Circuit Simulpaper

Authors: Jaden D'Abreo, Stephen Dai, and Devan Grover

Department of Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—This project is intended to serve as an educational tool for children in middle school to learn basic circuit functionality through drawing. The project will be implemented by using a computer vision algorithm to identify components on a hand drawn circuit, feed the identified circuit into a circuit simulator, and display the analyzed circuit, all of which will be done through a mobile application. This system hopes to serve as a fun and safe way for children to learn about circuits through the appeal of drawing and without the risks of electrical components.**

***Index Terms*— Circuits, DC Analysis, Computer Vision, Simulation**

I. Introduction

ELECTRICAL circuits are expensive and potentially dangerous to experiment with. Not all students have the resources available to them to learn about circuits at a young age. To purchase the bare minimum components to create a simple circuit (breadboard, wires, resistors, power supply) people must spend at least forty dollars, which is a luxury that everyone cannot afford. Furthermore, improper education can lead to dangerous situations. If a student unknowingly causes a short circuit or improperly uses a component like a multimeter, they can harm themselves and their equipment.

When learning about circuits, students typically first learn about the different symbols designating various electronic components. Learning to draw circuits and using the proper symbols are imperative steps in the process of mastering circuits. Young students are also often fond of drawing and recent studies show that drawing serves as one of the most effective ways to retain knowledge[3].

Our application hopes to solve the issues of a lack of accessibility and safety when learning about circuits and capitalize upon young students' fondness of drawing. We are creating an application in which users can take a picture of a schematic they draw, upload the picture, and then receive a simulated version of the circuit they drew with voltages and currents labeled.

This application is primarily targeted towards middle school students. Students at this age are old enough to learn the basics of circuits, have access to mobile applications, and still indulge in drawing through school. Currently, there are no technologies that allow a user to upload a drawn circuit and have it analyzed. In addition, all circuit simulators online are accessed through web applications, thus less accessible to our primary use group. Furthermore, there are no applications that accomplish the intended goals of this project.

II. Use-Case Requirements

The use case requirements for our application encompass many factors of our application, with a focus on accuracy and usability. The use case requirements are as follows:

1. *The computer vision algorithm must have an individual component detection accuracy of 90%*

To ensure a good user experience for our application, being able to properly identify components is required. Users should not have to constantly take pictures of their circuits to get components to be identified properly.

2. *The computer vision algorithm must have a circuit detection accuracy of 90%*

We want to not only ensure each component is detected with an accuracy of 90%; we also want to ensure that all circuits are detected with a 90% accuracy. We will achieve this by using an algorithm to provide users with the five circuits our computer vision algorithm has the highest confidence in. Users will then be able to select which amongst these five circuits is the correct circuit that they drew.

3. *The circuit simulator must simulate circuits with 100% accuracy*

Our circuit simulator must have perfect accuracy, otherwise the application cannot be used as an educational tool. Given an input circuit, our simulator must output the circuit with the correct voltages/currents at each node/component.

4. *The application's user interface must receive an average rating of 80%*

We will conduct user testing of our application on the target group of middle school students and provide them with a survey to record their feedback after testing. The survey questions must all have an average score of at least 8/10.

5. *The application must be free to the user*

To ensure that most children can use our application, we need to make it free. This app was created so that children can learn about circuits without having to spend money on components, therefore we must make it free.

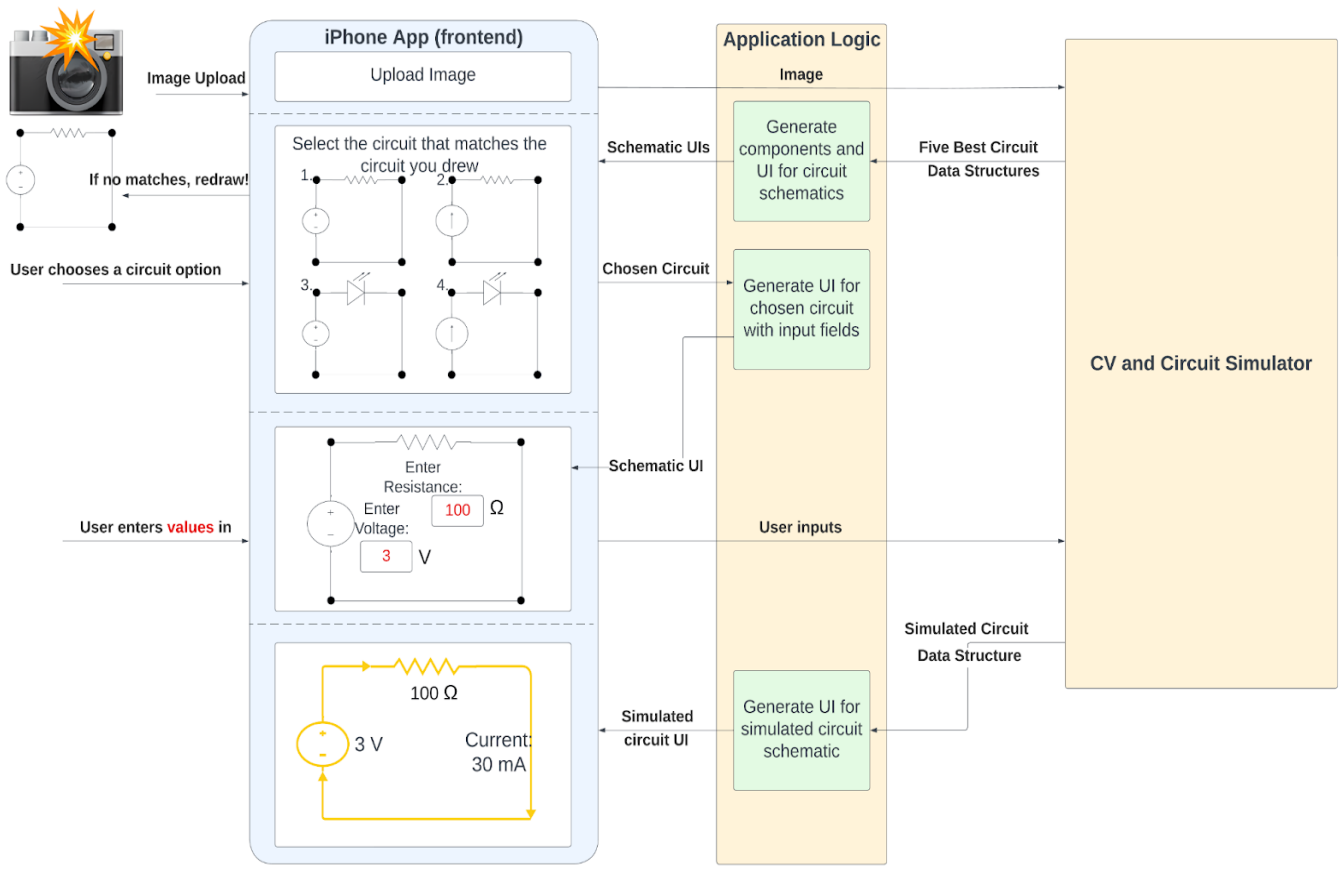

Fig. 1. System Architecture Block Diagram

III. Architecture and/or Principle of Operation

Our project is split up into three main parts: the computer vision system, the circuit simulator, and the frontend application. A block diagram of our application's high level functionality is shown in Fig. 1 above.

