
University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

# Animating Network Flow

**Lucy MacPhail 2183332m**

February 2019

# Abstract

The Ford-Fulkerson algorithm, used to calculate the maximum amount of flow over a network, is complex and often difficult to understand through textual means alone. This project was to create an animation of the Ford-Fulkerson algorithm for learning or teaching purposes. The animation was presented as a web application, designed to give the user as much control as possible over both the network and the playback controls to aid their understanding. When evaluated on participants having taken the Algorithmics II course, every one responded that their understanding had improved after watching the animation.

# Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Lucy MacPhail Date: 26 March 2019

# Contents

# 1 Introduction

The Ford-Fulkerson algorithm is a network flow algorithm which calculates the maximum flow through a network in an iterative process.

## 1.1 Motivation

A network here could refer to pipes, roads, wires, or anything used to transport resources. All of these edges that make up a network would have a maximum capacity for the resource being transported, such as diameter or current. Being able to determine the maximum flow of a network makes it as efficient as possible. Therefore, the application of network flow algorithms are incredibly widespread and crucial to resource allocation.

Concepts of algorithms are rarely discussed using words alone. Being able to give meaning to the words using diagrams and symbolism is a natural way to communicate these ideas. Is it easier to describe a graph to someone with words, or draw it? Animating an algorithm takes this a step further, not only visualising the objects in the algorithm, but showing the effect that the algorithm has on them as it is executed.

## 1.2 Aims

The aim of this project is to use the medium of animation to convey the Ford-Fulkerson algorithm, which determines the maximum flow of a network. The final product should be easily accessible, and be suitable for use in the context of a lecture, or revision of the algorithm. Not only should the information regarding the algorithm cover all possible instances of the algorithm, but a strong focus on usability should be taken to ensure that the application communicates effectively.

## 1.3 Overview

Chapter 2 will cover the basics of the Ford-Fulkerson algorithm, and evaluate some research into the effectiveness of animating algorithms as an educational tool. Research into similar products and applications is detailed in chapter 3, finishing with a list of requirements for the product. Chapter 4 shows the design process of the layout and method of animation. Chapter 5 will go into the details of implementing this design, including the technology used and different stages of the algorithm implementation. Chapter 6 covers the ways in which the product was evaluated, both in the correctness of the algorithm and the usability of the application. Finally, chapter 7 will summarise the project as a whole and cover its successes, failures, and possibilities for the future.

# 2 Background

## 2.1 Network Flow

A network is a directed graph $G = (V, E)$. There are vertices $S$ and $T \in V$. $S$ is the source, and has no incoming edges. $T$ is the sink, and has no outgoing edges. Each edge in $E$ has a capacity, which is non-negative.

A flow through a network $G$ must satisfy the following constraints:

- **Capacity constraint**: for every edge, $0 \leq flow \leq capacity$
- **Flow conservation constraint**: for every vertex other than $S$ and $T$, the total incoming flow must be equal to the total outgoing flow.

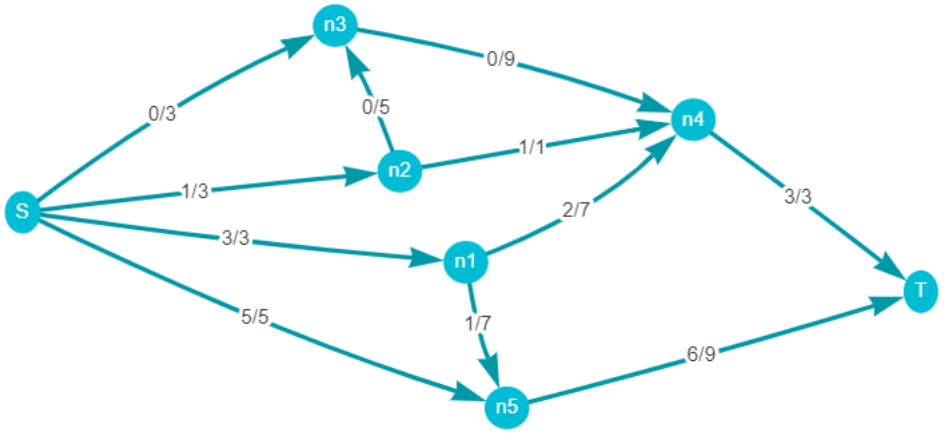



*Figure 2.1: An example of a network, with a flow of 9. Each edge has a label of the format $x/y$, where $x$ is the flow through the edge and $y$ is the capacity of the edge.*

### 2.1.1 Augmenting path

An augmenting path is a path from $S$ to $T$, comprised of the edges of $G$ but not necessarily directed as in $G$. Each edge $(u, v)$ in the path must satisfy one of the two following conditions:

1. $(u, v) \in E$, and $f(u, v) < c(u, v)$. This is a forward edge, and the difference $c(u, v) - f(u, v)$ is referred to as the slack of $(u, v)$
2. $(v, u) \in E$, and $f(v, u) > 0$. This is a backward edge.

The **Augmenting Path Theorem** states that a flow along a network is maximum if and only if the network admits no augmenting path.

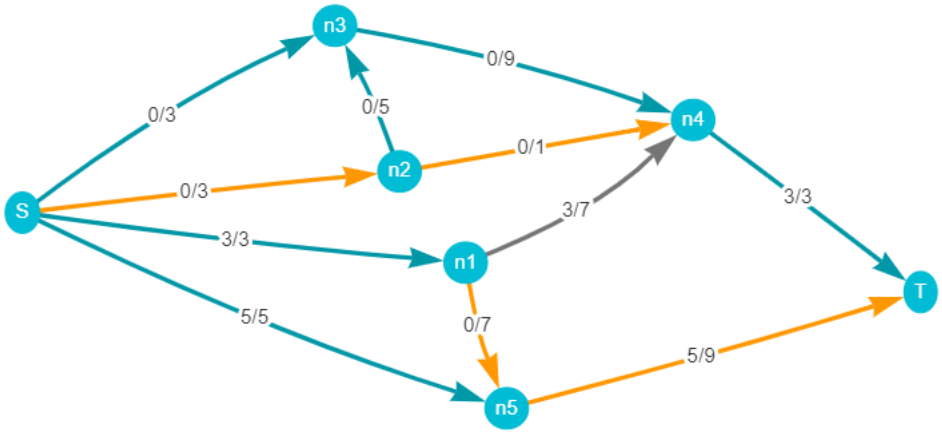



***Figure 2.2:*** *An augmenting path through the graph shown in Figure 2.1. Forwards edge are shown in orange, and backwards edges in grey.*

### 2.1.2 Residual graph

We have that $G = (V, E)$ is a network with capacity function $c$, and $f$ is a flow through $G$. Then the residual graph $G' = (V', E')$ with respect to $G$ is a directed graph with capacity function $c'$. $G'$ has the same set of vertices as $G$, $V' = V$.

$(u, v) \in E'$ if and only if:

- $(u, v) \in E$ and $f(u, v) < c(u, v)$. In this case, $c'(u, v) = c(u, v) - f(u, v)$. This is a **forwards edge.**
- $(v, u) \in E$ and $f(v, u) > 0$. In this case, $c'(u, v) = f(v, u)$. This is a **backwards edge.**

A directed path through the residual graph $G'$ from $S$ to $T$ corresponds to an augmenting path in $G$.

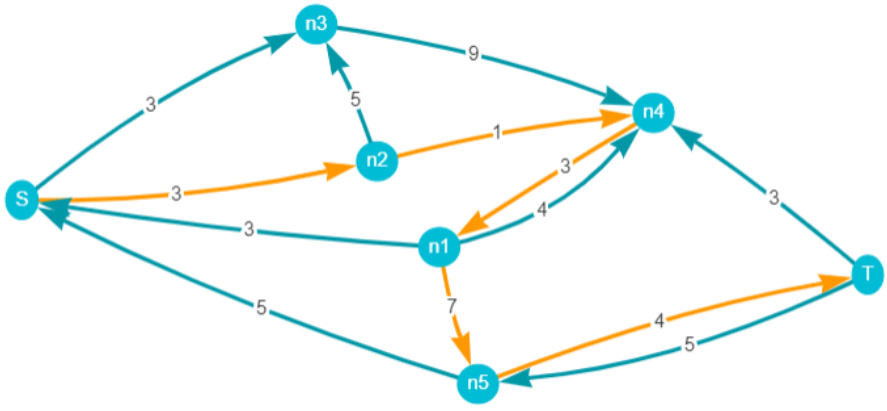



***Figure 2.3:*** *The residual graph of the network in Figure 2.1. The path from S to T corresponds to the augmenting path from Figure 2.2.*

### 2.1.3 Minimum cuts

A cut is a set of edges which separate $S$ and $T$. The *capacity* of a cut is the sum of the capacities of all the edges in the cut. A **minimum cut** is a cut for which the total capacity is the minimum

of all possible cuts.

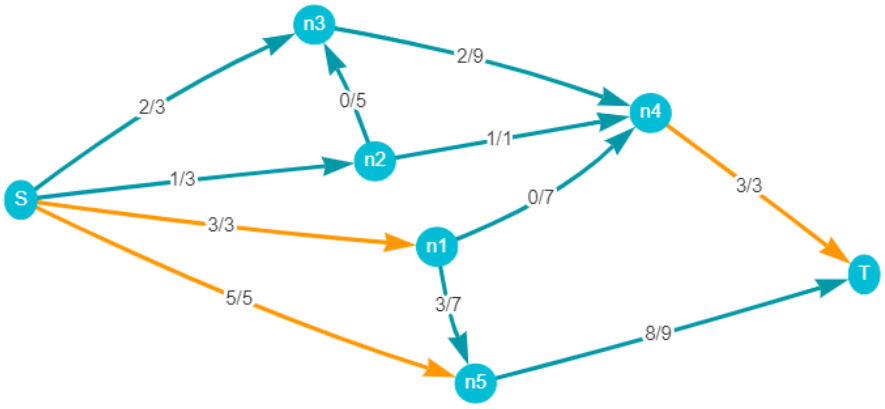



***Figure 2.4:*** *The network with a minimum cut highlighted.*

**The Max Flow - Min Cut theorem** states that the maximum flow of a network is equal to the capacity of a minimum cut of the network.

### 2.1.4 Ford-Fulkerson

Below is a version of pseudo-code for the Ford-Fulkerson algorithm.

```
For((u,v) ∈ E) f(u,v) = 0;
build residual graph G'=(V',E');
While(there is a path P from S to T in G'){
   m = minimum (c(u,v) - f(u,v)) of P;
   for((u,v) ∈ P){
      if((u,v) ∈ E) {  // (u,v) is a forwards edge
         f(u,v) += m;
      } else if ((v,u) ∈ E){  // (u,v) is a backwards edge
         f(v,u) -= m;
      }
   }
   update residual graph;
}
Maximum flow of graph has been reached
```

The algorithm continually checks for an augmenting path in the network using the residual graph. If one cannot be found, then by the augmenting path theorem the maximum flow has been reached. If the path finding algorithm is a Breadth-First search, then the algorithm becomes the Edmonds-Karp algorithm, which is identical to Ford-Fulkerson except that the path finding stage is defined.

## 2.2 Algorithm Animation

Teaching algorithms using animation is an idea that has been discussed for several decades. As computational power and display has improved, so has the complexity of the animations and the

algorithms used in them.

The effectiveness of using animation as a teaching tool for programming has also been the subject of several studies. One such study attempted to teach the Depth-First search path finding algorithm to students not studying computer science (3). The students were shown a lecture detailing the algorithm, and split into two groups. One group was given 10 minutes to read a three-page text describing the algorithm, and the other was given 5 minutes to read the same text, then shown an animation of the algorithm. Although results for predicting the algorithm proved similar between the two groups, the study showed that the group exposed to the animation were able to learn the algorithm faster and score higher when asked more difficult questions related to the algorithm. However when the algorithm was changed to a more complicated one of the binomial heap, the results were much more varied. The study concludes that there is an unreliable benefit to animating algorithms, and that the animation alone is not enough. In order to be effective, the design and presentation of the animation must be carefully considered.

This is also confirmed by a study looking into the role of animations in computer science education (2). In a portion of the study it was found that computer science educators overwhelmingly favoured visual methods to teach algorithms, but later stated that the effectiveness of the animations in teaching was not always guaranteed, and strongly relied on factors such as user engagement and graphical attributes. Therefore while the implementation of the algorithm is clearly important, the usability and design of the product is equally so.

# 3 Analysis/Requirements

For the product to be used in an educational context, it was decided that it should be designed as a web-based application. This would mean that it would be easily accessible and distributed.

