEA.

Architecturally speaking, because we have opted for a Component Based approach, everything is built so that it can be reused. Therefore, the nodes animators for all algorithms, as well as the codeline animators, are generic and only adapted to the specific algorithm.

Also, we have also added in assets, vendor and fonts components. These are just static files that will be used to render fonts, add images to the HTML pages as well as provide some extra JS modules to the specific algorithm animators.

Last but not least, the package.json file is automated so that we can add the releases as easy as possible to the release-builds folder which will contain all executables for the 3 main operating system families: macOS, Windows and Linux.

6.3 Animation Pipeline

After the user presses the Start Algorithm button there is a pipeline of processes that run “behind the scenes”. This pipeline is displayed in the UML use case diagram shown in Figure 6.2.

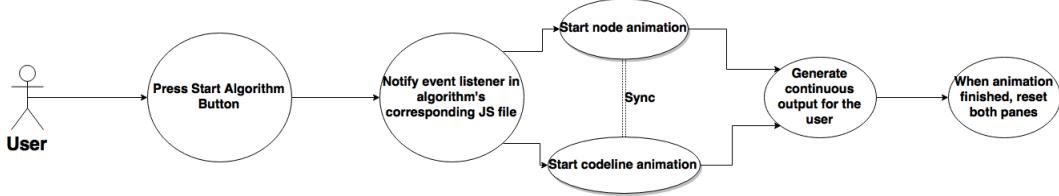


Figure 6.2: UML Use Case Diagram showing application flow of events.

As it can be seen, everything starts when the user presses the start algorithm button present on every page. From there, the event listener is set to check if the user clicks the button (in the corresponding JavaScript file) and starts the 2 animation components. The two components are always in sync being set to execute at the same time using the `setTimeout` delay functionality (e.g. both do iteration x after 1000 milliseconds multiplied by x), thus giving the user synchronous output from the nodes pane and codelines pane. When they have executed their last iteration, they both stop and the whole canvas is reset to its original state. This is done again with a `setTimeout` call with the delay set as unit of time (usually 1000 milliseconds) times the total number of steps to complete the algorithm. The number of steps is either computed automatically (e.g. in Huffman tree construction we can directly calculate the total number of steps needed to complete the animation) or through running the algorithm in the background and returning the value. As we discussed previously, the V8 engine is extremely fast thus the execution of any of our algorithms is in terms of milliseconds, the delay being indistinguishable for the human eye.

One concern might arise and that is: even though executing the algorithm an unnecessary extra time would not make the user notice any difference, why do it after all? Let's take the following example from the heap insertion algorithm (Listing 6.1).

```

1 insert item in new leaf node;
2 while (new_value not in root && new_value > parent_value) {
3     swap new_value with parent_value;
4 }
```

Listing 6.1: Pseudocode for inserting a node in a heap.

As you can see here, we have a while loop, therefore we do not have an already known number of steps until the condition fails and we exit the loop (as we would generally have in a for loop). Listing 6.2 shows a part of the actual implementation for animating an insertion in a heap.

```

1 const steps = getInsertSteps(node);
2 for (let index = 0; index < steps; index++) {
3     (function(i) {
4         highlightHeapCodeLine("insert", 2);
5         arrayOfSetTimeouts.push(setTimeout(
6             function() {
7                 if (prev) {
8                     prev.color = "#009688";
9                 }
10                prev = cursor;
11                let tempLabel = cursor.label;
12                cursor.label = cursor.parent.label;
13                cursor.parent.label = tempLabel;
14                cursor.color = "red";
15                cursor.parent.color = "red";
16                cursor = cursor.parent;
17                network = rebuildHeap(nodes, edges);
18            },
19            1000 + i * 1000
20        ));
21    })(index);
22 }
```

Listing 6.2: Actual implementation for animating the insertion of a heap node.