The user will first draw a circuit diagram for the circuit that they want simulated on a piece of paper. They will have to draw the circuit with dots at the ends of the components' terminals and with the components oriented horizontally or vertically. The application will have a built-in feature in which the user can either take a picture or select a picture from their camera roll to upload. Users will use this feature to upload the image of the circuit they drew to our application.

Once the user has uploaded their circuit image, it will get sent to our computer vision algorithm for processing. The computer vision algorithm will first perform preprocessing on the input image to remove noise and make nodes easier to detect. Once the image has been preprocessed, a circle detection algorithm is used to detect the nodes at the ends of each component. The algorithm will then detect all the components between nodes and create subimages of each component in the circuit.

Once the components are detected, ORB will be run on each subimage. ORB is an algorithm that generates keypoints and descriptors based on the input image. Keypoints are parts of the image that ORB determines are distinct. The descriptors are extracted from the keypoints and each one is a binary string of encoded information about each keypoint. These descriptors can then be used to match features with different images. After running ORB, we will use brute force matching to match the descriptors with precalculated descriptors from our reference dataset stored locally within the application.

At this point, we have scores associating the component subimage with a type of component, like resistor or voltage source. We will take the top three most matching component types for each sub image and generate every combination of circuits that can be made. For each of these circuits, we generate another score that considers the individual component scores and a likelihood score that prioritizes well-formed over malformed circuits, such as the circuit having a power source instead of none. The algorithm will output each circuit as a netlist for our frontend and circuit simulator to use.

The five highest scoring circuits will be shown to the user on the application's user interface. The user can then select which of the shown circuits was the one that they drew. After selecting the circuit, they will enter values for each component (resistance, voltage, etc.). The circuit will then be sent to our circuit simulator, which will use modified nodal analysis to simulate the circuit. Once the circuit has been simulated, it will be sent to the frontend where the circuit will be rendered.

Users will then be able to see a digital version of the circuit that they drew with the voltages and currents annotated. Each component will have a labeled current going through it, and each node will have a labeled voltage.

IV. Design Requirements

A high level description of the design requirements can be seen below in Table 1:

Table I. Design Requirements

Description	Requirement
Minimum iOS Version	8.0
Minimum Image Size	1920x1080 pixels
Minimum Component Size	350x200 pixels
Maximum Application Size	100 Megabytes
Maximum Number of Components	8
Supported Components	Voltage/Current Sources, Resistors, Bulbs, LEDs, Switches

We want to make sure that our app is easily accessible to all, even if they do not have the newest and best iOS devices. For this reason, we hope to achieve a minimum iOS compatibility of iOS 8.0. OpenCV can work on all iOS versions including and after 8.0, which is why we chose this specific version. This allows the application to be compatible with all iPhones since the iPhone 6 and with all iPads since the iPad 2. We also want to keep the application size low so that users do not have to uninstall or delete existing items from their devices to install our application. By looking at the size of other applications with similar functionality like "Tiny Scanner", we decided that our application would have a maximum size of 100 MB. These requirements are all set to increase accessibility to our application.

We also have restrictions on the components to maintain the effectiveness of our computer vision algorithm. We currently have a limit of eight components maximum per drawn circuit to ensure there is adequate spacing between each component and to ensure all components can fit on an image while maintaining the size requirements. The components must also be drawn horizontally and vertically at near-right angles so that we can easily detect components between nodes. We also are only performing steady-state DC analysis on the following components: voltage sources, current sources, resistors, bulbs, switches, and LEDs.

There are also requirements for the picture that users input into our application for analysis. The picture must have a minimum size of 1920x1080 pixels. All our supported devices except for the iPad 2 can take pictures at or above this resolution. This constraint is required so that our computer vision algorithm can easily identify components and nodes in the circuit. Similarly, there is a minimum component size of 350x200 pixels. This means that users will need to take close-up pictures of their circuit. Without this constraint, the components will be hard to identify.

V. Design Trade Studies

A. Offline Application

When running our application, we decided to implement it with no need for internet connection. Due to a large portion of our target audience not having access to the devices with an internet connection[4] we decided that keeping the app offline will strengthen the argument for the accessibility use case and extend to a larger audience.

This means that we will be storing everything locally through the user's device. All that needs to be stored is reference data for dataset components. As one of the main benefits of having an online application is having access to large amounts of storage, we expect that our application will not exceed more than 100 MB. Most offline apps are anywhere between 20 to 100 MB in size, and ones that perform image parsing tend more towards the higher end of 100 MB. For example, the offline mobile application "Tiny Scanner", which generates PDFs (Portable Document Format) from pictures, has an app size of 92.5 MB, which does not include document and data storage. Because this application generates and stores PDFs, it takes up lots of documents and data storage. For our application, we will not store images of previously simulated circuits because a user can always just reupload an image of the circuit from their camera roll. Thus, we consider the 92.5 MB as an appropriate benchmark for our application storage. By constraining the amount of storage our application will use, we believe keeping the application offline will reinforce the goals of our project while also functioning the same as an online application.

B. Mobile Application

For how we want to display our application, we decided to implement a mobile application. We needed to decide between either a mobile application or a web application and after much thought we decided that a mobile application not only aligns with our use case of accessibility more. If the project progressed with a web application the user would have a much harder time uploading their drawing and uploading their picture rather than just doing it on their phone. In addition, this application is intended to be offline due to accessibility, thus a web application would not be possible to implement. Recent statistics show that around 71% of 12 year old have phones and by age 14 roughly 91% have phones[5]. Furthermore, a different study stated that 95% of U.S. teens have access to a smartphone at home, while only 88% have access to a computer [1]. This comes out to around 2 million people of our target audience. As phones are much more of a necessity than laptops around the ages of 12-15, we felt like tailoring the application to be used on a phone would align more with our goal of making this project an accessible approach to help educate all those that are interested in circuits. Therefore, we decided that these reasons were enough to pursue the mobile application.