## 3.1 Analysis of Similar Examples

The following examples will be evaluated on the following metrics:

1. **Layout**. The information included on screen, and how each component relates to each other.
2. **Animation quality**. The clarity and style of the animation, and how well it can communicate information.
3. **Playback controls**. All the ways in which the user can control the animation, such as play/pause, move forwards/backwards, speed up/slow down.
4. **User inputs**. The effectiveness of the user interface, and the degree of control the user has over the content on screen.

The only example of a web application animating network flow that could be found was on the site *visualgo.net* (5).

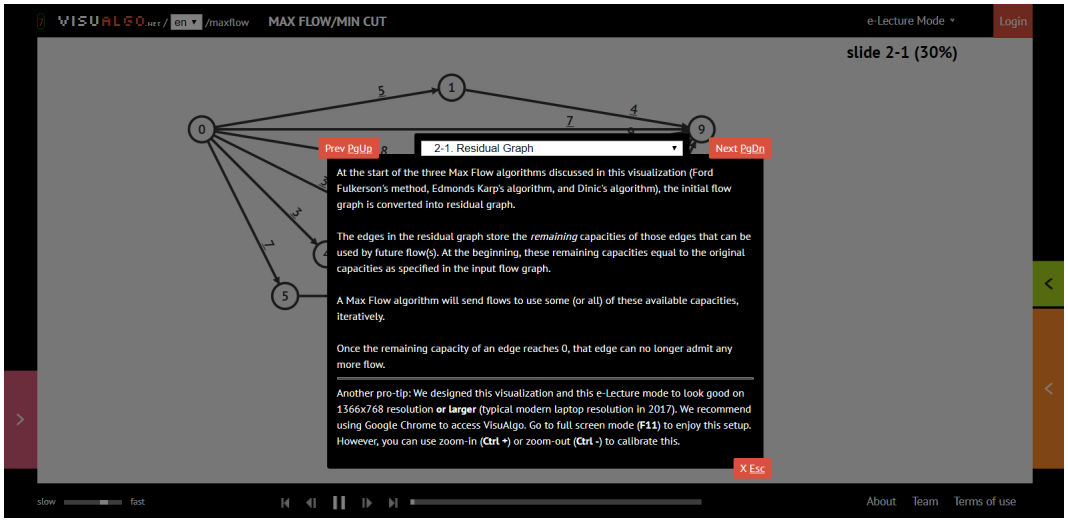



***Figure 3.1:*** *The first screen shown to the user when the user opens the algorithm. Lecture notes describing what you can see overlay over the graph, until closed.*

To interact with the application, the user must open the pink tab in the lower left corner. This allows them to select the method of network flow analysis (Ford-Fulkerson being one of the options), and which nodes are the Source and the Sink node.

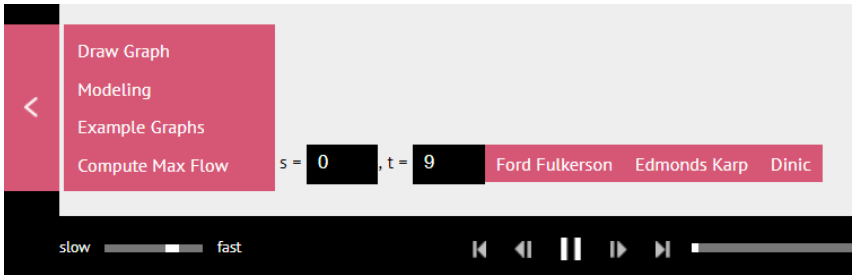



*Figure 3.2: The overlay to choose calculate the algorithm.*

There were additional overlays on the right-hand side of the application. One (yellow) printed out the action that the animation was executing, and another (orange) had a pseudo-code panel which highlighted the line that was being executed at each point in the animation.

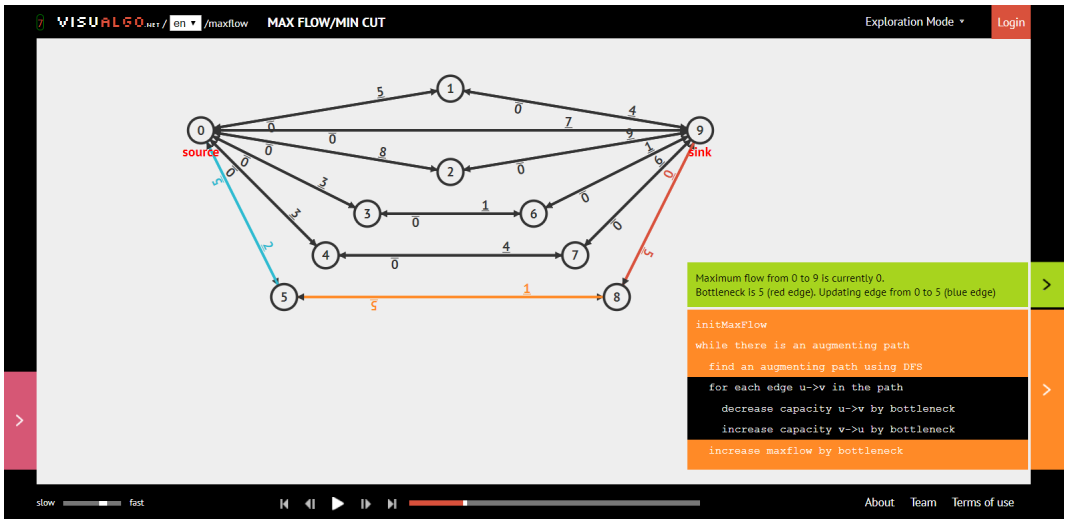



*Figure 3.3: The animation, part-way through execution.*

1. **Layout**. Some of the better aspects of the layout included the large central portion of the screen containing the animation effectively making it the main focus. Also, the slide out features on either side of the screen provided a sensible level of information, without interfering with the display of the animation. However, the initial slides at the beginning were a text-heavy way to convey information, where an animated algorithm should try and use graphical means to teach. The application also relied on these slides to teach the user how to interact with it.
2. **Animation Quality**. Visualgo's animation was very confusing, such that even a person highly familiar with the algorithm would have difficulty determining what it is doing. The layout of the graph was made illegible by each edge having a number upside-down. Not only is this harder to read, but it was not clear what the numbers represent. There was also no residual graph, which is a powerful graphical tool for teaching the Ford-Fulkerson algorithm.
3. **Playback controls**. These covered pretty much any action a user may want to perform, with the options to step forward, backward, and change the speed of animation. There was also a progress bar (bottom-middle), and the user could drag it forward and back to see different points of the animation.
4. **User input**. The user had lots of options for the animation content, including choosing

example graphs which forced the algorithms to exhibit a special case, or the ability to draw their own graph, and the choice of a particular network flow algorithm (including Ford-Fulkerson). However there were many bugs and usability issues. The drawing feature allows easy addition of nodes and edges, but there is no clear way to input the capacity of edges. The progress bar seemed to not work at all once an animation had completed. Overall, attempting to interact with the application was frustrating.

In general, *Visualgo* provided a lot of features which could be really effective for teaching network flow, but executed them poorly.

As no other examples of network flow animation could be found, other algorithm animation web applications were analysed which involved networks. The first of these was *algomation.com* (4), which hosts a large variety of algorithms as well as providing the ability for users to create or fork them. The algorithm in this example is Prim's Minimum Spanning Tree, as it also uses a network.

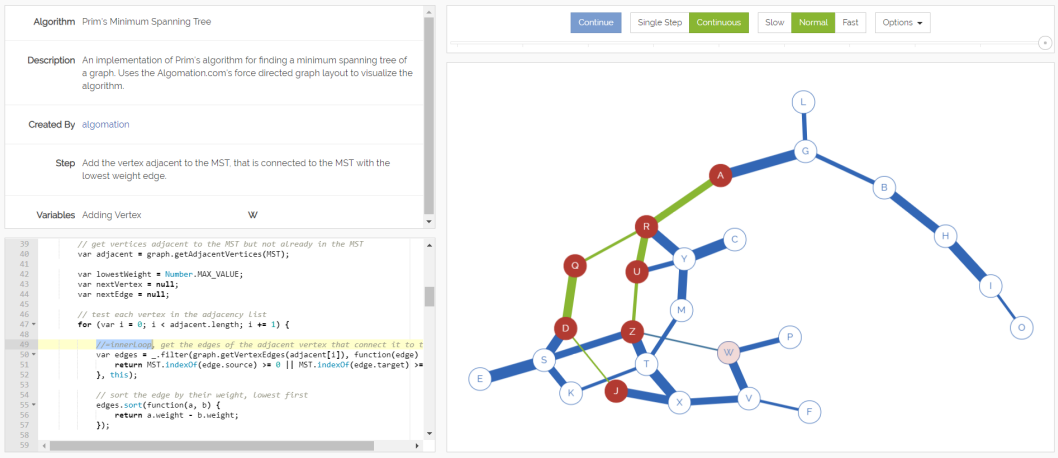



***Figure 3.4:*** *Algomation's representation of Prim's Minimum Spanning Tree*

1. **Layout**. Similar to *Visualgo*, the animation took up the largest portion of the screen. There was a concise description of algorithm (top-left) which avoids unnecessary information and assumes user has some knowledge, and the animation described each step of the algorithm as it executes with the relevant variables. The bottom-left panel jumps to the current place in the code throughout the animation. However, the use of the algorithm's source code is perhaps too over-complicated.
2. **Animation Quality**. The animation was well executed, with really clear diagrams and smooth animation. There was one issue in which the step and variable sections of the data that updated for each stage of the algorithm did not draw the eye when they changed. This was useful information that could have been made more obvious.
3. **Playback controls**. *Algomation* had clear and intuitive playback controls. Like *Visualgo* there is a progress bar, but here it expands as the animation runs. Once animation is complete, the user can drag the progress bar back and forth. The user may also select one of three speeds for the animation to run.
4. **User input**. The user was mostly only able to control the playback features, as the graphs were random and not customisable in any way.

Each feature of Algomation was clean and well executed, but did not include enough for the ideal outcome of this project.

Another example of animated algorithms included in research was a website run by David Galles from the Computer Science department of the University of San Fransisco (6). Once again,

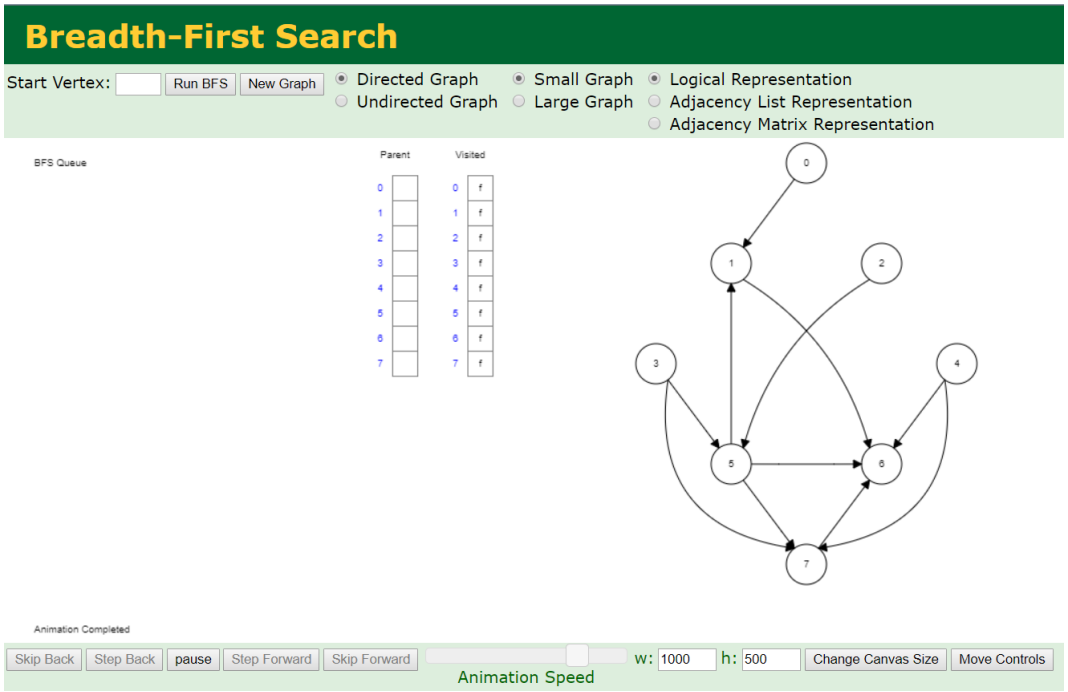



*Figure 3.5: The USFCA representation of the BFS*

there was no example of network flow but a different network related algorithm was chosen, the Breadth-First Search path finding algorithm.

