AUTO-COUNT SYMBOLS IN PORTABLE DOCUMENT FORMAT (PDF)

by

Andrew Florian

April, 2021

Project Advisor: Mark Hills, PhD

Major Department: Computer Science

Estimating electrical costs often involves counting symbols in a PDF document. Existing software has sped up this process compared to manual counting, but there is room for further improvement. The proposed solution builds on open source components to efficiently search a PDF document for the outlines of all symbols, including letters or numbers, used by electrical engineers to differentiate between otherwise similar symbols. It then sorts these outlines into groups and counts each occurrence. Symbol for symbol, it takes less than half the time required by two leading competitors. Unfortunately, current settings often produce numerous sub-groups which need to be combined to provide meaningful totals. K-means and other improved clustering methods are being explored. The proposed concept could also be helpful in other similar applications that identify symbols or text in images.

AUTO-COUNT SYMBOLS IN PORTABLE DOCUMENT FORMAT (PDF)

A Master's Level Project

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

Andrew Florian

April, 2021

Copyright Andrew Florian, 2021

Table of Contents

LIST OF TABLES iv

LIST OF FIGURES v

1 INTRODUCTION 1

2 COMPETITION 3

3 RESEARCH 5

4 DESIGN 7

4.1 Development Environment 7

4.2 Render Original PDF 8

4.3 Outline Symbols 9

4.4 Organize Similar Symbols 11

4.5 Count Each Type And Display Results 14

5 TESTING 15

6 SCHEDULE 21

7 CONCLUSIONS AND FUTURE WORK 24

BIBLIOGRAPHY 27

LIST OF TABLES

5.1 Comparison: Typical Drawing (439 kb size, with 77 outlet symbols) 17

5.2 Comparison: Scan (2,135 kb size, with 77 outlet symbols) 18

5.3 Comparison: Benchmark (859 kb size, with 1,000 outlet symbols only) 19

LIST OF FIGURES

4.1 Scanned Image of Outlet Symbol (enlarged) 9

4.2 Vector Image of Outlet Symbol (also enlarged) 9

4.3 Outlines of Above Images (applies to both scan and vector) 10

4.4 Segments "Jogging" in 2 or 3 Directions 11

4.5 Quadrants & "Center of Gravity" 13

6.1 Schedule 22

Chapter 1

Introduction

My plan for this final project was to build a practical tool expanding on what I have learned in previous ECU software projects. In SENG 6285 - Cloud Computing (spring 2019 with Dr. Tabrizi) my team set up a simple Python application on Amazon Web Services to dynamically update a web page with information from a small MySQL database. We built several web applications using Flask in CSCI 6710 - Web Applications Development (spring 2019 with Dr.Wu). In CSCI 6905 Topics in Computer Science (summer 2019 with Dr. Hills) we created Android apps to display photos and location data on a map.

My group used Django to demonstrate a concept for an online library system in SENG 6230 - Software Engineering Foundations (fall 2019 with Dr. Tabrizi). In CSCI 6040 - Computational Analysis of Natural Languages (spring 2020 with Dr. Gudivada) we built a Python model from 1.5 GB of Gutenburg texts that used probability to generate similar (often humorous) text. I designed a Java estimating web application in SENG 6245 - Software Construction (fall 2020 with Dr. Azizi). Although JavaScript was not the primary language for any of these, most did require a JavaScript component.

After working with several platforms and in a variety of languages, I have seen the power of JavaScript. It is convenient to write code that can be easily used on a wide range of devices using various browsers. If an application implemented in Python or Java has a web component, it can waste valuable time and energy packaging and transferring information to and from a JavaScript layer. In some cases, the tasks could more efficiently be processed directly in JavaScript without involving the other languages.

Here, the intent is to quickly and accurately provide counts of components for estimating an electrical project. Ideally, the counts would be added as comments to the legend area of the input PDF (which could also be reduced in size if symbol recognition eliminates the need for similar information to be included multiple times in the same file) and easily exported to a .csv spreadsheet if needed.