C. Application Stack

Swift is the most popular framework used for iOS applications today - almost all applications nowadays use it. Swift is a more modern language and easier to learn when

compared to Objective-C, which is why we are using it for our frontend. Objective C has traditionally been used to develop iOS applications because it has existed for nearly forty years. Swift, which is much newer, is also 2.6 times faster than Objective-C[8] and has memory and type safety built in. We will have to use Objective-C as well because OpenCV is only available as an Objective-C library. Therefore, the backend of our application will have to use Objective-C. To allow the Swift application to interact with the Objective-C application, we will need to create a bridging header file to bridge the Swift and Objective-C code.

D. Computer Vision Architecture

We opted to use a more traditional computer vision architecture that does not utilize a neural network. Given that we decided on creating a mobile application that does not require the internet, we cannot feasibly utilize a neural network as such. Loading and using a neural network locally in-app is too computationally and storage intensive for a mobile phone, especially for older iPhones. Ultimately what we considered is how much of a difference there is in terms of accuracy when using and not using a neural network. Some research that has been previously conducted has achieved a 95% accuracy in classifying electrical components using a CNN (Convolutional Neural Network)[9], and additional research has been conducted to achieve a 90% accuracy using KNN (K Nearest Neighbors) without a neural network [10]. Given we are also not using a neural network, we use this 90% as a benchmark for our design as well. We acknowledge that using a neural network would greatly increase the accuracy of classifying individual components, but this would come at the expense of integrating our mobile application with a backend server that can support the neural network. Because the image preprocessing and usage of ORB would stay the same, we have left the integration of a neural network for work that can be done post-MVP to further improve our accuracy goals if desired.

E. Circuit Segmentation

In order to be able to detect a circuit, we need to know what individual components make up the circuit. To know what the individual components are, we need to generate separate subimages of each component in the circuit and feed those subimages into a classification workflow. The most intuitive way to separate components in a circuit is by looking at pairs of adjacent nodes in a circuit, because a component is always between them. Because we also use nodes in netlists and to perform nodal analysis, first detecting the nodes in a circuit is the most intuitive first step in our computer vision subsystem.

We considered two solutions to detecting nodes: Hough line transform and Hough circle transform. Hough line transform would be used to detect the line parts of components that represent their ends, and we could denote the existence and location of a node as the point where two lines intersect. For Hough circle transform, we would require that every node be drawn as a circle, and the transform will just detect every circle in the circuit as a node. After testing both implementation ideas, we ultimately decided on using Hough circle transform and requiring the user to draw nodes as circles.

From testing with Hough line transform there were two fundamental problems. The first problem was that the line transform does not perform well with hand drawn lines. Hand drawn lines are usually never straight, and the transform consistently breaks down one line representing one end of a component into multiple connected lines. The other critical problem is that there are many lines that intersect in circuits that do not represent nodes. For example, the "+" symbol in a voltage source is made of two lines that intersect at 90 degree angles, which is exactly what we expect for our nodes. Thus, using Hough line transform is highly unreliable for us to use for node detection.

We believe that requiring users to draw nodes as circles is extra work for the user but has benefits that outweigh the extra effort needed. Detecting the nodes is arguably the most important step in the circuit detection: if one node is missed, that means that an entire component will be missed, rather than just one component being classified incorrectly. This means that properly detecting the nodes is critical to achieving our circuit detection accuracy marks. With the proper image preprocessing, node detection is highly reliable with Hough circle transform, which will be discussed later. Also, we believe that being able to identify where nodes are in a circuit is beneficial to the learning of our users. Being able to recognize points of shared voltages and know where current diverges is incredibly useful in doing elementary circuit analysis. Thus, we chose to use Hough circle transform and require users to draw filled-in circles for their nodes for increased circuit detection accuracy and for their own learning sake.

F. Feature Detection Algorithm

We primarily considered two feature detection algorithms, ORB and SIFT, and ultimately decided on using ORB. For feature detection, ORB uses the FAST algorithm, which identifies pixels that have a high gradient and arranges them in a circular pattern. SIFT uses a DoG (difference of Gaussian) pyramid to search for scale-space extrema, which is an expensive compute process. We chose ORB because it is rotation-invariant and more lightweight than SIFT, which we describe below.

What is desirable about both feature detection algorithms is that they are scale-invariant. This means that features that are the same but differently sized still get classified as similar features. This is important because users will likely draw components that aren't all the same size as each other or the components in our dataset. But, only ORB is inherently rotation-invariant. Without rotation invariance, if you took the same image and rotated it, features would be detected differently. Rotation invariance is important for our work because while we require components to be either horizontally or vertically aligned, some may be drawn with slight offsets in angle. Fig. 2. contrasts the number of matched features with

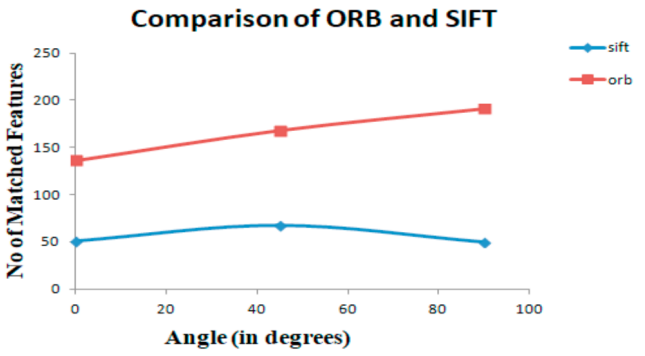

Fig. 2. Difference of number of matched features when running ORB & SIFT on an image and a copy of the image rotated by various degrees

ORB and SIFT when comparing an image with its rotated self, with ORB clearly performing better across all degrees of rotation. SIFT does actually perform better given in-plane rotation but given that our image features are purely two-dimensional and the image of the circuit is always taken face-on, this is unimportant for our use case. The other key difference is that ORB is more lightweight. SIFT is more computationally intensive because it must construct DoG pyramids and calculate gradient histograms. Additionally, SIFT descriptors are larger than ORB's BRIEF descriptors. SIFT descriptors are 128-dimensional floating-point vectors, spanning 512 bytes each. On the other hand, BRIEF descriptors are only binary strings that are 32 bytes each. Shown in Table 2 is a size comparison between SIFT and BRIEF descriptors run on a dataset of component subimages. Although the number of descriptors generated from ORB is five times that of SIFT, because of how large each SIFT descriptor is, the average number of bytes for one component is 3.5 times less with ORB. Because we are making an iPhone application and storing the dataset locally (in-app), we want a less computationally intensive algorithm and to minimize allocation of the user's storage, hence ORB is our feature detection algorithm of choice.