We can see that we are first getting the number of steps (the constant variable steps) and then executing a for loop with the test condition being that index is smaller than steps. Then, we are passing index as a parameter to the self-invoking function inside the for loop. On line 19 you can observe how we are delaying that function using the parameter *i* which in turn is index from the outer loop. Imagine that we had a while loop instead of this and we started the algorithm animation. The first step was that as the interpreter would start fetching, reading and executing the source code, it would go into an infinite loop as soon as it hit the while (test-condition) line. This would happen because it would not execute the inner code (i.e. swapping the new value with the parent value; this is because when a setTimeout is read by the interpreter, what is inside is never executed until that specific delay set by the developer), thus progressing towards meeting the condition and breaking the loop. However, in a for loop *i* is incremented until it breaks the test condition and the loop is exited.

On the other hand, we have implemented algorithms that do not require an extra execution, such as Dijkstra's Shortest Path. There we would just call the for loop directly and animate the 2 panes synchronously.

6.4 JavaScript Multi-Threading

This was the most important issue faced throughout the whole software process. Because I did not have previous experience with web applications that might require multi-threading support, I did not know if JavaScript provides this or not. In Listing 6.3 we provide a code snippet first to better describe the problem and discuss on it afterwards.

```

1 (function(ind) {
2     setTimeout(
3         function() {
4             unHighlightAllCodeLines();
5             highlightCodeLine(1);
6             if (nodes[ind] === nodeRoot) {
7                 distances[nodes[ind].label] = 0;
8             } else if (containsObject(nodes[ind], nodeRoot.adjacencyList)) {
9                 distances[nodes[ind].label] = getEdgeWeight(nodeRoot, nodes[ind]);
10            } else {
11                distances[nodes[ind].label] = Number.POSITIVE_INFINITY;
12            }
13            appendRowToTable(nodes[ind].label);
14            setupDistance(nodes[ind].label, distances[nodes[ind].label]);
15            if (ind > 0) {
16                if (!containsObject(nodes[ind - 1], S)) {
17                    nodes[ind - 1].color = "#009688";
18                } else {
19                    nodes[ind - 1].color = "#3f51b5";
20                }
21                unHighlightTableRow(nodes[ind - 1].label);
22            }
23            nodes[ind].color = "red";
24            highlightTableRow(nodes[ind].label);
25            network = rebuildNetwork(network, container, options, nodes);
26        }, 2000 + 3000 * ind);
27 })(i);

```

Listing 6.3: Animation function used in Dijkstra's SP algorithm.

As you can see in the code snippet, this is a very complicated function which sets up initial distances to all vertices in a graph from the root - if they are adjacent, then the distance will be the weight of the edge; otherwise, the distance will be set to infinity. SetTimeout is a function that delays another function's execution until a specific point in time, thus we are sequencing multiple calls to this self-contained function based on the *i* index which is the variable used in the loop where this piece of code is defined.

This is the way all algorithm animations are implemented. However, the problem is as you can notice on lines 4, 5 or 21 that we are also animating other components apart from the nodes in the same body. In other programming languages, such as Java, this could easily be done by using 2 threads which would split the work between themselves - one would deal with node animation, while the other could highlight and unhighlight codelines dynamically. However, JavaScript does not allow support for multi-threading so we had to resume to merging the 2 functionalities into one. Even though one might argue that along with HTML5 the W3C introduced Web Workers [28], these do not provide a solution to our problem because they allow only primitives and simple objects to be passed inside the running thread, while we have Nodes that can have other Nodes as children (which is a property of the Vis Node class), thus creating circular dependencies.

Therefore, this sort of complex arithmetic had to be performed (see line 27) in order to make sure the calls are invoked at the correct times and that they do not overlap.

Moreover, another type of issue occurred when there were setTimeout invocations inside outer timeouts.