There are some existing programs such as Bluebeam Revu [14] or OnCenter OST [10] which for several years now have allowed a user to draw a box around a symbol that the user would like to find more of. See chapter 2 for comparisons of competing systems. It would be helpful to have a system that automatically counts all symbols without a user needing to draw boxes around anything. Given the similarity of some symbols, rotations, overlaps, and lack of resolution on some drawings, it is not expected to be perfect, so it should also include an easy way for an estimator to delete/change false positives or add ones that are missed. As discussed in chapter 7, this and other features will likely be added in the future.

ReportLinker calculated the global market for construction estimating software (of which counting software is only a part) in 2019 at $1.04 billion and expects it to grow by over 10% annually to $1.88 billion by 2025. [1]. Checking back a year later, they have revised – increasing these same predictions by a factor of 10 (it may be interesting for the reader to check current predictions). For this project, a more complete auto-count system is expected to be easily marketable, but pricing and marketing have not been considered at this point.

In the future, it could be extended to include other features of this global construction estimating software market by linking a specific symbol such as ⊖(outlet) or $(switch) to a database of parts and labor hours associated with the conduit, wire, box, mounting screws, wire nuts, device, cover plate, etc. that would be implied by that symbol. It could develop into a complete estimate generating system that would require less oversight and manual work by the human estimator than current “systems” require.

Chapter 2

Competition

From personal experience working as an electrical estimator for 10 years and conferring with other estimators, Bluebeam Revu [14] and OnCenter OST [10] stand out as the leading competition for symbol counting. Each has auto-count features along with some limitations. Both require the user to drag a box around a symbol that is to be counted. Then the user must wait until results are displayed with options for deselecting symbols that may have been improperly included or adjusting sensitivity settings to perform the search (and waiting period) again.

Bluebeam adds count markers directly to the input PDF. This makes it easy to share counts with suppliers or subcontractors. Bluebeam's process for splitting and merging counts seems more cumbersome than the corresponding processes in OnCenter.

OnCenter has some convenient features. For example, missed symbols can be easily added to a group by clicking one item in the group, pressing the space bar, then clicking on each item to be added. Unfortunately, OnCenter operates more slowly on PDF files. It works better with .tiff files and does have a built-in conversion utility. This (and the way counts are stored in a proprietary .ost file) makes it cumbersome to share counts.

In spite of these limitations, these features were impressive when they were initially rolled out (and often save time compared to manual counting), but there is definitely room for improvement. If the user has many different symbols to count, their "auto-count" routine becomes a time-consuming process. There is opportunity in the market for software that is able to automatically and accurately count all symbols in a set of construction drawings without user involvement (dragging a box around each symbol or experimenting with settings). The proposed solution aims to count symbols quickly and with minimal user involvement.

Chapter 3

Research

Many symbol-spotting systems turn to convolutional neural networks trained on large sets of data [2, 11]. Some use complex template matching systems [8]. This project seeks a simpler solution not dependent on vast amounts of data and not even requiring an internet connection.

PDF is a common format for distributing construction drawings since they can generally be relied on to produce a consistent image when printed or displayed on a wide variety of devices. PDF files can contain a lot of information such as when, how, and by whom they were created or modified – which may be important in some situations.

In other cases, the only thing that is needed is an accurate representation of the final image. Unless an estimator has been included in all the versions throughout the design process, he or she likely is not interested in whether the PDF was created using vectors, contains hidden layers with subsequently superseded information, or was scanned from a paper copy – as long as the top visible layer is clear. Extra information can be a distraction and can unnecessarily increase file size.

Preliminary research uncovered open source code that could be used as a starting point. Traversy Media demonstrated how to make a JavaScript application into a desktop application using Node-Webkit [13]. This was not hard to set up, but the resulting desktop application relies on cumbersome supporting .dll files (see chapter 5 for more information).