TABLE 2. COMPARISON OF STORAGE USAGE WITH ORB AND SIFT

Algorithm	# of Components	Avg. # of Descriptors	Avg Bytes per Component	Total Bytes
SIFT	13	55	28681	372864
ORB	14	289	9401	131616

G. Image Preprocessing

Before feeding images into our computer vision algorithms, we want to perform preprocessing to better isolate the user's drawing and get rid of unintended features like shadows and light marks on the paper. Upon receiving the user's image, we decided on first performing thresholding and median blurring. Thresholding is useful because it generates a binary image, and we can set the threshold to isolate the darker markings that would come from concentrated amounts of pencil lead or ink and ignore the effects of light shadows and accidental markings. Because the first step in our circuit detection is detecting where each node is and nodes are filled-in circles, thresholding will best isolate these nodes.

After thresholding, we decided on using median blurring over other blurring algorithms such as Gaussian blurring and bilateral blurring. Median blurring is most effective after thresholding for our use because it is the best in removing salt-and-pepper noise and details. After we perform thresholding, it is likely not only the filled-in nodes are left, but some darker pen/pencil spots from the components as well. Because we want to isolate the nodes, median blurring sees the thin parts of the drawing that correspond to the components as noise (the pepper) and removes them. Additionally, sometimes thresholding will create white spots in the filled-in circles because the darkness of the circle is not uniform. Median blurring will fill in these white spots (the salt) because it sees it as noise, creating filled in circles. This way there is no possibility of a circular white spot in the node to also get detected as a circle/node.

Gaussian blurring and bilateral blurring are similar, and both use Gaussian distributions to remove noise, and bilateral blurring uses two separate distributions instead of one. They both produce more of a motion blur and retain features of an image better compared to median blurring, which we don't want for the image preprocessing for node detection. But, we do want this effect before doing edge detection on individual component subimages, which is separate from the node detection. For this, we chose bilateral blurring. We use bilateral blurring because unlike Gaussian blurring, it keeps edges sharp while blurring the rest of the image, which is exactly what we want to perform edge detection. We use Canny edge detection instead of thresholding before feature detection because while both generate binary images, thresholding will more likely eliminate entire parts of component drawings if the pen/pencil lead is too thin and light. Additionally, we care more about the outline of the component as features, so edge detection is the most appropriate for detecting individual components.

H. Value Detection

Rather than having users write down the values for each component next to the component, we elected to have users type the values onto the application itself. Although there are many libraries available for text detection, it would be hard for the computer vision algorithm to figure out which value corresponds to which component. This can lead to dissatisfied users because they may have to constantly redraw and take pictures of their circuit until it gets properly identified. Once we reach our MVP, we are considering adding the ability for users to write component values if they please.

VI. SYSTEM IMPLEMENTATION

We can separate our implementation into three separate subsystems: the computer vision system, the circuit simulator system, and the mobile application. We can further separate the computer vision system into three separate workflows: parsing the user image, performing individual component detection, and generating netlist data structures.

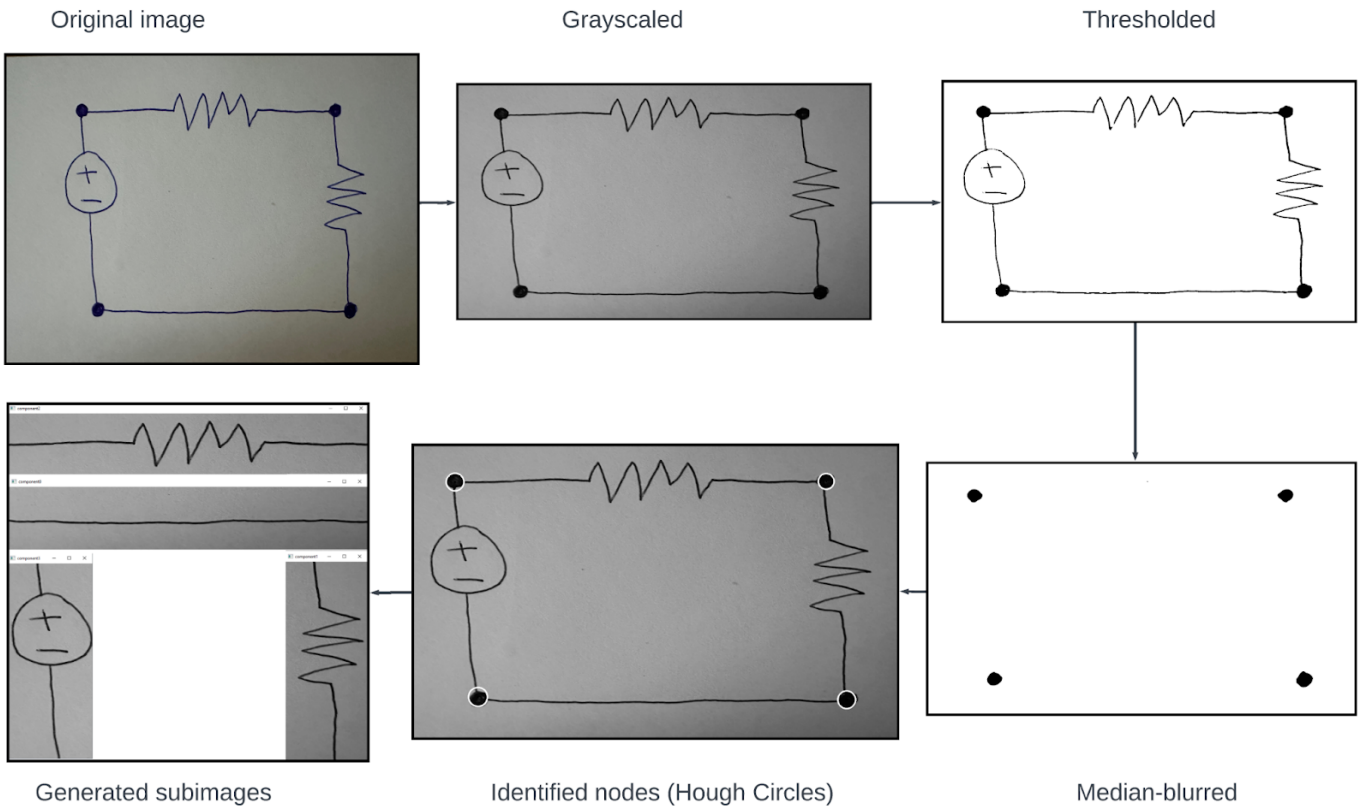

Fig. 3. Node Detection Pipeline

A. User Image Parsing

Before running feature detection algorithms on individual images of components, we must generate the subimages of the individual components from the user's original image. This process is shown in Fig. 3. Once the user uploads the image of their drawing, we load it as grayscale to perform basic thresholding. Thresholding is a simple method of image segmentation that will create binary images from a grayscale image. After thresholding, our binary image is entirely white except for the drawing itself, which is black.