1. **Layout**. The layout of the web page detracts from the animation. The middle section is divided into 3 equal sections, where only the graph off to the side fills up its entire space. Using the "Change Canvas Size" controls in the bottom does not affect the size of the objects on the screen, just the screen itself, which makes this feature feel redundant. It is also a little confusing to have the “Run BFS" button at the top, but the rest of the playback buttons on the bottom.
2. **Animation Quality**. The animation is relatively simple, but presents itself well. All the additional information for the algorithm is presented clearly.
3. **Playback controls**. There are a sensible variety of controls for the user, which are easy to use playback controls with speed control.
4. **User input**. The application provides clear, randomly generated graphs, with options for small or large. In general it is a sensible interface, with each feature working as expected.

The lack of animations for network flow algorithms showed that they are challenging to visualise effectively. There are many different aspects of the algorithm to consider, and the graphical animation alone is not enough to communicate the algorithm to the user.

## 3.2 List of requirements

Taking each of the different examples into consideration, a list of fundamental requirements for the project was devised, based on the following principles.

- **A clear, readable graph**, including few edge overlaps, a large canvas size, and clear labels on edges. This is so that the display of the graph won’t obscure crucial information conveyed during the animation.

- **Playback controls** of start, stop, speed up/down, and move forwards/backwards. This allows a user to traverse through the animation at their pace, whether teaching or learning from the algorithm.
- **The ability of a user to generate or draw a graph**. If the user has greater control over the network used in the animation, it means they can display certain cases of the algorithm, or use familiar cases to learn from.
- **Information on each step of the algorithm during its execution** including pseudo-code of the algorithm.
- **Intuitive interface**, so that the user isn't distracted from the main point of the application by struggling to effectively use it.

The requirements were developed in the form of MoSCoW. Functional requirements are shown in bold, and non-functional in normal weight:

**Must Have**:

- **Playback controls including play, pause, rewind, step forward, and step backward.**
- **Default and random graph generation.**
- **A pseudo-code panel of the algorithm.**
- **A way to control the speed of the animation.**

**Should Have**:

- Duplicating the graph at points, to show the residual graph.
- Clear interface.
- **Users drawing their own graph.**

**Could Have**:

- **Saving and uploading graphs as a file.**
- **Animate the separate algorithms used in Ford-Fulkerson (like path searching, or different forms).**
- **Flow counter, to keep track of total flow.**
- **Logging panel to print out detailed steps of animation.**

**Would like**:

- **Skip to end or beginning of animation.**
- Mobile-friendly interface (for use in lectures).

# 4 Design

This chapter will cover the design of the application. First there will be a general overview of the layout and how the different components are grouped together, and then a more detailed look at each of the elements. Finally, the design of the animation will be looked at.

## 4.1 Layout

Similar examples to this project all shared a common layout theme; a canvas taking up a large portion of the screen would contain the animated network, with supplementary information in containers off to the side or top of the main canvas. The product was designed to fit entirely on one web page, with the graphs taking up three quarters of the screen, and the remaining portion dedicated to a information bar on the left.

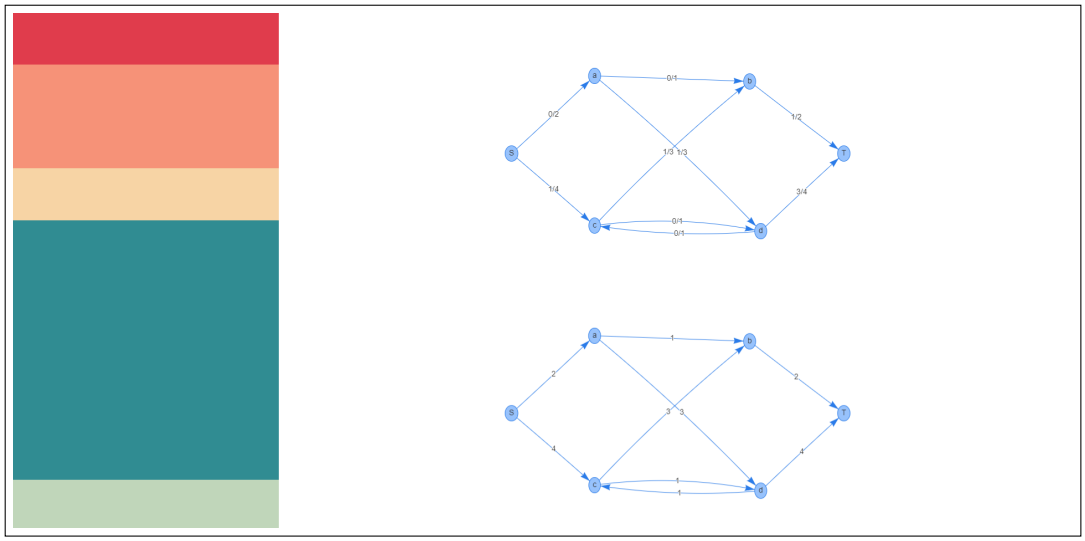


***Figure 4.1:*** *The initial design of the layout. This was constructed very early on based on rough paper wireframes. The information bar (left) is segmented for the different elements that it will contain.*

As the product began to include more and more features unrelated to the animation, the space on the sidebar became limited and placing information in an intuitive, clear, and sensible way became infeasible (Centre of Figure 4.2). A redesign was implemented, taking great care to place each feature in a way that would be intuitive to the user. The greatest portion of the screen was still dedicated to the animation canvas, but a new row added beneath it containing the playback buttons, speed control slider, and flow counter. The former two were placed there to be similar to video player applications, and the flow counter stood alone to draw attention to it.

Updates to the information bar included a more intuitive placement of the network generation functions; while they are grouped in a collapsible accordion menu indicating relation, their

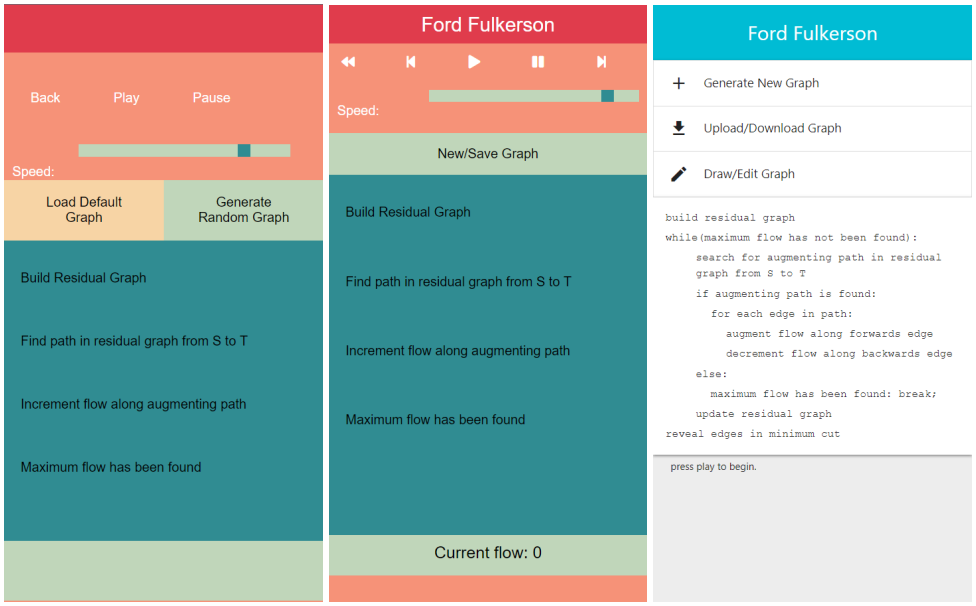



***Figure 4.2:*** *The design progression of the information bar*

purposes are all distinct and clear. The collapsible menu then allowed extra room for more detailed pseudo-code, and the execution trace placed below that. The user would now be able to collapse the unnecessary graph information while the animation runs, and focus on the algorithm details.

Additionally, a more cohesive colour theme was chosen: teal, with an orange accent to draw attention to important information (e.g. the highlighted line of pseudo-code (Figure 4.3), among other things). To avoid confusion, the residual graph would not be revealed until the animation had begun. Iconography was used to replace textual instructions where possible.

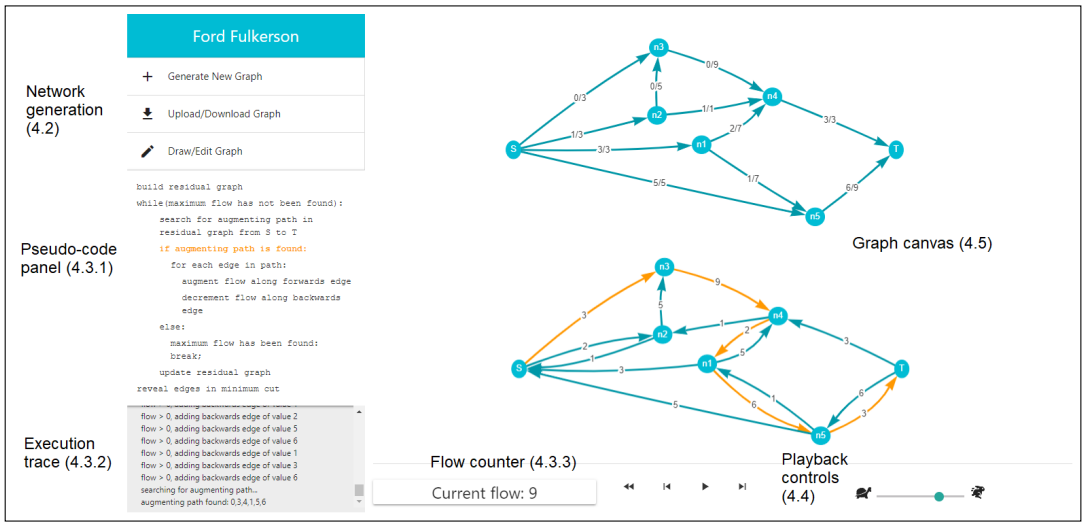



***Figure 4.3:*** *The final product design, with the different components labelled.*

## 4.2 Network generation

A fundamental requirement of the tool is the user's ability to generate a network in multiple different ways. Someone teaching the Ford-Fulkerson algorithm may want to display a specific network, which exhibits a rarer case of the algorithm during execution. For variety, a user may want a randomly generated network if aiming for a more general understanding of the algorithm.

Two of the network creation methods require very little input from the user. The simplest is a default network taken from the Algorithmics II teaching notes. This exhibits a case of a backwards edge being decremented, and is a consistent option for the user. The random generation takes the desired number of nodes in the graph, and will draw a randomised network with node *S* on the far left and node *T* on the far right side of the screen. Further information on the implementation of this is provided in section 5.2.1.

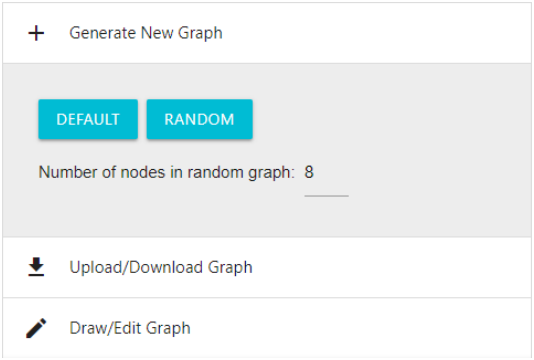



***Figure 4.4:*** *The menu for network generation tools.*

If the user desires greater control over the graphs, the design includes additional network creation methods. The first of these features allowed the user to download or upload a file representation of a graph. This can be used to save randomly drawn graphs that may serve as good teaching examples to use later.

The second is a drawing feature, with which the user may directly interact with the canvas and create networks of their own design. This raised issues, as there were a variety of potential ways in which a user could add nodes and edges to join them. Not only this but the nodes *S* (the source node) and *T* (the sink node) had to be included in the network, in positions consistent with the other networks created. The final design of the drawing feature included the following aspects:

- Once the drawing mode was activated, *S* and *T* would be displayed in set positions at the left and right of the screen, so that the user could add as many nodes in-between as required.
- To interact with the canvas there would be a toolbar at the top with two buttons, 'add node' and 'add edge'.
- When 'add node' is selected the user can click to place one node anywhere on the screen, or cancel the action. The nodes are labelled '$n1, n2, n3 \ldots nx$' corresponding to the order in which they are placed.
- When "add edge" is selected, the user may drag an edge from one node to another. Once two nodes are connected, the user is prompted to choose a capacity (default 4).
- When any node or edge is selected, a third button appears in the toolbar to 'delete selected'. If a node is deleted, it is removed from the graph along with any edges connected to it. The user may delete any node except for *S* or *T*.
- The user may not connect a node to itself, or enter a capacity that is either not an integer or negative, but there are very little other restrictions. It is assumed that if the user wishes

to draw a graph that could not possibly run the Ford-Fulkerson algorithm, they have some reasoning behind it.