One particular problem we faced was that if you call a setTimeout function inside another setTimeout, the time you set the inner one to execute at is relative to the time the outer one will be invoked. Listing 6.4 provides an example for a better visualization.

```

1  setTimeout(
2    function() {
3      unHighlightHeapCodeLine("delete", 1);
4      highlightHeapCodeLine("delete", 2);
5      impose(nodes[0]);
6    },
7    3000
8  );
9 // impose function starts here
10 for (let index = 0; index < steps; index++) {
11   (function(i) {
12     let prev = null;
13     highlightHeapCodeLine("impose", 1);
14     setTimeout(
15       function() {
16         if (prev) {
17           prev.color = "#009688";
18         }
19         prev = cursor;
20         let largerValue;
21         if (cursor.children.length === 1) {
22           largerValue = cursor.children[0];
23         } else {
24           if (cursor.children[0].label > cursor.children[1].label) {
25             largerValue = cursor.children[0];
26           } else {
27             largerValue = cursor.children[1];
28           }
29         }
30         cursor.color = "red";
31         largerValue.color = "red";
32         let temp = cursor.label;
33         cursor.label = largerValue.label;
34         largerValue.label = temp;
35         cursor = largerValue;
36         network = rebuildHeap(nodes, edges);
37       },
38       1000 * i
39     );
40   })(index);
41 }
```

Gethin: an
you sum
you mea
line 27?

Listing 6.4: Heap animation that uses nested setTimeouts.

The for loop below the comment is only a part of the impose function, but for shortening the length as much as possible we have included what was necessary to demonstrate the issue mentioned above.

As it can be observed, we have the call to the impose function (line 5) inside a setTimeout. Thus, that will execute after 3000 milliseconds, but inside the for loop every iteration will be invoked at 1000 milliseconds times index. Because of a lack of tutorials and resources on the web regarding such possible nested setTimeouts, I believed for a long time that the $1000 * i$ should be replaced with $3000 + 1000 * i$, but this was a big mistake because 3000 is inherited and it does not need to be included in the nested loop. After I realized that inner setTimeout delays are relative to the corresponding outer setTimeouts (e.g. outer setTimeout executes every 3000 milliseconds and the inner one executes every $3000 + \text{the delay it is set to}$, but we do not need to specify the 3000 milliseconds in the delay parameter as it is inherited from the parent setTimeout; thus, in the inner setTimeout, `setTimeout(..., 3000 + x)` becomes `setTimeout(..., x)`), the timings would not overlap and the animation would perform as required.

6.5 Extra Features

Both the potential users and the supervisor have requested that all algorithms should contain a random functionality could easily have a graph or tree to animate that did not require manual insertion and deletion. Therefore, every algorithm page has a random button which will automate the whole process of creating a Vis network.

Also, for the Huffman encoding algorithm, we have also added an Input File button that can take a plain text file from the user's machine and, after parsing the document, execute the algorithm against that input.

Moreover, another feature implemented that was very popular was the landing page in the application - as the user opens the animator they will be presented with a carousel where they can interactively select what algorithm to see animated. Thus, we needed to place emphasis on this rather simple stylistic element so that users would enjoy the application since they opened it for the first time. Figure 6.3 depicts a screenshot of the landing point.



Figure 6.3: Landing page in the application.

Last but not least, we believe that having a modern and attractive webpage where users can download the application and get to know some basic information is essential, thus we have created a custom website where we host the releases and users can simply download and run Palgo with only a couple of clicks. This was a very nice feature voted for by many of our end users. Figure 6.4 shows a screenshot of the website.