Next, we apply a median blur to the binary image. Median blurring is most effective in removing noise from images. The first reason why we do this is because the blurring will remove lighter pen/pencil markers from the image that corresponds to the components themselves. This will allow us to isolate only the filled-in nodes that the user has drawn. Also, the blurring makes the nodes themselves more filled-in and distinct. Likely from the thresholding the less-filled in parts of the node will be removed, leaving a black circle with spots of white. The blurring will fill in this circle, which will remove all the possibly smaller, white circles that could be accidentally identified with Hough Circle Transform.

Now that we have an image with just black circles representing nodes, we use Hough Circle Transform to identify the location of each circle with x and y pixel coordinates. We know that a component must be between each pair of neighboring nodes, thus we have the coordinates that represent the far ends of the component, and we can use them to extract subimages of each component. The subimages are taken from the original grayscale image.

B. Individual Component Detection

Before doing feature detection, we first apply Bilateral blurring to the grayscale subimages. This will make the subimages more naturally blurred while preserving the general edges of the drawing. We then perform Canny edge detection, which will isolate the edges of the drawing and black-out everything else. For feature detection we use the ORB algorithm. This will generate FAST keypoints and BRIEF descriptors. The descriptors are generated from the keypoints as binary feature vectors that represent all the important features in the subimage.

Using these BRIEF descriptors we can perform brute-force matching with BRIEF descriptors we have stored in the application's data storage. These stored descriptors have associated component names with them, such as "resistor", or "voltage source". We perform the brute-force matching with Hamming distance to quantify the accuracy of each match of features; the lower the distance, the better the match. Taking an average of the distances across all matches, we now have a score we can use to quantify the similarity between a subimage and a dataset image. Doing this with each subimage and each set of BRIEF descriptors in storage, we can rank what the best component match is for each subimage. See Fig. 4 on the next page as an example of the Canny edge detection and classifying the best match for subimages.

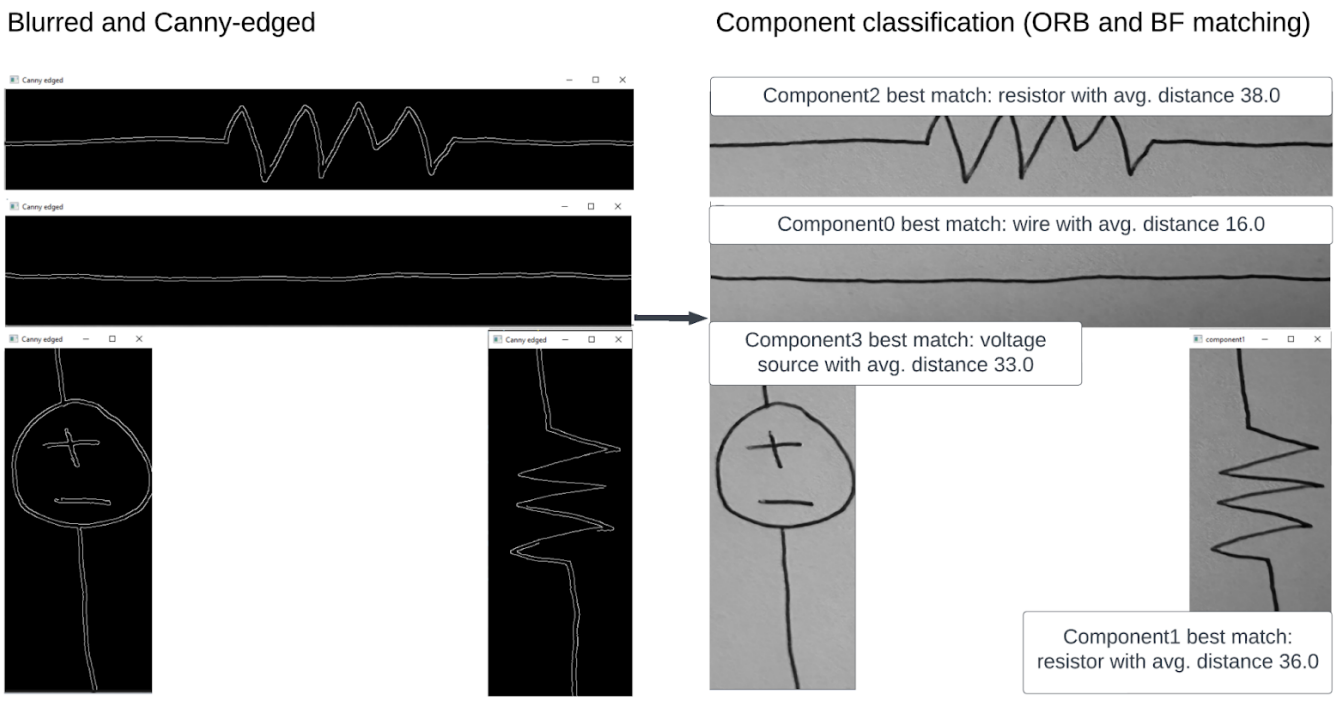

Fig. 4. Canny Edge Detection and Component Classification

C. Netlist Generation

From parsing the user image and detecting individual components, we now have identified nodes and the component types that represent edges between each node. From this, we want to generate a netlist to represent the circuit so that the frontend of the mobile application and the circuit simulator can easily reconstruct the circuit. A netlist is similar to a list of edges, which we already have, so this reconstruction is easy. We assign nodes an index and construct an edge that entails the component type and which nodes are the tail and the head. The only edge case is regarding wires. In a netlist, nodes that are only connected by wires are the same node. The fix for this is simple because we can just case on when the component between two nodes is a wire. An example circuit and corresponding netlist is shown in Fig. 5.