## 4.3 Additional Algorithm Information

The graphical element of the algorithm would not be enough alone to convey a meaningful amount of knowledge to the user. Therefore a pseudo-code panel, execution trace section, and flow counter panel were added, to be updated at different stages of the animation.

### 4.3.1 Pseudo-code

The pseudo-code panel contained a broad description of the algorithm. As different sections of the algorithm were executed in the animation, the corresponding line of the pseudo-code would be highlighted.

```
build residual graph
while(maximum flow has not been found):
      search for augmenting path in residual graph from S
      to T
      if augmenting path is found:
         for each edge in path:
            augment flow along forwards edge
            decrement flow along backwards edge
      else:
         maximum flow has been found: break;
      update residual graph
reveal edges in minimum cut
```

***Figure 4.5:*** *The pseudo-code panel, midway through an animation.*

### 4.3.2 Execution Trace

The execution trace is a section which prints a new line after each significant action is completed in the animation. For example, during the building of the residual graph, the execution trace will print 'adding a forwards/backwards edge of capacity x' for each edge. Or if the algorithm is currently searching for an augmenting path, the latest line will be 'Searching for augmenting path...' and subsequently 'augmenting path found: 0, x, y, z, T'.

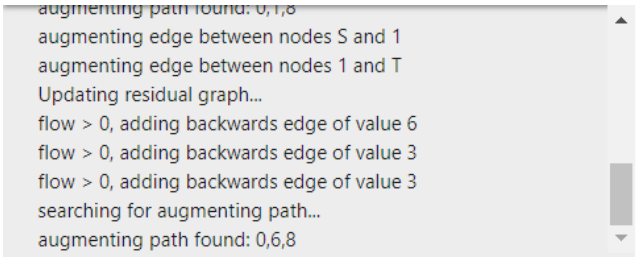



***Figure 4.6:*** *The execution trace during an animation.*

### 4.3.3 Flow Counter

After each iteration of the algorithm, the flow counter will update to match the current amount of flow passing through the network. By the end of the animation, it will contain value of the maximum flow.

## 4.4 Playback Controls

An end goal of the project was to have controls to play, pause, change the speed, or move step-wise through the algorithm. Therefore, the design included four playback buttons used to rewind, step backwards, play/pause (toggle), or step forwards. Additionally a speed control slider was included, which affected the speed of the animation both forwards and backwards.


***Figure 4.7:*** *The playback buttons, and the speed control slider while being adjusted*

To avoid the use of needless written instructions where possible, the use of the tortoise and the hare icons were used to indicate the purpose and direction of the speed control slider.

## 4.5 Graph design

The graph canvas occupied the largest portion of the interface. The network that the algorithm is executed on is named the top graph, referring to its placement during an animation, and the second network is the residual graph. Before an animation is played, the residual graph is hidden, and the top graph takes up the whole canvas. During the animation the canvas is split in half horizontally for the two graphs.

As the random graph generation occasionally creates graphs with a layout that can appear 'tangled', the user has the ability to drag nodes to any position to make the graph easier to read (Figure 5.1). To indicate that the graphs are interactive in this way, nodes will change colour when hovered over. Any changes to the node positioning in either graph will be copied in the other, so that both maintain consistency in appearance.

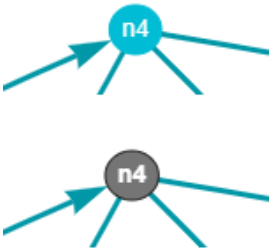



***Figure 4.8:*** *A node with and without a cursor hover.*

## 4.6 Animation

There were two major areas to consider for the animation method. The first was the order in which the calculation of the algorithm and the execution of the animation should go in. Either the entire algorithm could be run and the animation prepared before anything was shown to the user, or the algorithm could be calculated 'live' as it executed during the animation. The former was chosen, so that stepping forwards/backwards could be done efficiently. The second area was the data structure needed to store the information that this animation used, and what to include in the objects stored in the structure. This was decided early on in the design process, and proved robust enough to remain mostly unchanged throughout the project.

### 4.6.1 Method

The animation manipulates one edge at any time from either the top graph or the residual graph, as the nodes are unaffected throughout the algorithm. The data stored is an array composed of objects called “animation steps". Each object details the edge that is being changed, the action in which it is being changed (e.g. change of colour or label), and additional information such as which pseudo-code line should be highlighted during that step. When the user presses play, the animation engine iterates through this array with the desired intervals, and the graphs change accordingly.


***Figure 4.9:*** *Each stage of an edge in the top graph having flow incremented. The order of execution goes from left to right.*

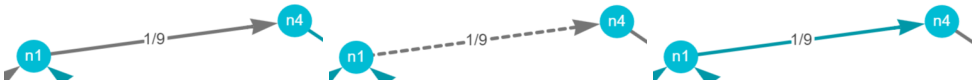



***Figure 4.10:*** *Each stage of an edge in the top graph having flow incremented. The order of execution goes from left to right.*

In order to have a rewind function, whenever an animation step is executed it stores the previous state of its edge. When the user rewinds the animation, the animation traverses the array of animation steps backwards from its current place, and restores each edge to its previous state.

### 4.6.2 Styling

From the background research in Chapter 2, it is clear that the presentation of the animation is crucial to aid understanding. Therefore the styling was designed to ensure that the user could recognise the different stages of the algorithm, and that the eye would be drawn to the graph currently displaying an important action.

When first implemented, the default colour of the edges was blue, with one accent colour red to draw the eye. The animation executed in the following stages:

1. **Build residual graph**: Each edge $e1$ of the top graph would be highlighted red, and its corresponding edge $e2$ in the residual graph added. $e1$ would then return to blue.
2. **Find augmenting path**: The augmenting path $P$ would be highlighted in red in the residual graph.
3. **Increase flow along augmenting path**: Each edge along $P$ in the top graph would have its label updated accordingly (Figure 4.11). After this, the residual graph would return to its original state.
4. **Update residual graph**: Each edge along $P$ would be highlighted in red, and the corresponding edge(s) are updated in the residual graph.

The main problem with this design was found in stage 3. The change of label alone was very hard to spot, as there were no other indicators that the edge in question would be changing. Additionally, at the end of this stage both graphs would only contain blue edges, suggesting (incorrectly) that the end of an iteration had been achieved.

To counter these problems, additional steps and styling was incorporated into the animation design. The edge width increased, along with the background of the labels to make them clearer. The accent colour, now orange, was joined with an additional neutral colour grey. Dashed lines were also included. The stages of the animation were augmented in the following ways:

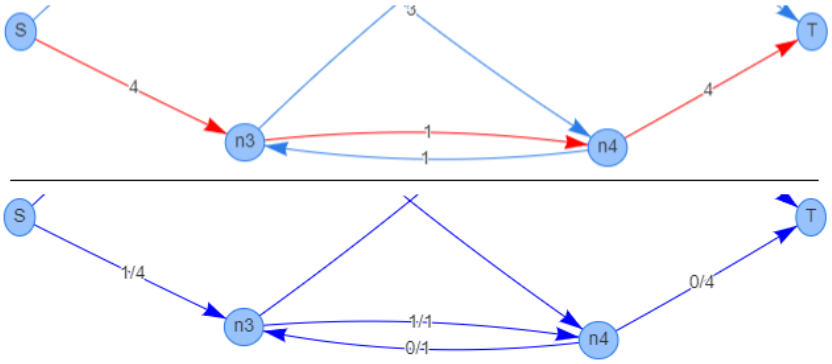



*Figure 4.11: An early animation of highlighting the augmenting path in the residual graph, and subsequently augmenting each edge by 1 along the path in the top graph.*

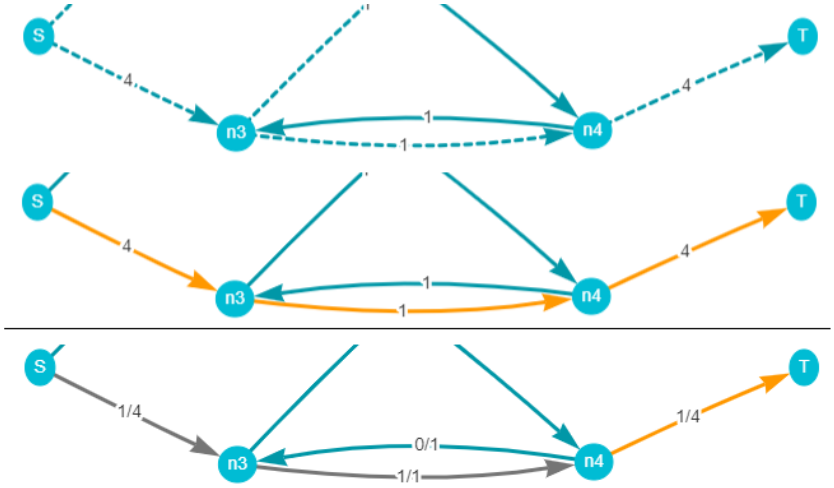



*Figure 4.12: The styling of the animation in the current project.*

1. **Build residual graph**: This remained the same method as before, but using grey to highlight to top graph's edges.
2. **Find augmenting path**: An additional stage was added just before highlighting the augmenting path, in which each edge to each node visited during the search would be dashed. Once found, the edges not along $P$ would return to solid and the edges included in $P$ highlighted in yellow (Figure 4.12).
3. **Increase flow along augmenting path**: Each edge along $P$ in the top graph would be treated as in the first 4 stages of Figure 4.10. At the end of this stage, $P$ in the top graph is highlighted in grey (to indicate it is still significant), and $P$ in the residual graph is returned to teal.
4. **Update residual graph** Each edge in $P$ in the top graph is dashed (stage 5 of Figure 4.10) and the corresponding edge(s) of the residual graph are updated accordingly.

# 5 Implementation

## 5.1 Technology

Due to the decision to create the product as a client-side web application, it was entirely written HTML, CSS, and JavaScript. A server-side feature was considered in which users could save graphs with their account, but not implemented as it required a relatively large investment of time compared to the overall benefit of the feature. The javascript library *vis.js* was used in both the calculations behind the algorithm and the graphical display of the networks. A different library *sigma.js* was also considered, but was more suited to visualising larger sets of data. *vis* employed a much more versatile set of functions, including styling, data management, and user interaction. For the UI, *materialize* (11) was used to create a familiar and unified feel to the design.

### 5.1.1 vis.js

A *vis* Network (7) is the object used to visualise the networks displayed, comprised of several modules that define specific aspects of the network, most notably the node and edge DataSets. *Vis* DataSets (8) are key/value based structures used to store and manipulate the sets of nodes and edges that comprise each network. Each object in a DataSet has a unique identifier, used to access and manipulate it. Node objects (9) contain several options, including styling, manipulation, physics, and positioning. Edge objects (10) have similar properties, with additional options specifying the nodes that the edge connects.

In addition to the node and edge DataSets were options governing node layout, user interaction, and an interface for user manipulation. Networks are displayed graphically using an HTML element as a canvas.

The Ford-Fulkerson algorithm in this project required two networks: the main (top) graph and the residual graph. In the implementation each network had a duplicate, one for display, and one for the calculation of the algorithm. This is so that all the data for the algorithm could be calculated before the animation began, independent from the display. As previously stated, a network is made up of two *vis* DataSets, nodes and edges. Both graphs share the same set of nodes, but have distinct sets of edges. This meant that there was one common DataSet of nodes (named *topNodes*) for all the networks, and four DataSets of edges (*topEdges*, *resEdges*, *algTopEdges*, *algResEdges*) for the graphs and their duplicates.

In addition to these data structures each graph has an adjacency matrix representation (*topAdjMatrix* and *resAdjMatrix*), where the cell between two connected nodes contains the unique ID of the edge connecting them. The ID of an object in a DataSet provides access to it with the *get*() function, allowing its manipulation. The additional matrix representations hugely improved efficiency at several points of the algorithm.