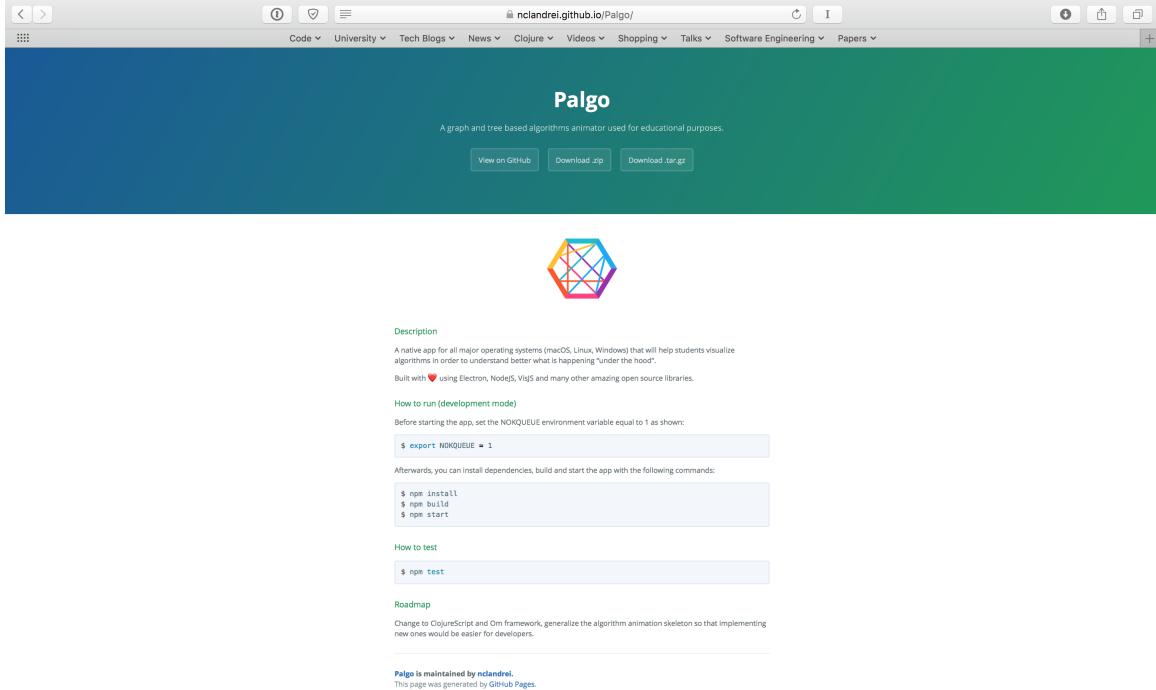


Figure 6.4: Palgo website hosted by GitHub.

6.6 Lessons Learned

One of the most important lessons learned was that web development, even though it might sound easy, is not like that at all. Because everything around us nowadays is web-based or it has something to do with it, there are a lot of technologies in various sub-fields that help developers solve certain problems. As discussed in the previous sections, we have struggled with many design decisions mainly because of the large palette of choices. Therefore, having to research and eventually learn many of them so that argumented comparisons could be made regarding which one is better was time consuming. However, once the developer possesses the basics, they can evolve and apply the fundamentals to learn any new technology with ease.

Another lesson learned was that one needs to take very good care to not fall into the trap of “premature optimization”. As we started the project back in first semester, I was very keen on implementing a large number of UI features such as stylish buttons or attractive fade-ins when the window content changed. However, these early iterations were designed for setting up the basics and implementing the algorithms, as well as animate them. Learning that optimizing prematurely can lead to wasted effort, I have changed my approach and started prioritizing our user stories more effectively.

A third lesson I learned was that any JavaScript can fall in the “callback hell”. This is such a common pattern among web developers that people created a website that specifically addresses this issue and gives advice on how to avoid it [22]. I have also fell in this trap but with advice taken from the website mentioned previously, I could easily refactor and follow the best practices promoted by industry leading developers.

Another vital aspect I have learned throughout this software process was the importance of code style in

JavaScript. Even though other programming languages promote a robust and consistent code style as well (e.g. Java, C), JavaScript is the most popular technology used, and thus it has a very widespread reach. Therefore, many styling guides have been created so that new developers would find code easier to read and, subsequently, to understand. The format I have used throughout the whole project is a tool called Prettier [34] and the guide I used for reference is the one Airbnb have open sourced on GitHub (it is now one of the most popular projects on the platform with almost 50,000 stars [1]). This will ensure that if the project gains contributors in the future, then their accommodation period with the codebase will be smooth and fast.