Because we want to generate the five most matching circuits, we must generate potential circuits and scores associated with each circuit. For each individual component, we consider the three best component types, and then generate every combination of these components. Because our maximum supported component limit is eight, the maximum number of combinations we generate would be $3^8 = 6561$ different circuits. One of the five most matching circuits that we output to the user will be the circuit with the best matching component type for each component. For each other circuit, we compute a score that factors in the scores for each individual component, as well as a likelihood score:

$$score = \sum(individual\ component\ distances) + likelihood\ score$$

The likelihood score is a simple metric that is intended to skew towards well-formed over malformed circuits. For example, a circuit generated that has no power source will have a larger (worse) likelihood score, as well as for a circuit with all power sources and no loads. Well-formed circuits will have smaller likelihood scores. The weight of the likelihood score will be determined through testing. Utilizing a weighted summation of the component distances where the number of matched features corresponds to the weight is also being considered and will be tested.

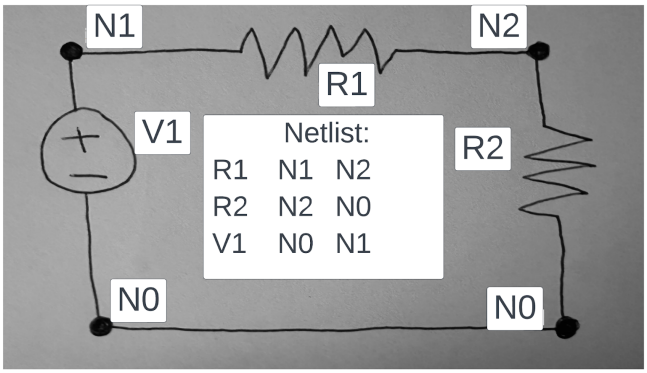

Fig. 5. Sample Circuit and Generated Netlist

D. Dataset

Our dataset is not a collection of images of individual components, but rather a text document where each line corresponds to an image's BRIEF descriptors and classification for the component. Because the BRIEF descriptors are what we care about from the image, we can immensely save on the application's file data. As mentioned previously, BRIEF descriptors are 32-byte binary string representations of features in an image.

For each component type (voltage and current sources, resistors, wires, bulbs, switches, and LEDs), we will have five sets of reference descriptors (five images for each component). Important to note is that to account for the polarity/orientation of voltage and current sources and LEDs, we will have separate classifications for their directions. This means our dataset will span the equivalent of 50 total images of components. Using an average of 9401 bytes per component from Table 2, the text file will be around 500 KB large.

We will use five reference images for each component type to account for variation in drawings and orientations of the components (horizontal and vertical). Because ORB can account for orientation, drawn components that aren't perfectly vertical or horizontal will still have similarly detected features.

E. Circuit Simulator

We will be using modified nodal analysis to solve and simulate circuits[2]. The simulator will take in a netlist as input and generate matrices that will be used to solve a system of equations describing the circuit. Modified nodal analysis tries to solve the equation

$$Ax = z$$

in which A describes the connections and conductance of the elements of the circuit, x describes the unknown values that we are trying to solve, and z describes the current and voltage sources. The A matrix is of size $(v+n)(v+n)$, where v is the number of voltage sources and n is the number of nodes. It is constructed of four smaller matrices

$$A = \begin{bmatrix} G & B \\ C & D \end{bmatrix}$$

The G matrix details the conductance of the circuit elements, the B and C matrices detail the connections of the voltage sources, and the D matrix will always be a zero matrix if the circuit only contains independent voltage sources. The simulator will generate the required matrices and solve the equation above for the x matrix, which contains the unknown voltages and currents. Once these are determined, they will be added to the netlist and sent to the frontend for rendering. Although modified nodal analysis generates a larger system of equations than other algorithms such as traditional nodal analysis or mesh analysis, modified nodal analysis is easier to implement algorithmically on a computer system.

F. Mobile Application

The mobile application will be written in Swift. The home screen consists of an application description and two tips to help the user upload their circuit correctly. From here there is a skip button that leads to an upload page where the user will upload their drawing. From there a wrapper must be created to bridge the C++ code with the Swift code. In doing so, the Swift codebase will have access to all the functions from the C++ code and will be able to run the computer vision code. However, the image that was sent through the app will have to be parsed and converted with Swift's "UIImage" class. When the computer vision algorithms finish, five netlists will be generated.

Based on these netlists, we will craft and display circuits. For this we will utilize reference images stored in the application's file data for each component and connect them with lines for proper wiring and circles for distinction of nodes. Each circuit will be displayed on a swipeable page. In addition to the five pages corresponding to each circuit option, there will be a sixth swipeable page that will allow the user to reupload their circuit if none of the five displayed circuits match the circuit the user intended to create.

Once the user selects their circuit, they will then be able to input all the values of each component drawn. This includes resistor values, voltage values for voltage sources, knee voltage for LEDs, etc. The user will be able to input either the raw numbers for values or shortened ones. For example, "10000" and "10k" will both be parsed as the quantity ten thousand. After the user is satisfied with the values they have inputted, they will be given an option to "Analyze" the circuit through a button directly below the display. Once the user has clicked "Analyze", the contents of the completed circuit will be sent to the circuit simulator via another Swift-C++ wrapper, and the data of the circuit analysis will be overlayed on top of the circuit diagram. At any point the user can refresh the app and go back to the home page to start the process over.

VII. Test, Verification and Validation

Each subsystem of the project will undergo testing all throughout development. These subsystems include individual component detection, full circuit detection, circuit simulation testing, and accessibility testing. We plan to have a test group of seven children between the ages of 12 and 14 to conduct many of our tests. These tests are designed to reinforce all our use case requirements as well as improve the overall quality of the entire system.

A. Unit Testing for Individual Component Testing

To test for individual components, we plan to have the test group each draw every component six times. These components include voltage and current sources, resistors, wires, bulbs, switches, and LEDs. Each of the component drawings will be rotated to test both the vertical and horizontal orientations. To ensure we hit our accuracy goals, we will also test with other varying factors, pictures with poor lighting and components drawn on lined paper. These tests will be conducted to determine the best diameter of each pixel neighborhood and sigma values for the bilateral blurring, as well as the upper and lower threshold limits for the Canny edge detection. Success is indicated by the best match for each component image being correct 90% of the time.

B. Unit Testing for Circuit Detection Accuracy

To test the circuit detection functionality, we plan to have the test group draw four different circuits each. In addition to this, each of the group members will draw ten circuits. These circuit images can be used for all the testing mentioned below.

The first circuit detection unit test is regarding the node detection. We take the raw images of each circuit and feed it into our preprocessing algorithms and see if nodes can be

correctly identified. This testing will be used to determine the best thresholding value, aperture linear size for median blurring, as well as the parameters for Hough circles (minimum distance between circles, min/max circle radius, and min/max thresholds for edge detection). We measure success by manually inspecting that each of the intended (and only the intended) nodes were identified.