## 5.2 Network Generation

One of the foremost tasks of implementation was to have a tool to create networks on which to run the Ford Fulkerson algorithm. First of all, a default network was needed for consistent testing. This was achieved by hard-coding an array of nodes with preset $x$ and $y$ values for a predictable layout (see Figure 5.1), and an array of edges with predetermined capacities and connecting nodes.

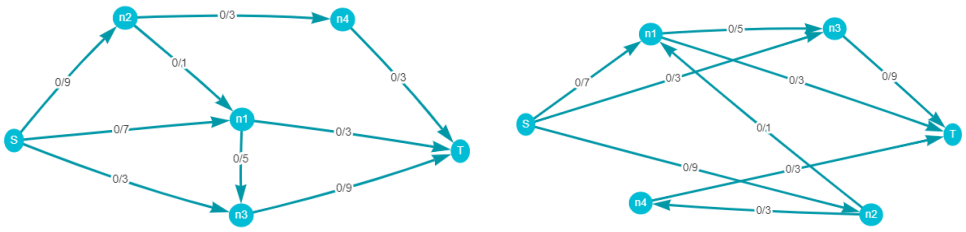



***Figure 5.1:*** *How differing x and y values can drastically change the look of a graph*

### 5.2.1 Random drawing

It was important to my requirements to be able to also randomly generate networks. However, these 'random' graphs would have to fit a set of standards to still be a useful and clear tool for the display of the algorithm, as follows:

- The source must be the left-most node, and the sink the right-most
- Each node must have two edges, and be on a path from the source to the sink
- There must be no circular (i.e. from and to the same node) or parallel edges
- There should be no edges incoming to the source, or outgoing from the sink
- Nodes should not overlap
- Edge capacity should be between 1 and 10. The limit is not for any functional reason, only that it is easier for a user to read and prevents needlessly long animations.
- The number of nodes $N$, which the user may input, is limited from 3 to 50. Larger graphs are entirely possible, and not prevented in testing (Section 6.1), but look almost entirely unreadable to a user at high values of $N$. Values of $N$ at which the animation is most effective visually is relatively small, about 5 to 15.
- The number of edges is calculated as $N * 2 - 3$. This specification is also cosmetic and serves the purpose of preventing 'tangled' graphs.

The first requirement was implemented by initialising the source and sink with preset positions on the left and right of the canvas. The $x$ and $y$ coordinates for all other nodes are randomly assigned on creation. On rare occasions they can overlap, but this was mitigated by the *vis* physics engine causing nodes to repel each other, or in the event of that failing, the user's ability to drag and place the nodes anywhere on the canvas.

The remaining nodes and edges are added by the following method:

1. Any two unconnected nodes would be connected to the sink ($T$). This ensures that $T$ has at least two incoming edges. Every unconnected node after that would be either connected to $T$, or any node already on a path to $T$. This continues until all nodes form a spanning tree. At the end of this stage, all the nodes with no incoming edges form a set ($A$).

2. The next two edges generated would be from the source *S* to any node *n* from *A*. *n* would then be removed from *A*. This ensures that *S* has at least two outgoing edges. Edges would then be added from *S*, or any node on a path from *S*, to any node in *A* until *A* was empty.
3. Any edges not yet added would be added anywhere in the graph, provided they fitted the above criteria.

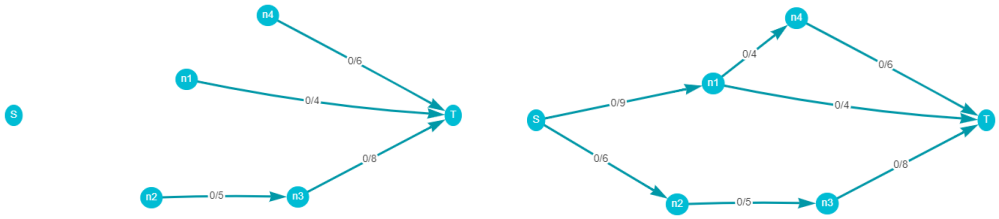



***Figure 5.2:*** *Steps 1 and 2 of the random network generation. At the end of step 1, A consists of n1, n2, and n4.*

This approach ensured that there would be plenty of variety in the generated networks, while also providing a network that a user would be able to easily comprehend.

### 5.2.2 User Inputs

For the uploading and downloading feature of the networks, the format of the files required careful thought. It must be detailed enough to preserve the layout of a network upon uploading it, but simple enough to understand with little instruction. An example of the file format is show below.

```
{
"nodes" : [
{ "id": 0, "x": -275, "y": 15},
{ "id": 1, "x": -115, "y": -108},
{ "id": 2, "x": -5, "y": -47},
{ "id": 3, "x": -8, "y": 78},
{ "id": 4, "x": 275, "y": 66}
],
"edges" : [
{ "from": 3, "to": 4, "capacity": 9},
{ "from": 2, "to": 4, "capacity": 5},
{ "from": 1, "to": 2, "capacity": 1},
{ "from": 0, "to": 3, "capacity": 3},
{ "from": 0, "to": 1, "capacity": 7},
{ "from": 0, "to": 4, "capacity": 7},
{ "from": 0, "to": 2, "capacity": 3}
]
}
```

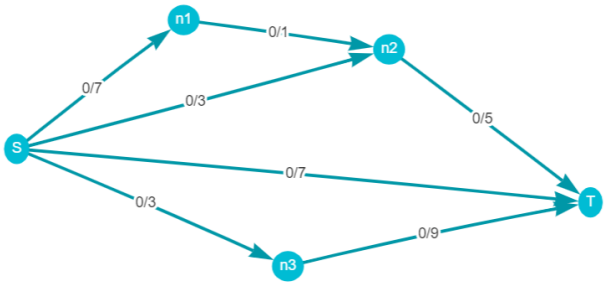



***Figure 5.3:*** *The JSON format for a small graph, and its graphical representation*

The type of file chosen was JSON, in order to work efficiently with JavaScript, and the data stored with two arrays of nodes and edges. Each node must have an id, with the node with id = 0 being the source *S*, and the node with id = $N - 1$ being the sink *T*, where *N* is the number of nodes. The *x* and *y* values for nodes are optional; if not supplied, the node will be assigned a suitable random coordinate (*vis*'s physics engine repels nodes from each other). It is crucial to most of the calculations involving graph data that the node ids are in consecutive order. Each edge would have 3 parameters, all compulsory. The two nodes that the edge connected '*from*' and '*to*', and the capacity of that edge.

Downloading a graph would write a representation of the graph currently showing on the interface, and download it to the user's file system. If a file was uploaded, it would first be parsed to check for formatting errors, and then undergo a series of checks to ensure it was a correctly written graph. If there were any errors found in the file, an alert would be shown to user detailing the error.

Drawing a graph is done by using *vis* graph manipulation feature. This provides functions for '*addNode*', '*addEdge*', '*deleteNode*', '*deleteEdge*', '*editNode*', and '*editEdge*'. It had to be modified to suit the purposes of the application and stay within the points detailed by Section 4.2. Adding any nodes or edges would not only use the functions provided by *vis*, but also the adding functions custom made for the other graph creation features, so that the data remained suitable for graph functions used in the algorithm calculation. The order of the node and edge ids must be consecutive, both for indexing and for the adjacency matrices used (Section 5.1.1). Therefore deleting a node or edge would mean decrementing by 1 each node or edge id greater than that of the node/edge being deleted.

## 5.3 Algorithm

### 5.3.1 Building/updating residual graph

Building the residual graph the first time copied each edge in the top graph to the residual graph with a weight of the edge's capacity, as no edge in the top graph could have a flow greater than 0. After each instance of the flow increasing, the residual graph would be updated along the augmenting path.

The path is represented with an array of nodes that it traverses, from 0 to $T$, so each edge in the path can be accessed by taking two consecutive nodes, '$from$' and '$to$'. The '*capacity*' and '$flow$' of each edge in the top graph is used in the calculation, as are the '$forwards$' and '*backwards*' edges in the residual graph. The forwards edge goes in the direction of the augmenting path ($resAdjMatrix[from][to]$), and the backwards edge in the opposite direction ($resAdjMatrix[to][from]$).

```
if(augmenting && (capacity - flow > 0)){
  update forwards edge label to (capacity - flow);
} else if (decrementing && (flow > 0)){
  update forwards edge label to flow;
} else {
  if((oppEdge = topAdjMatrix[to][from]) exists){
    oppFlow = getFlow(oppEdge);
    oppCap = getCapacity(oppEdge);
    if(augmenting and (oppFlow > 0){
      update forwards edge label to oppFlow;
    } else if (decrementing and (oppCap - oppFlow > 0)){
      update forwards edge label to (oppCap - oppFlow);
    } else {
      remove forwards edge;
    }
  } else {
    remove forwards edge;
  }
}

if(backwards edge exists){
```

```
22   if(augmenting) {
23     update backwards edge label to flow
24       or add backwards edge with label flow
25   } else {
26     update backwards edge label to (capacity - flow)
27      or add backwards edge with label (capacity - flow)
28   }
29 }
```

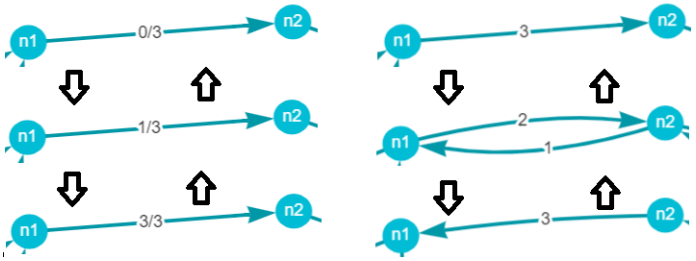



***Figure 5.4:*** *A basic example of the flow of an edge being incremented (downwards) or decremented (upwards) in the top graph (left), and its corresponding changes to the residual graph (right)*

This method of updating the residual graph leaves it updated such that there are no parallel edges. In the case of having two possible edges in the same direction, the algorithm will pick one.

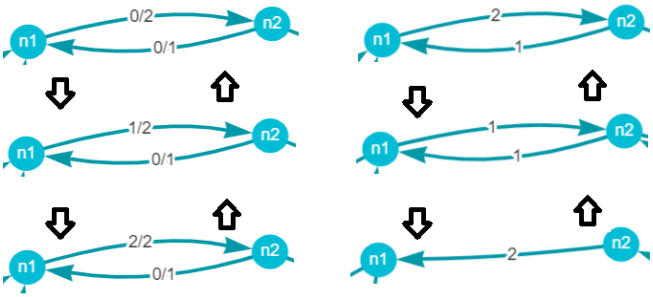



***Figure 5.5:*** *An example of updating the residual graph in which there are two edges, and one of two possible edges is chosen.*

This makes the step at line 6 crucial in preventing a bug that removes an edge where it is still required.


***Figure 5.6:*** *Here there are two edges in the top graph, and both are augmented (at different iterations of the algorithm). As the second edge is augmented, the forwards edge is deleted as the capacity - flow = 0. However, it is still required for the edge that has not been updated. This is why there is a check for an opposite edge at line 6.*

### 5.3.2 Finding augmenting path and minimum capacity

The augmenting path was found using a breadth-first search algorithm, beginning at *S*. First each node would be set as a 'parent' to itself. Then all of *S*'s neighbour nodes would be visited, and *S* would be set as a parent to each of them, before visiting all of their neighbours, and so forth. Once there are no more neighbours to visit, it is checked if *T* is it's own parent (ie, it couldn't be reached). If so, there is no path from *S* to *T*. If not, then *T* is pushed to an array of the path, followed by its parent *p*, and *p*'s parent recursively, until *S* is pushed. The path from *S* to *T* is then the path array, reversed.

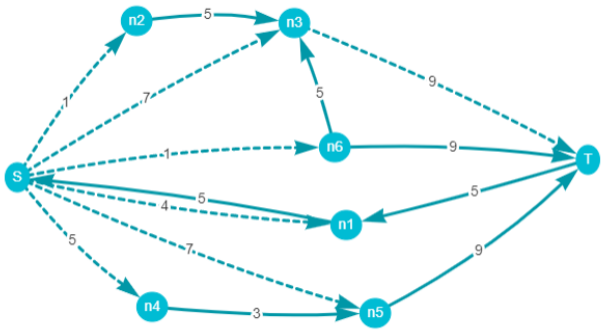



***Figure 5.7:*** *Searching for a path in the residual graph. The edges to the nodes being visited are shown dashed.*

Once the path is found, the minimum capacity of the path '*m*' has to be calculated. This is the value of the edge with the smallest amount of remaining capacity, and therefore the largest amount of flow that could be augmented along the path. A variable '*minCap*' would be compared to the available capacity of each edge. If *minCap* was greater than this capacity then it would be set to the value of the capacity.