Overall, I believe that this project was a real success and it was a great way to both learn new technologies as well as learn how to research into previous work and apply the best software engineering practices. I think that without following set rules, deadlines and good methodologies already proven successful in the industry, no project can ever succeed.

Chapter 7

Testing

In this chapter we will discuss how we have tested our application to conform to the highest quality standards, how we used CircleCI mixed with GitHub to do integration testing, as well as how we evaluated our application at the end of our project and what results we achieved.

7.1 Unit Testing

Unit testing is a part of software testing in general that deals with individual parts of the source code are tested to check if they perform as expected.

JavaScript provides a large pool of frameworks to do unit testing. Even though they are quite different than what I was used to (e.g. JUnit, Mockito, PyUnit) from university coursework and work experience, the fundamentals stay exactly the same.

For our purposes we decided that MochaJS [26] was a great fit for the following reasons:

- flexible and implemented with simplicity in mind;
- runs on node.js which was already one of our powering engines, thus the integration was straightforward;
- promotes a very descriptive documentation when writing tests because it uses the “describe” and “it” keyword which let the developer spot the failures easily;
- can integrate with various other frameworks such as should.js [36], better-assert [4] and chai [6].

Through MochaJS we tested various animation and UI related components such as do the buttons work properly, are algorithms correctly implemented in JS and give correct output or are codelines highlighted properly. However, in order to test Electron itself and see if the native features behave as expected we hooked up another component: Spectron [17]. It is a testing framework built by GitHub that uses ChromeDriver and WebDriverIO.

As we discussed previously, using Mocha was very similar to what Java unit testing frameworks provide, but Spectron is quite different. Listing 7.1 is an example of one unit test written using Spectron.

```

1 var Application = require('spectron').Application
2 var assert = require('assert')
3
4 describe('application launch', function () {
5   this.timeout(10000)
6
7   beforeEach(function () {
8     this.app = new Application({
9       path: 'release-builds/macOS/Palgo'
10    })
11    return this.app.start()
12  })
13  afterEach(function () {
14    if (this.app && this.app.isRunning()) {
15      return this.app.stop()
16    }
17  })
18  it('shows start screen', function () {
19    return this.app.client.getWindowCount().then(function (count) {
20      assert.equal(count, 1)
21    })
22  })
23})

```

Listing 7.1: Spectron unit test which checks correct behaviour of the Electron window.

This is a simple unit test that starts the executable on macOS and checks if the window actually starts or not - basically we are running a smoke test. Spectron was used mainly for checking if the application starts properly, closes gracefully and also if a regular user workflow would break it.

Moreover, Spectron documentation promotes using MochaJS as the main testing framework to integrate with so our accommodation with the two was natural.

7.2 Integration Testing

In terms of integration testing we have configured a `circle.yml` file which is required by CircleCI where we specified that we want mocha to be our testing framework as well as the exact command how to run all the tests before merging the code into master. The pipeline we have set up is as follows:

- start running all the tests by running the mocha command (mocha was installed on the Linux server we have hooked up with CircleCI);
- if all tests passes, merge it into master, else (i.e. at least one test fails) abort merge and send an email specifying what tests failed and why;
- publish statistics on the CircleCI website and a small indicator next to the latest commit on the GitHub page showing the status of the build (i.e. green if passed, red if it did not).

This whole process ensured a smooth workflow and a continuous feedback on the status of our application and the changes we were making.

7.3 Prototype Evaluation

The evaluation process was undergone together with 16 Level 4 Computer Science students at University of Glasgow. We have asked them to complete a short form after using the application for approximately 20 minutes.

We provided the participants with a consent form where they were assured their whole input would be anonymous. Also we have told them that they could stop at any time if they felt uncomfortable or any other reason during the process.

The participants were split into 2 groups of 8 people each and the first group was told to use Palgo for 10 minutes while the other group used Galant [40]. We chose Galant because it is one of the most recently implemented algorithm animators and it is also under active development on GitHub [39].