The second unit test is for the creation of the component subimages. Given coordinates of detected nodes, we must determine what are appropriately sized bounding boxes to encompass a component in between each pair of neighboring nodes. Success is measured by manually inspecting the generated subimages, ensuring they encompass the main features of the component and don't include extra features, such as the nodes or other components.

The third unit test is constructing the five best circuits given a list of components and what edges they correspond to. Here we need to ensure that polarity of components is properly handled, and circuit scores are computed accurately. This test will also be used to tweak the likelihood score weight when calculating circuit scores. We compare the generated netlists from this test with what we expect as the means of measuring success.

The integration testing for this circuit detection subsystem requires connecting all three of these components. Because there is a gap in this subsystem where the individual component detection system is used, we can manually create the list of components from the subimage generation to isolate the integration testing of this subsystem.

C. Unit Testing for Full Circuit Detection

This testing is as simple as connecting the full circuit detection subsystem with the individual component detection subsystem. We can use the same images from the full circuit testing for this. Success here means that for 90% of the input images, one of the five output netlists corresponds to the circuit in the input image.

D. Circuit Simulator Testing

The plan to test the circuit simulator is to cross reference results with a different circuit simulator online, LTspice. We will write a script that generates SPICE netlists with components we support and input these into both simulators. The accuracy goal for this subsystem is 100%, thus we plan on the output of both simulators to be the same every time.

E. Circuit Simulator Testing

The test group will be given the phone application and will be given a series of tasks to complete. These will include navigation through the home page to the upload page, going back to previous pages, and uploading a picture. In addition to this, Once doing this, we plan to give each member a survey of questions to rate their experience with the UI/UX on a scale of 1-10. These questions will include:

- "How easy was it to upload your circuits?"
- "How clear were the schematics of all the circuits displayed?"
- "How easy was it to input values for your circuit?"
- "How useful were the tips on the home screen when drawing your circuit?"
- "How easy was it to recognize what you needed to redraw with your circuit if it wasn't an option?"

The result of the survey will help with future developments as it allows us to understand either what is missing from the app or what has been a success amongst the test group thus far. In addition to this, we will perform unit tests on the application to make sure all subsystems are working properly. Our first unit will be verifying that the image sent from the user is uploaded to the application correctly. The iOS application is planned to temporarily store the user's image; thus, we can manually check that the image is not altered and up to standards for the circuit simulator.

Secondly, we plan to feed netlists into the application to test the functionality of displaying circuits given a netlist. We plan to test fifteen for three sets of recommended circuits as well as another five for the final page. We plan to draw these circuits and create the netlists accordingly. Lastly, we are going to test the value input functionality. This will be done by inputting the values on the selected circuits and verifying the netlists being generated are correct. We will draw full circuits, with values, as well as the netlists that correspond and will cross reference with the output of the iOS application.

F. Circuit Simulator Testing

The first test would be the image upload functionality. To do this we plan on uploading circuits through the application and verifying that the computer vision algorithm both receives and reads the image. We will measure success manually by the output of the algorithm. The second test would be the netlist parsing to create circuit UI. Once a user uploads a circuit five netlists are created that the iOS application must parse and display. This will also be manually tested by cross referencing the netlists and the circuits displayed on the application. Finally, we must test that the values created from the circuit simulators are parsed correctly and displayed in the right location on the application interface. We will cross reference with the values that are resulted from the circuit simulator to make sure the values are placed accordingly to the diagram.

VIII. Project Management

A. Schedule

Fig. 6 is the Gantt Chart with the schedule of work as well as each team member's weekly tasks.

B. Team Member Responsibilities

Each member has taken on one of the main three subsystems of the project.

- Stephen Dai: Computer Vision
- Devan Grover: Circuit Simulator
- Jaden D'Abreo: iPhone Application

All the members have worked together to design the architecture of the system; however, Jaden will be mostly in charge of testing and integrating towards the end of the project.

C. Bill of Materials and Budget

All necessary materials will be items the user must have themselves. These materials consist of an iPhone with iOS 8.0+, a writing utensil, and paper.

D. Risk Mitigation Plans

If we are struggling to reach our 90% individual component detection accuracy, a simple risk mitigation is just increasing the size of the dataset. Even if we doubled our original dataset size, it would still be only around 1 MB of data. Increasing the size of the dataset provides more options to find stronger matches between images, which can never hurt the component detection accuracy assuming that the additions to the dataset are properly classified. A related idea is that we can tailor each user's application to their individual drawing abilities by going through a form of calibration. If a user draws an individual component and classifies it, we can use it in the dataset. If we were using a neural network, this would be susceptible to adversarial machine learning because users could purposely add improper classifications to the dataset. Because the dataset is local to the user with our application, they would only be affecting their own accuracy. For this, we would need to add the option to recalibrate and remove the previous calibration's data in case the user accidentally misclassified components. This idea will likely be implemented post-MVP unless we struggle reaching the 90% detection mark.

In the case where we struggle to have one of the five displayed circuits be correct, a simple boost can be to increase the number of displayed circuits. Another more critical issue that affects the circuit detection is if nodes can't be identified properly. This issue can be handled in testing by experimenting with different values used in the image preprocessing algorithms, as well as changing the type of preprocessing algorithm (ex: different blurring).

If the circuit simulator we create is unable to work, we can use an existing tool like SPICE. The newest version of SPICE uses C, which we can package into our iOS application. SPICE is an open-source circuit simulator that uses the same netlist format that we are planning on using in our own circuit simulator. Therefore, if our circuit simulator does not work, we only must change the actual simulation logic to use SPICE - no change will have to be made to the computer vision algorithm or frontend to account for the new simulation library.

IX. Related Work

There are numerous online circuit simulators such as LTSpice. But all these simulators require building the circuit in the application itself and are not free of cost to users. Regarding the computer vision aspect, there exists research that was conducted to contrast the use of SIFT and ORB with drawn electrical components[7], but these components were digitally and not hand drawn, and they only classified individual components and not entire circuits. Research has been conducted to reconstruct full circuits from drawn circuits[6], but this uses digitally drawn circuits by providing an online GUI, and it also uses a CNN (convolutional neural network). Neither of these tools are available to the public.

X. Summary

The Circuit Simulpaper system is designed to allow a younger audience to learn basic circuit functionality through an iPhone application coupled with drawing. It serves as an educational tool that can be accessible for all. The system requires three items, items that most households already contain, to produce a desired circuit analysis. Foreseeable challenges include circuit detection accuracy, integration between the phone application and the computer vision aspect, and user testing. We are confident that through our thorough design and integration plans that we will be able to overcome all challenges and provide a high-quality educational tool.