### 5.3.3 Augmenting/decrementing edges

To represent the capacity and flow of a particular edge, the label property of an edge was assigned a string of the form '$x/y$' where $x$ is the flow and $y$ is the capacity. There were three functions used throughout nearly every step of the algorithm: *getCapacity*, *getFlow*, and *setFlow*. Each of these took the label string and either returned the capacity, flow, or a new label.

```
m = getMinimumCapacity(path);
for(edge in path) {
  if(edge direction == forwards) {
    edge flow = edge flow + m
  } else if(edge direction == backwards) {
    edge flow = edge flow - m
  }
}
```

Determining the direction of the flow was initially done by using the top graph's adjacency matrix (topAdjMatrix). A function *getEdgeData* would take the 'from' and 'to' node ids, and return an object containing the edge id (in the top graph) and its direction. If $topAdjMatrix[from][to]$ was found to exist, then the direction would be forwards (represented as 1), but if it did not exist and

*topAdjMatrix*[*to*][*from*] did, the direction would be backwards (0). However, this overlooked the possibility of there being two edges between 'from' and 'to', as then the forwards edge could be selected when the other was correct.

This was fixed by adjusting the residual graph to include a tag on each newly added backwards edge, during the updating of the residual graph. Then instead of checking the direction of the edge in the top graph, the tag would be checked. If true, the direction would be backwards. If false, forwards. This was also used during the updating of the residual graph, to determine if the edge had been augmented or decremented.

### 5.3.4 Finding a minimum cut

To find a minimum cut, a set of nodes *A* is composed of the source node *S* and all nodes that *S* can access via the residual graph, after the algorithm has completed. The set of nodes *B* is all other nodes in the graph. The minimum cut *C* is the set of edges in the top graph from any node in *A* to any node in *B*.

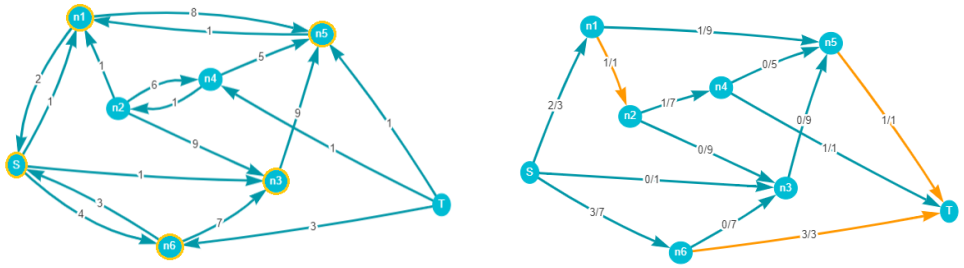



***Figure 5.8:*** *The residual graph with set A highlighted, and the top graph with minimum cut C highlighted*

## 5.4 Animation

This section will go into greater detail for the implementation of the animation, first analysing the detail used, and then how it is processed.

### 5.4.1 Animation Steps

The information that an animation step contains is shown in 5.1. Every step must contain a network id, an action, an edge id, and a '*pStep*' id. Other arguments are only required depending on the action (e.g. a 'highlight' action only needs to know the colour).

Animation steps are held in an array, *animationSteps*, which is added to throughout the calculation of the algorithm. Here is a code snippet of the residual graph being built at the start of the algorithm:

```
1  function buildResidualGraph(){
2     prepareOutputLine(6);
3     var edges = [];
4     var cap, i;
```

***Table 5.1:*** *A table of all the variables that one animation step contains.*

| Variable | Description | Values | When used |
|---|---|---|---|
| **network** | An id of the network that the edge belongs to | 0 for top graph, 1 for residual graph | Always |
| **action** | The action to be executed by the step | One of:["highlight","dash", "label","add","remove"] | Always |
| **edgeID** | The id of the edge that will be changed | int | Always |
| **pStep** | The pseudo-code line that will be highlighted during this step | int | Always |
| **color** | The new colour of the edge | Object of the form {color: "desired colour"} | action == "highlight" |
| **label** | The new label of the edge | String | action == "label" |
| **from** | The id of the node that the edge will be going from | int | action == "add" |
| **to** | The id of the node that the edge will be connected to | int | action == "add" |
| **prevData** | The data of the edge before being changed (null on creation of step) | Object | Always |
| **dash** | Boolean indicating if the edge should be dashed | Boolean | action == "dash" |
| **outputID** | The id of the execution trace line | int | Optional |
| **outputData** | Data to insert into execution trace line, if any | Array of ints | Optional |

```
5     // For each edge in the top graph, add corresponding edge with value = capacity
6     for(i = 0; i < algTopEdges.length; i++){
7        var edge = algTopEdges.get(i);
8        createHighlightAnimation(TOP, i, 0, '#757575'); // show top edge in grey
9        cap = getCapacity(edge.label);
10       createAddEdgeAnimation(RES, edgeID, 0, cap, edge.from, edge.to, 2, cap);
11                                        // add edge to residual graph
12       addEdgeToAlgRes(edgeID, cap, edge.from, edge.to);
13       edgeID++;
14       createHighlightAnimation(TOP, i, 0, '#0097A7'); // return top edge to normal
15    }
16    algResEdges.update(edges);
17 }
```

Though this is a relatively simple point of the animation, there are four points at which animation steps are added to in this snippet: lines 2, 8, 10, and 13. To avoid entering twelve arguments every time a new animation step is added, more specialised functions are used. We will focus for now on *createHighlightAnimation* (lines 8 and 13) as an example. It takes four arguments in this instance (network, edgeID, pStep, and color), but may take an additional two for outputID and outputData:

```
1  function createHighlightAnimation(network, edgeID, pStep, color,
```

```
                                outputID, outputData){
   addAnimationStep(network, "highlight", edgeID, pStep, color,
               null, null, null, null,outputID, outputData);
}
```

As can be seen above, *createHighlightAnimation* then formats the arguments it has been given into the correct order for *addAnimationStep*. This method is the same for all specialised functions. *addAnimationStep* then creates and pushes the animation step as an object to *animationSteps*.

There are two extra actions not included in the table, 'reveal' and 'updateFlow'. Before the animation is run only the top graph is shown taking up the entire canvas, so 'reveal' is the first step to be executed, which splits the graph canvas in two and sets the residual graph beneath the top graph. Whenever the flow along an augmenting path has been completed, 'updateFlow' adds the extra flow to the value in the flow counter. Both of these special cases are pushed to *animationSteps* outside of *addAnimationStep*.

Another special case of these animation functions is line 2 of *buildResidualGraph*() shown above, *prepareOutputLine*(6). This is adding an animation step which doesn't impact the graphs at all, and only prints out a line in the execution trace panel. Each line is one of 11 possible templates, such as 'capacity - flow = $, adding forwards edge of value $', 'augmenting path found: $', or 'Building residual graph...', for example.

The character '$' acts as a placeholder, and if the line specified by the outputID contains a '$' character, then it must also have outputData to put in its place. *prepareOutputLine*(6) is indexing the line 'Building residual graph...', so it needs no outputData. Line 11 of the initial excerpt, '*createAddEdgeAnimation*(RES, edgeID, 0, cap, edge.from, edge.to, 2, cap);', has *outputID* = 2 and *outputData* = *cap* for its final two arguments. This is specifying the output line 'adding forwards edge of value $', where '$' will be replaced by *cap*, so the line printed out would be 'adding forwards edge of value 6', if *cap* = 6.

### 5.4.2 Animation Engine

Each step of the animation was executed using the function *executeAnimationStep*(). There were 5 possible states for the animation to be in: *STEP_BACKWARD*, *REWIND*, *PAUSE*, *PLAY*, *STEP_FORWARD*. Each of these represents an integer from -2 to 2, respectively. This was defined as the *playState*. *executeAnimationStep*() increments/decrements the index in *animationSteps* depending on the playState, and takes the next/previous animation step. It prints a line in the execution trace (if there is one in the step object), uses a switch statement on the step action to execute the appropriate animation, and highlights the specified line of pseudo-code.

The switch statement evaluates the action of the current step, and executes a function accordingly. For example:

```
case("remove"):
   executeRemoveEdgeStep(edges, edgeID, currentStep);
   break;
```

In which 'edges' is a DataSet for the edges from either the top or residual graph, selected earlier from the network id. This leads to:

```
function executeRemoveEdgeStep(edges, edgeID, currentStep){
   if((playState == PLAY) || (playState == STEP_FORWARD)){
```

```
3         currentStep.prevData = edges.get(edgeID);
4         edges.remove(edgeID);
5      } else if ((playState == REWIND) || (playState == STEP_BACKWARD)){
6         edges.add(currentStep.prevData);
7      }
8   }
```

When the animation is moving forwards, the current data of the edge being changed is saved in *prevData* before any action is executed. This is then recovered when the animation is moving backwards.

Stepping forward and backward is possible due to *executeAnimationStep*() only running one step at a time. Automating the process when rewinding or playing is achieved with *animateAlgorithm*(), which sets a time interval based on the speed control bar and runs *executeAnimationStep*() for each interval, stopping when the start or end of *animationSteps* is reached.

# 6 Evaluation

The purpose of the evaluation was to answer the following questions:

- Is the implementation of the Ford-Fulkerson algorithm correct?
- Is the design of the application intuitive to use?
- Does the application present the algorithm in a way that strengthens understanding?

## 6.1 Correctness testing

The implementation of the algorithm was tested using the Min-cut/Max-flow theorem, which states that the maximum flow of a network is equal to the sum of the capacities of the minimum cut of the network.

The test used is shown below, simplified:

```
generateRandomGraph(); // generates a random network with N nodes
FFMaxFlow = fordFulkerson(); // Runs Ford-Fulkerson algorithm on this network,
                             // and returns maximum flow found
cut = findMinimumCut();
cutMaxFlow = removeEdgesInCut(); // Removes edges in minimum cut from network,
                                 // and returns the sum of their capacities
path = findPath(); // Attempts to find a path from S to T
if(path != null){
   min-cut is invalid, test has failed
}
if(FFMaxFlow != cutMaxFlow) {
   max-flow min-cut theorem shows test has failed
} else {
   test has passed
}
```

A function *testFF* ran the test. It was used in two ways:

1. *testFF*($n$, *iter*) would take a value for $N$ ($n$) and a value for the number of iterations (*iter*). It would then execute the test *iter* times, and then print out the percentage of incorrect tests at completion.
2. *testFF*() would run until failure, at which point it would print out the network data used in the failed test.

The former way was used to uncover the presence bugs in the implementation, as subtle or rare bugs could be discovered with higher values of $N$ and more iterations. The latter was used to debug them with smaller sets of data (eg $N \approx 5$). When a failed test returned, the data from it could be entered into the application and the animation could be run to determine the cause of failure.

When the correctness testing was first implemented, it revealed that the implementation was incorrect 0.05% of the time on average. A bug partially behind the failure rate was caused during the updating of the residual graph, which was then fixed (Figure 5.6).

Failure rate during the second round of testing was less than before, but was still non-zero. The cause of this was also from the residual graph being updated, in rare cases when a backwards edge would be used multiple times. Once this was fixed, any value of $N$ or iterations would return no failed tests. This satisfied the first of the questions to be answered by the evaluation.

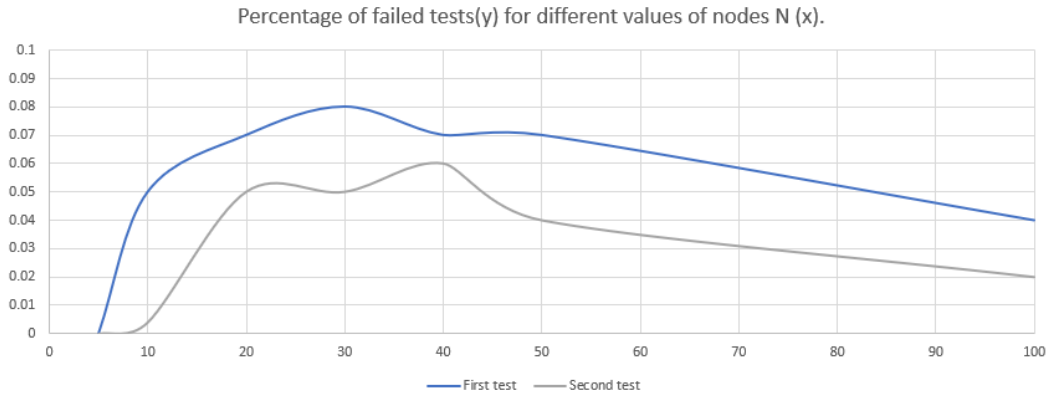



***Figure 6.1:*** *The application was tested with varying numbers of nodes from 5-100 with 10000 iterations for each value. This figure shows the drop in failure rates between the first and second tests (not the final results).*

## 6.2 User evaluation

To evaluate the UI and effectiveness of the animation, usability tests were conducted on eight participants, all of which had had prior experience of the Ford-Fulkerson algorithm in their courses. The evaluation was split into two sections. The first section covered a series of tasks to create graphs, and the second section covered the animation and its effectiveness. To begin the evaluation, participants were asked to rate their understanding of the Ford-Fulkerson algorithm on a scale of 1 to 10, where 1 is no understanding at all and 10 is complete understanding.