The set of tasks they had to complete in both Palgo and Galant were:

- start Palgo or Galant;
- select Depth-First Search algorithm;
- built a graph of 5 nodes, each being adjacent with at least another one;
- press start algorithm button;
- after it finished, press random button and observe the animations again;
- select Algorithms dropdown and go to Dijkstra's Shortest Path algorithm;
- repeat steps 3, 4 and 5;
- finally close the application through shortcut depending on the OS of choice or via the UI button;

After each group completed the tasks they switched with their colleagues and performed all the above steps for the other animator. At the end of the evaluation trial we ran a questionnaire (a Google form) regarding how they found the two applications overall, what did they like/dislike and what they thought could be improved in Palgo.

We have used the best approaches in designing the questionnaire by getting expert advice from Dr. Oppenheim's book "Questionnaire Design, Interviewing and Attitude Measurement" [31]. Thus, one of the techniques we have used was that we have mixed possible responses for multiple choice questions such that some begin with "Much better" while others begin with "Worst". Also, we have kept the questionnaire short so that the users would not get exhausted after already completing 20 minutes of testing the 2 applications. Overall, everyone seemed satisfied with our evaluation and no one experienced any problems throughout the process.

We have collected all results via Google Forms' built-in functionality and they will be presented in the next section. The full questionnaire can be found in the Appendix.

Gethin:
make sure
you have
added the
form to the
appendix

7.4 Results

The results were astonishing as I did not believe that the application would receive such great impressions from the participants and even desire to contribute to the project. The first question we selected was: "How did you find using Palgo?". Figure 7.1 shows the feedback we received.

How did you find using Palgo? (16 responses)

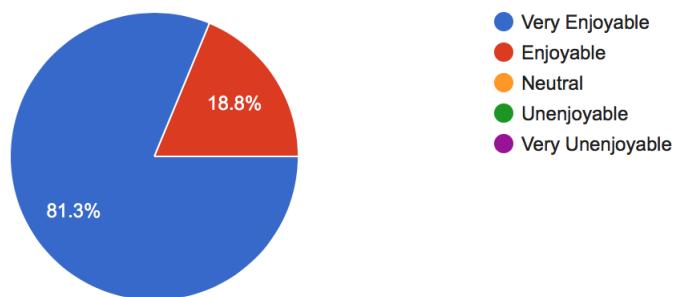


Figure 7.1: Feedback on question: How did you find using Palgo?

As it can be observed, all responses were either “Enjoyable” or “Very Enjoyable”. This was an important achievement for our project because it showed how focus on simplicity, efficiency and state-of-the-art software engineering practices can produce an application of high quality that would indeed be enjoyed by end users. Even though most of the questions were related specifically to our implementation of the animator, we also introduced Galant as a point of reference and asked our participants to compare the two. The results were even more surprising as depicted in Figure 7.2.

Which application would you recommend to a friend? (16 responses)

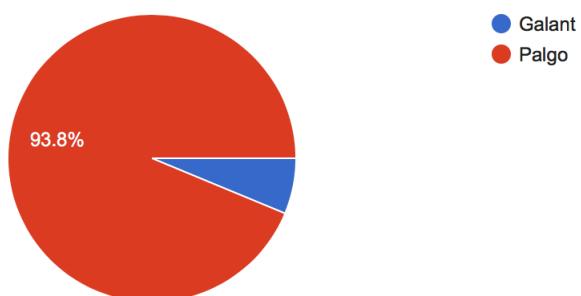


Figure 7.2: Feedback on question: Which application would you recommend to a friend?

15 out of 16 participants preferred Palgo over Galant even though the latter is a tool being actively under development for almost 3 years.

Summing up our findings, we can happily say that the evaluation proved that our project has been a real success with an actual potential of becoming developed by more than one contributor and used by many more.