Glossary of Acronyms

BRIEF - Binary Robust Independent Elementary Features
FAST - Features from Accelerated Segment Test
LED - Light emitting diode
ORB - Oriented FAST and Rotated BRIEF
SIFT - Scale-Invariant Feature Transform

References

[1] Anderson, M. (2018, May 31). *Teens, Social Media and Technology 2018*. Pew Research Center: Internet, Science & Tech. https://www.pewresearch.org/internet/2018/05/31/teens-social-media technology-2018/

[2] Cheever, E. (n.d.). *Analysis of Circuits*. Analysis of circuits. https://lpsa.swarthmore.edu/Systems/Electrical/mna/MNA1.html

[3] Heideman, P. D., Flores, K. A., Sevier, L. M., & Trouton, K. E. (2017). *Effectiveness and adoption of a drawing-to-learn study tool for recall and problem solving: Minute sketches with folded lists*. CBE life sciences education. https://www.ncbi.nlm.nih.gov/pmc/articles/ PMC5459246/

[4] Mejía, D. (2023, April 13). Four out of five households with children owned tablets. Census.gov. https://www.census.gov/library/stories/ 2023/04/tablets-more-common-in-households-with-children.html#:~ :text=The%20share%20jumped%20to%2075,17%20years%20old%20% E2%80%94%20owned%20tablets

[5] Miller, C. (2023, March 13). *When should you get your kid a phone?*. Child Mind Institute. https://childmind.org/article/when-should-you-get-your-kid-a-phone/

[6] Keerthi Priya, A., Gaganashree, N., Hemalatha, K. N., Chembeti, J. S., Kavitha, T. (2022). AI-based online hand drawn engineering symbol classification and recognition. Lecture Notes in Networks and Systems, 195–204. https://doi.org/10.1007/978-981-16-8512-5_22

[7] Pavithra, S., Shreyashwini, N. K., Bhavana, H. S., Nikhitha, G., & Kavitha, T. (2023). Hand-drawn electronic component recognition using Orb. Procedia Computer Science, 218, 504–513. https://doi.org/10.1016/j.procs.2023.01.032

[8] *Swift*. A powerful open language that lets everyone build amazing apps. Apple. (n.d.). https://www.apple.com/in/swift/

[9] Haiyan Wang,Tianhong Pan, efei,Anhui and Jiangsu.(2020) "Hand-drawn electronic component recognition using deep learning algorithm"China, Int J:Computer Application in Technology

[10] Dewangan, A. and A. Dhole,(2018) "KNN based hand drawn electrical circuit recognition" International Journal for Research in Applied Science & Engineering Technology:p. 1111-1115.

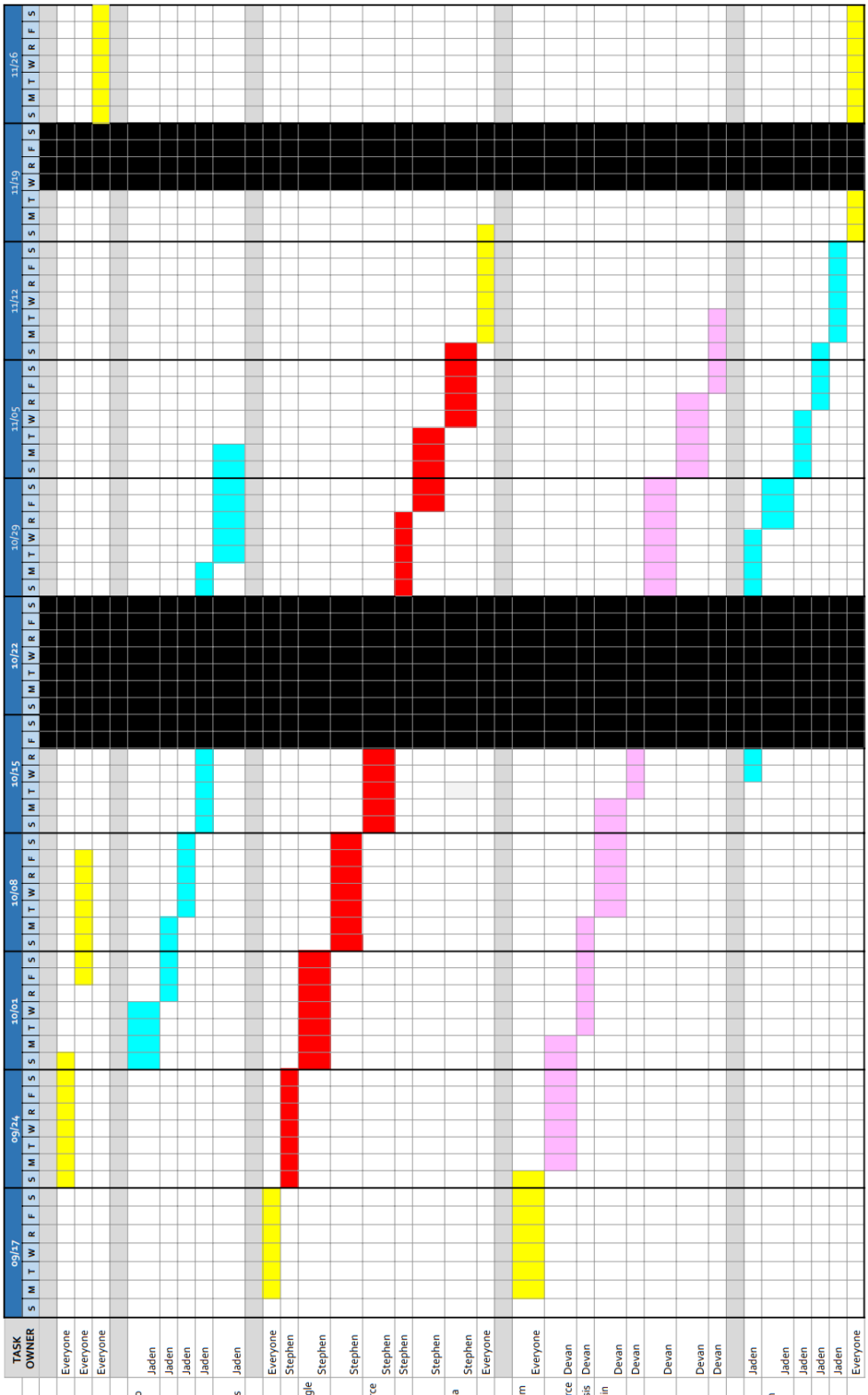

Fig. 6. Gantt Chart Schedule