The questionnaire used in the evaluation is provided in the appendices, along with a signed ethics checklist form.

### 6.2.1 Section 1: Graph generation

After this, the first section of the evaluation began. The participant was asked to complete three tasks:

1. Generate a random graph with 8 nodes
2. Download the graph from the previous task, and add a new edge (in the file) between any two nodes. Upload this altered graph.
3. Draw this example graph (Figure 6.2), and save it.

Each task was timed, and any critical or non-critical errors were noted. The tasks were designed so that the user would have to explore each feature of graph generation. The purpose was to

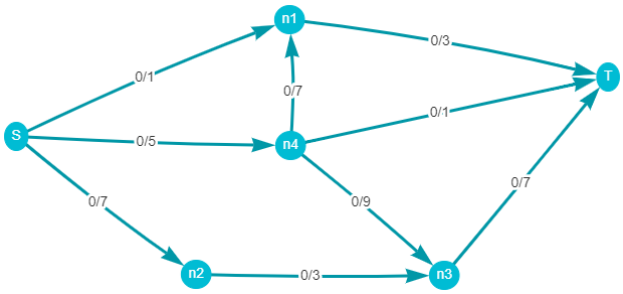



***Figure 6.2:*** *The example graph evaluation participants were asked to draw.*

| Task | Control Time | Mean Time |
|---|---|---|
| Generate Random | 00:04 | 00:10 |
| Upload/Download | 01:04 | 01:20 |
| Draw | 01:27 | 01:48 |

***Figure 6.3:*** *Comparing the mean times to complete the three tasks.*

check that the design of each feature was intuitive. I timed myself performing each of these tasks to compare the results to, shown under “Control Time" in Figure 6.3.

There were no critical errors for any tasks, all participants were able to complete their tasks within a few minutes. Three Participants experienced non-critical errors with Downloading a graph as a JSON file and altering it:

- Added a duplicate edge.
- Trailing comma, and added edge to non-existent node.
- Trailing comma.

Four participants encountered non-critical errors with drawing a new graph:

- Accidentally cleared progess. The ‘enter’ button clears canvas when nothing selected.
- Tried to edit edge capacity, not possible.
- Not clear on “capacity" meaning (entered 0/6 when prompted).
- Could not find “draw graph" right away.

After these tasks, the following qualitative questions were asked:

1. Was there anything you liked about generating a new graph?
2. Was there anything you disliked about generating a new graph?
3. Do you have any recommendations for graph generation?

The results for the qualitative statements are grouped into common categories and represented in Figure 6.4.

Comments regarding candidate’s positive opinions on graph generation often mentioned that it was intuitive and easy to use, with one participant stating that they "didn’t have to read much" and another saying there were "no extra distractions". Participants also enjoyed the “wiggly" appearance of the edges when nodes are dragged: this is categorised under graph animation.

The most common aspect that candidates disliked about graph generation was that to add multiple nodes or edges when drawing graphs, the ‘add node’ or ‘add edge’ button must be pressed after each addition.

This was brought up twice when participants were asked if they had any recommendations for graph generation. In addition, one participant recommended clearer instructions for the

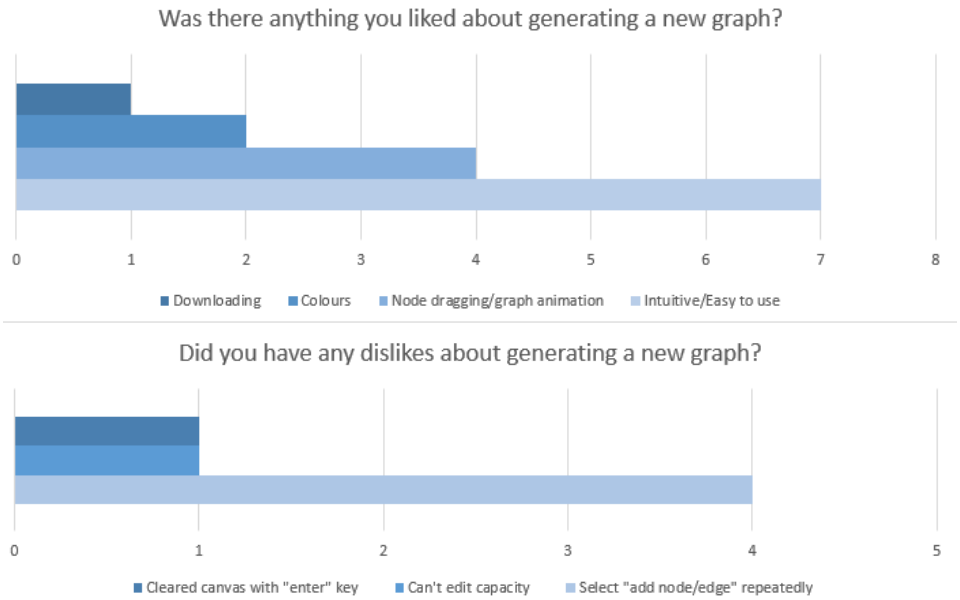



***Figure 6.4:*** *Common categories of candidate's opinions, grouped by rate of occurrence.*

upload/download feature, pointing out that the JSON example was not correctly formatted JSON. Edge capacity was also mentioned twice in the recommendations, one suggesting it should be entered via a pop-up rather than an alert, and another candidate wishing to edit the capacity of an edge without replacing it.

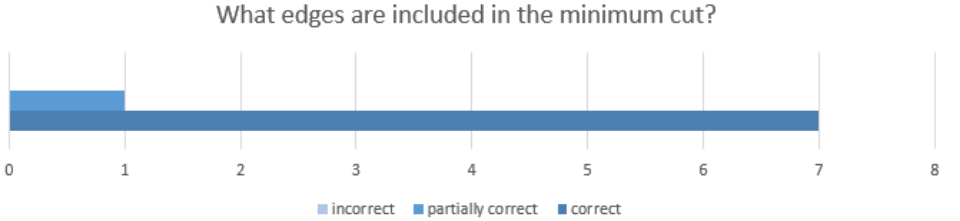
### 6.2.2 Section 2: Animation

In this section the participant was asked to: "Run the animation for the current graph. Utilise the playback features as much as you need to try and best understand the algorithm. This is not timed.".

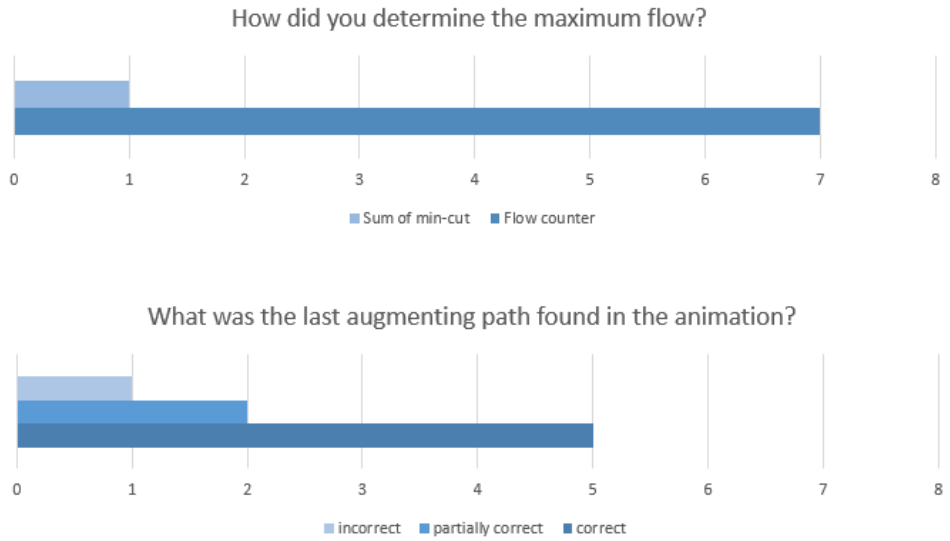
After they stated that they had completed this task, the following questions would be asked:

1. What edges are included in the minimum cut?
2. What is the maximum flow of this graph?
3. How did you determine the maximum value?
4. What was the last augmenting path found in the animation?

Each of the answers could be answered by using different components shown on the screen. The answers to questions 1, 2 and 4 were able to be quantified into correct, partially correct (for questions 1 and 4), or incorrect. Question 3 was to test that the flow counter component was clear. Question 4 was designed to make the participant check the execution trace for the answer.



The participant who had a 'partially correct' response to the edges in the minimum cut also stated that they did not know at all what a minimum cut was.




Participants were advised that the answer to the last augmenting path on the screen. Answers marked "partially correct" had difficulty locating the execution trace line.

The participants understanding of the animation was then evaluated by rating the following statements on a scale from 1 to 5, where 1 is totally disagree, and 5 is totally agree.

***Table 6.1:*** *The responses to the statements, grouped by mean and mode.*

| Statement | Mean | Mode |
|---|---|---|
| I was able to tell when the residual graph was being built. | 4 | 5 |
| I could tell when the augmenting path was found, and where it was. | 3.5 | 4 |
| It was clear to me when each edge of the augmenting path was augmented/decremented. | 4.1 | 4/5 |
| It was clear to me which edge of the residual graph was being updated. | 4.4 | 5 |

Then the following qualitative questions were asked:

1. Were there any parts of the animation that weren't clear to you?
2. Are there any features of the playback/animation that you liked?
3. Are there any features of the playback/animation that you disliked?
4. Do you have any recommendations about the animation?

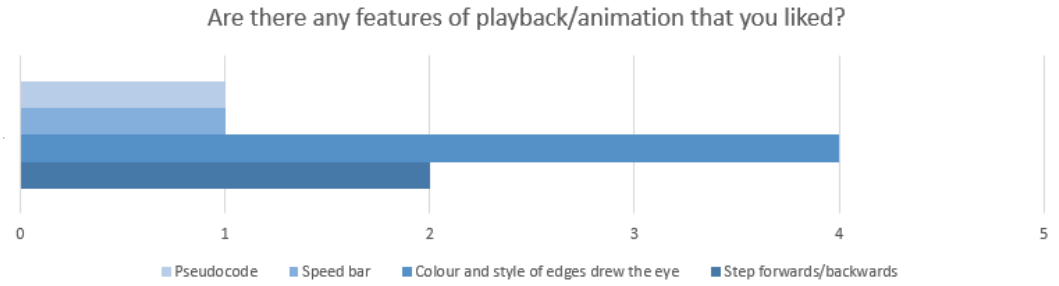



***Figure 6.5:*** *Common categories of candidate's positive opinions, grouped by rate of occurrence.*

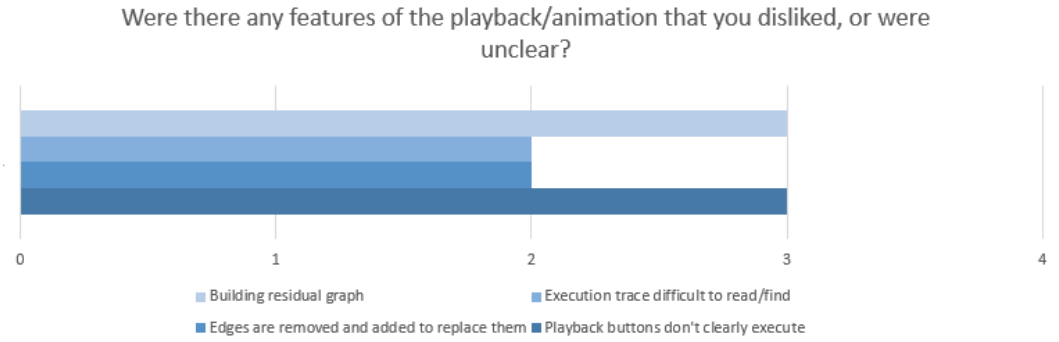



***Figure 6.6:*** *Common categories of candidate's negative opinions, grouped by rate of occurrence. Due to lots of overlap between the responses to the qualitative questions 1 and 3, they have been merged here.*

Finally, participants were asked to re-evaluate their understanding of the Ford-Fulkerson algorithm in the same way as the start of the evaluation. They were also given the opportunity for any additional comments on the project.