Chapter 8

Conclusions

In this chapter we will wrap up this document with why we want to keep the application open source and try to promote it so that there would be more contributors, future work and the roadmap we have in mind, some final thoughts and, last but not least, acknowledgements.

8.1 Open Source

Open Source is one of the main engines behind the whole software world. Virtually every developer uses open source projects in their day-to-day workflow and many of them contribute to this amazing community. Without Electron, Vis.JS, MochaJS and many other technologies, Palgo would not exist today so we could not be more grateful to the great community that produced all this high quality software.

Therefore, we would definitely proceed with hosting the project on GitHub and start to promote it on various platforms (e.g. social media, email within university students) so that more people would use it and, even better, contribute to improving it.

8.2 Project Roadmap

Future work is vital for the project as the application is yet to be fully finished. We have divided these plans into the following categories.

- Change the JavaScript code into ClojureScript [10] - this is because ClojureScript is an amazing functional programming language which puts emphasis on immutability and a pure functional style; we believe that this would benefit us in the long term through a more concise and simple approach to implementing new algorithms.
- Modify the codebase to include React [15] - React is a technology built by Facebook which basically adds a new virtual DOM (hooks it at some point on the actual DOM) and focuses on reusability of components. Moreover, it uses JSX [24] which is a mixture of JavaScript and HTML5, thus you can create your whole page through pure JS;
- As previously mentioned, we will keep the project open source on GitHub and try to add more collaborators so that people can contribute to Palgo and add in both new features and algorithms to be animated.

- Even though we have mentioned in Chapter 2 about related work to more complex algorithms (e.g. parallel computing) and how there is virtually no animator that can show such algorithms, we will try to explore the opportunity and see if we can actually add them to Palgo in the future.
- I have already began negotiations with some Computer Science teachers from home (Romania) to consider adding Palgo as a teaching tool in their classes; if there will be people who agree to the idea we will add Romanian localization and maintain both languages from that point onwards.
- Having discussed with Dr. Norman about possible outcomes for Palgo we have also talked about the idea of proposing the application as a starting point for students that will take the “Algorithm Animator” project next year; this way we can make sure we will have collaborators to help us continue our work.

Because I have grown to be attached to the application the last point that the reader needs to get from this section is that the project will definitely not be thrown away and we will keep developing it and make it even a better tool to help people understand algorithms easily while also having fun.

8.3 Final Thoughts

I believe that the Level 4 Project was an amazing opportunity to not only learn new technologies, but also apply software engineering practices in real world. I have learned how to use many tools that I did not know about beforehand such as MochaJS or Electron. Also, I have researched and found out best approaches to write clean and simple code that would be easy to maintain and scale.

Moreover, applying what we have learned both last year in Professional Software Development and this year in Advanced Software Engineering Practices was both fun and hard to master. Theoretically speaking, nothing seems hard, but when you put it in practice you realize you did not know that much about the techniques. This was a valuable lesson I was given during this project.

In the end, I do believe that Palgo will not “die” because I will begin to promote it and continue to develop it even further. Also, after discussions with Dr. Norman, the project will be included as a resource on the Moodle page for Algorithmics I students, thus its reach will expand from that moment onwards.

8.4 Acknowledgements

To begin with, I would like to thank Dr. Gethin Norman who was not only my supervisor but also a great team member. He has helped me with everything throughout the whole year and without his input and help Palgo would not exist today.

Furthermore, I’d like to say thank you to every student who has participated during “customer” meetings during the year as well as to all the people who helped us with the evaluation at the end of the software process.

Last but not least, I would like to thank every developer that has contributed even one source code line to the amazing open source projects we have used throughout our algorithm animator. Without them Palgo would not look and perform the way it does in the final release.