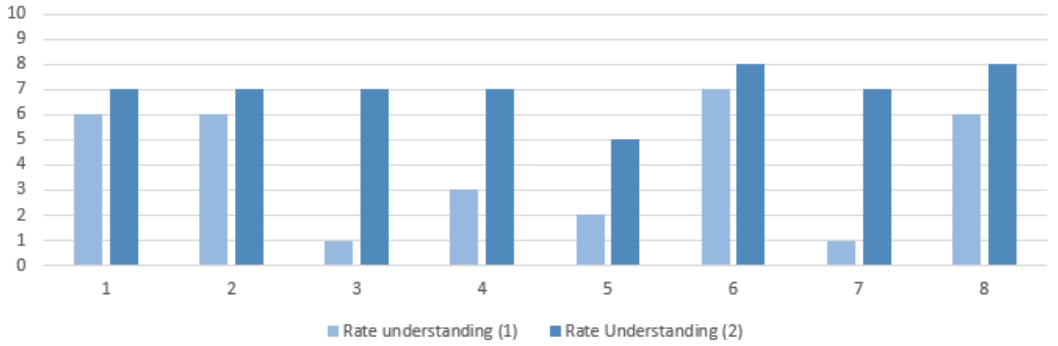



***Figure 6.7:*** *The responses for the candidate's understanding of the algorithm, before and after evaluation.*

## 6.3 Discussion

The mean time of the participants to complete graph generation tasks, when compared to the developer time to complete the same tasks, show similar results. None of the participants had ever

interacted with the application before, yet the difference in times were relatively low (Figure 6.3).

Aspects that slowed down users were usually parts with written instructions, such as incorrectly formatted JSON files, instructions to enter the capacity not being clear, edit edge only allowing change to its connecting nodes and not capacity (as would be reasonable to assume).

Qualitative statements also support this. Many candidates would have preferred to not have had to repeatedly press 'add node/edge', but this was still an aspect learned quickly.

When asked the questions: "*What edges are included in the minimum cut*" and "*What is the maximum flow of this graph?*" after the animation, participants were able to quickly and correctly answer, using the components on the screen.

However when "What was the last augmenting path found in the animation?" was asked, all hesitated, and some were not able to find or had not noticed the execution trace panel. This was mentioned often in the results of the qualitative feedback evaluating users dislikes, or parts they did not understand.

Also mentioned an equal number of times was that the playback buttons were not always clear in purpose or execution. No candidate struggled to run the animation due to the familiarity of the 'play/pause' icon, but some had difficulty quickly understanding what the other buttons represented. Some candidates on discovering the 'step forward/backward' feature commented that they would have rather known sooner that that had been an option.

While the execution trace panel and playback controls detract from the ease of use of the application design, both the quantitative and qualitative results point to the design and user interface being intuitive to use.

The statements were designed to evaluate if the user could recognise or understand different stages of the algorithm, based on the animation. On the whole, the results were a success. The statements with the highest rates of understanding were both related to the residual graph:

- *I was able to tell when the residual graph was being built.*
- *It was clear to me which edge of the residual graph was being updated.*

This is an understandable result, as these stages of the animation were the simplest: an edge in the top graph would be highlighted, and the corresponding edges in the residual graph updated to match. It was also enhanced by the physics of the edges themselves, as when two edges connect the same two nodes, if either are updated they will spring about before settling. One user commented on this specifically.

The third statement, "*It was clear to me when each edge of the augmenting path was augmented/decremented.*", also returned confident responses. I think this is due to the extra step in highlighting each edge in the top graph in orange before changing the label, to draw the user's eye, and then changing it to grey once updated, so that the next edge along the path has the same impact.

The statement with the lowest results was "*I could tell when the augmenting path was found, and where it was.*". The responses were much more varied here, and there may be multiple factors contributing to this. The most likely is that the animation in this section is more complex and comprised of more stages, handling multiple edges across the residual graph as opposed to changing one at a time. Another factor could be that the statement itself is too ambiguous. To me, the point where the augmenting path is found is the point where it is highlighted in orange along the residual graph. To a user not too familiar with the animation, it could be at any stage of searching for, finding, or updating the augmenting path.

The colour and styling of the graphs and the ability to step forwards or backwards through the animation were most commonly mentioned when participants were asked what they liked about about the playback/animation.

There were a few points in the design of the playback and animation which hindered the understanding of the algorithm. Most notably of these was the execution trace not being clear, with one user commenting that they wish they had noticed it nearer the beginning. Another was parts of the animation not being as clear as possible, such as the order in which the residual graph is built, or some styles (such as dashed lines) being used for different purposes, rather than a consistent one.

However there is evidence that the presentation of the algorithm did strengthen the user's understanding, in general. The clearest of this is the results from when the user evaluates their own understanding of the algorithm from 1-10 at the beginning of the evaluation, and then again at the end. Every participant reported an increase in understanding, with the average increase being 2.4, and the average second rating being 7.

Recommendations in this area suggested that the execution trace panel was under-used, as it was commented on by five separate candidates. Two of these mentioned that it wasn't obvious enough, and another two suggested that it should print out when a button is pressed in the interface, to increase feedback for the buttons. Another suggestion was that the user could be able to select a line in the panel to jump back to that stage in the animation. Some of the styling was brought up in the recommendations, with one user suggested that the label text should be changed when updated to attract attention, and another desiring a brighter colour in the top graph when the residual graph is being built. Finally, a participant said they would like be able to jump back to the very start of the animation, rather than rewinding. All of this feedback could make for reasonable future work.

# 7 Conclusion

## 7.1 Summary

The aim of this project was to animate the Ford-Fulkerson algorithm, in a way which is easily accessible and suitable for use in the context of a lecture or revision. The research into animated algorithms show that they can be an effective tool in quickly learning the algorithm, but it is necessary to present animation well in order to see a difference. The project was created as a web application, so that anyone with the link may access it. Analysing similar examples of algorithm animation showed that animating network flow had not been attempted by many before, as after extensive searching only one such example could be found.

The application provides multiple ways for user to manipulate graphs and playback, so that they have a large amount of control over the animation. Additional information included a pseudo-code and logging panel so that the user could know the details of what is showing on screen.

Correctness testing used the Max Flow - Min Cut theorem to evaluate the implementation. It revealed some rare bugs in the residual graph, but once attended to the failure rate of the tests dropped to 0. Usability testing showed that users liked the interface and found it intuitive at many points, but revealed there were still areas for improvement, such as the execution trace panel not always being clear, and playback buttons not giving enough feedback on interaction. Each evaluation participant was asked to rate their understanding of the algorithm both before and after the evaluation. Despite some shortcomings in the interface, every response showed an increase in understanding.

## 7.2 Future Work

A 'Could have' requirement and two 'Would like' requirements are still unimplemented:

- Animate the separate algorithms used in Ford-Fulkerson (like path searching, or different forms).
- Skip to end or beginning of animation.
- Mobile-friendly interface (for use in lectures).

Not having a mobile-friendly interface may not be a detraction in hindsight. Using the animation on a mobile screen might not be practical, as there is a lot of additional information that may not fit or be too small to see.

However, being able to skip to the end or beginning came up in qualitative parts of the evaluation, so clearly there is a demand for this and it would be a small but effective change. The animation of separate algorithms used in Ford-Fulkerson, or other network flow algorithms could take the project in an exciting direction, using the interface to host a suite of algorithm animations.

Evaluation participants had some really constructive ideas for improvements of the interface, as well as new features. Seeing the importance of the execution trace panel means that a redesign of the interface could be necessary to give it more space on the screen. Also there could be many improvements to the feedback that the playback controls provide.

## 7.3 Reflection

Calculating network flow is a process which involves many different stages and factors. There were some deceptively complex stages of the algorithm, such as updating the residual graph along just the augmenting path. There were some oversights which only caused bugs on very specific graphs, and it was easy to forget to accommodate for backwards edges at points due to their rarity.

The Ford-Fulkerson algorithm also needs a lot of information at points which can't always be expressed graphically. One of the biggest challenges was trying to communicate enough detail to teach the algorithm effectively without overloading the user, and losing engagement. Usability and front end design is not something I've experienced a lot of, so these aspects of the project pushed me out of my comfort zone.

When looking at similar examples of this project I noted the ambition of *visualgo*'s application compared to *algomation*'s simplicity, yet the final product of *algomation* is much more pleasant to use. No feature was added unless it could be made robust, as I wanted to ensure that the user felt they were interacting with a 'complete' application rather than a rushed one. Despite this, I feel as though I have sufficiently fulfilled all the end goals of the project, and a significant proportion of the MoSCoW requirements. In general the end result of this project is a product that I feel I can be proud of.

# 7 Bibliography

[1] http://www.doc.ic.ac.uk/ nd/surprise_95/journal/vol2/ad1/article2.html

[2] https://www2.cs.duke.edu/csed/rodger/papers/wgReport02.pdf

[3] Do Algorithm Animations Aid Learning? Michael Byrne, Richard Catrambone, John T. Stasko.

[4] http://www.algomation.com/algorithm/prim-minimum-spanning-tree

[5] https://visualgo.net/en/maxflow

[6] www.cs.usfca.edu/ galles/visualization/BFS.html

[7] http://visjs.org/docs/network/

[8] http://visjs.org/docs/data/

[9] http://visjs.org/docs/network/nodes.html

[10] http://visjs.org/docs/network/edges.html

[11] https://materializecss.com/about.html

## School of Computing Science
## University of Glasgow

## Ethics checklist form for 3rd/4th/5th year, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

**If no other people have been involved in the collection of information, then you do not need to complete this form.**

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee (matthew.chalmers@glasgow.ac.uk) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.

   *Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback*

2. The experimental materials were paper-based, or comprised software running on standard hardware.

   *Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.*

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

   *If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.*

   *Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.*

4. No incentives were offered to the participants.

   *The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.*

5. No information about the evaluation or materials was intentionally withheld from the participants.
   *Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.*

6. No participant was under the age of 16.
   *Parental consent is required for participants under the age of 16.*

7. No participant has an impairment that may limit their understanding or communication.
   *Additional consent is required for participants with impairments.*

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   *A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.*

9. All participants were informed that they could withdraw at any time.
   *All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.*

10. All participants have been informed of my contact details.
    *All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.*

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    *The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.*

12. All the data collected from the participants is stored in an anonymous form.
    *All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.*

---

**Project title** Animating Network Flow Algorithms

**Student's Name** Lucy MacPhail

**Student Number** 2783332

**Student's Signature** LMacPhail

**Supervisor's Signature**

**Date** 21 Feb 2019

# Evaluation of personal project

On a scale from 1 to 10, with 1 being no familiarity at all and 10 being complete understanding, how would you rate your current knowledge of the Ford Fulkerson algorithm? ____

## Graph generation

| Task | Errors during task | | | Time taken |
|---|---|---|---|---|
| | Critical errors | Non-critical errors | Error free (T/F) | |
| Generate a random graph with 8 nodes | | | | |
| Download the graph from the previous task, and add a new edge between two nodes of your choice. Upload this new graph. | | | | |
| Draw this graph (some example graph provided), and save it. | | | | |

Was there anything you liked about generating a new graph?

Was there anything you disliked about generating a new graph?

Do you have any recommendations for graph generation?

## Animation

**Task: Run the animation for the current graph. Utilise the playback features as much as you need to try and best understand the algorithm. This is not timed.**

What edges are included in the minimum cut? [Correct/Partially correct/Incorrect]

What is the maximum flow of this graph? [Correct/Incorrect]

How did you determine the maximum flow? _____

On a scale from 1 to 5, with 1 being complete disagreement and 5 being complete agreement...:

- I was able to tell when the residual graph was being built. ___
- I could tell when the augmenting path was found, and where it was. __
- It was clear to me when each edge of the augmenting path was augmented/decremented. __
- It was clear to me which edge of the residual graph was being updated. __

What was the last augmenting path found in the animation? [Correct/Partially correct/Incorrect]

Were there any parts of the animation that weren't clear to you (maybe demonstrate)?

Are there any features of the playback/animation that you liked?

Are there any features of the playback/animation that you disliked?

Do you have any recommendations about the animation?

Could you rate your understanding of the Ford-Fulkerson algorithm again on a scale from 1 to 10? ____

Do you have any additional comments about the project?