Bibliography

- [1] Airbnb. Airbnb JavaScript Code Style Guide. <https://github.com/airbnb/javascript>.
- [2] Base. Base Website. <http://getbase.org>.
- [3] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.
- [4] better assert. better-assert GitHub Page. <https://github.com/tj/better-assert>.
- [5] Grady Booch. *Object Oriented Design: With Applications*. Benjamin/Cummings, 1991.
- [6] ChaiJS. ChaiJS Website. <http://chaijs.com>.
- [7] Bharat Choudhary and Shanu K Rakesh. An Approach using Agile Method for Software Development. *Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, 2016.
- [8] CircleCI. CircleCI Website. <https://circleci.com>.
- [9] Dai Clegg and Richard Barker. *Case method fast-track: a RAD approach*. Addison-Wesley Longman Publishing Co., Inc, 1994.
- [10] ClojureScript. ClojureScript Website. <https://clojurescript.org>.
- [11] Connextra. Connextra User Story 2001: ConnextraStoryCard. https://en.wikipedia.org/wiki/User_story#cite_note-connextra-4.
- [12] Oxford Dictionaries. *Oxford English Dictionary*. OUP Oxford, 2012.
- [13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [14] Electron. Electron Website. <https://electron.atom.io>.
- [15] Facebook. React Website. <https://facebook.github.io/react>.
- [16] David Galles. David Galles Algorithm Animator. <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>.
- [17] GitHub. Spectron Website. <https://electron.atom.io/spectron/>.
- [18] Google. Google Chrome V8 Engine Website. <https://developers.google.com/v8/>.
- [19] James Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Renaissance Software Consulting*, 2002.
- [20] Simple Grid. Simple Grid Website. <http://thisisdallas.github.io/Simple-Grid>.
- [21] Steven et al. Halim. VisuAlgo Website. <http://visualgo.net>.

- [22] Callback Hell. Callback Hell Website. <http://callbackhell.com>.
- [23] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages and Computing*, 13:259–290, 2002.
- [24] JSX. JSX Website. <https://jsx.github.io>.
- [25] Mike Mannion and Barry Keppence. SMART Requirements. *ACM SIGSOFT Software Engineering Notes*, 20(2):42–47, 1995.
- [26] MochaJS. MochaJS Website. <https://mochajs.org>.
- [27] John Morris. A toolkit for algorithm animation. *Proceedings AEESEAP*, 2004.
- [28] Mozilla. Mozilla Developer Network - Web Workers. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.
- [29] NPM. NPM Website. <http://npmjs.com>.
- [30] Omnigroup. Omnipage Website. <https://www.omnigroup.com/omnipage>.
- [31] Abraham Naftali Oppenheim. *Questionnaire design, interviewing and attitude measurement*. Bloomsbury Publishing, 2000.
- [32] Susan Palmiter and Jay Elkerton. An evaluation of animated demonstrations of learning computer-based tasks. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1991.
- [33] Brenda Parker and Ian Mitchell. Effective methods for learning: a study in visualization. *Journal of Computing Sciences in Colleges*, 22(2):176–182, 2006.
- [34] Prettier. Prettier Github Page. <https://github.com/prettier/prettier>.
- [35] Guido Rößling, Markus Schüer, and Bernd Freisleben. The ANIMAL algorithm animation tool. *ACM SIGCSE Bulletin*, 32(3), 2000.
- [36] ShouldJS. ShouldJS GitHub Page. <https://github.com/shouldjs/should.js>.
- [37] Ian Sommerville. *Software Engineering*. Pearson, 2015.
- [38] StackOverflow. StackOverflow 2017 Survey. <http://stackoverflow.com/insights/survey/2017#technology>.
- [39] Matt Stallman. Galant GitHub Page. <https://github.com/mfms-ncsu/galant>.
- [40] Matthias F Stallmann. Algorithm Animation with Galant. *IEEE Computer Graphics and Applications*, 37(1), 2017.
- [41] Trello. Trello Website. <https://trello.com>.
- [42] Velocity.js. Velocity.js Website. <http://velocityjs.org>.
- [43] VisJS. VisJS Website. <http://visjs.org>.
- [44] Federico Zivolo. Material Design Bootstrap Website. <http://fezvrasta.github.io/bootstrap-material-design>.