Mozilla has developed PDFjs [9] as a Portable Document Format (PDF) viewer with a reliable method for rendering a PDF document as an image. Another option for converting PDF files to images is Khishigbaatar's pdf-poppler NPM package [5]. As discussed in section 4.2, some of Mozilla's code was used in this project.

Peter Selinger developed Potrace [12] to efficiently convert a bitmap image into a vector outline to avoid pixelation when enlarging. Selinger's code was later ported to JavaScript by Kilobtye [6]. This project implements a modification to Kilobtye's code as discussed in section 4.3.

John Lindquist demonstrated how Scalable Vector Graphics (SVG) can be easily converted into a PDF using PDFKit [7]. Since the browser already has a default method for saving as a PDF, this was not explored. In the future, it might be worth looking at in more detail.

In 2016, Im and others used a variation of k-means clustering [4] for some dot patterns in Visual Research. A similar clustering method could be used to group like symbols in the proposed solution.

Chapter 4

Design

4.1 Development Environment

This project was developed on an HP ENVY Desktop 750-427c PC with Intel(R) Core(TM) i7-6700 CPU @ 3.4GHz processor, 16 GB ram, a NVIDIA GeForce GTX 750Ti (2 GB) graphics card, and Windows 10 Pro x64 10.0.19042 operating system. The latest version of VS Code (1.53.2) was installed with the Live Server 5.6.1 plugin. Chrome (88.0.4324.190) was the primary browser used for debugging and accessing Overleaf (to develop this document). The solution is implemented as a JavaScript (version 1.7) web application with a minimal HTML framework and some CSS. The latest code (including previous iterations and a copy of Mozilla's pdf.js version 2.8.2 build 6249ef517 and associated pdf.sandbox.js and pdf.worker.js) is stored in GitLab [3].

For maximum speed, it can be run locally (not needing an internet connection). It includes an NW.js package file, so it can also be built into a desktop application for various operating systems via Node-Webkit [13]. The counting process could have arguably been performed faster without a display component in Python, but visualization seemed worthwhile mainly for verifying results. JavaScript was used exclusively to avoid inefficiencies discussed in chapter 1.

The solution can be broken down into sections for rendering, outlining, organizing, and counting. See section 4.2 below for details related to rendering the original PDF. The process for outlining each symbol (simplification of work by Selinger [12] and Kilobtye [6]) is then covered in section 4.3. Section 4.4 represents the majority of the effort in organizing symbols for similarity regardless of position or orientation. The final section 4.5 addresses counting and displaying totals for each unique symbol.

4.2 Render Original PDF

PDF files can be extremely complex. Drawing producers employ a large variety of methods. Some vastly differing methods can produce remarkably similar final images. Exploring all the ins and outs of the PDF format is well beyond the scope of this project. A way to normalize the input (ignoring metadata, hidden layers, and irrelevant information) was needed.

Since Mozilla's viewer.js [9] already displays PDF files well, an early idea was to modify it to analyze what it was displaying. Such modifications proved more difficult than expected. The main issue is that Mozilla's viewer has been optimized to only display the amount of detail needed at the current zoom level and only for the portion of the file that is currently in the viewing window. This organizational structure is very effective for viewing, but not for analyzing every detail of an entire document.

Another possible "ready-made" approach is an NPM package called pdf-poppler [5] for converting a PDF to an image. This would have worked well with a desktop-only deployment produced by Node-Webkit [13], but could not easily be used in a web application. Instead, a simple viewing system was developed. It includes simple HTML with an event listener to receive and process any PDF files dragged onto it. Using Mozilla's PDF.js (version 2.8.2, build 6249ef517 without Mozilla's associated viewer) it gets the entire document, then renders a high resolution PNG image of each page (by calling *pdfjsLib.getDocument()*, *.getPage()*, and *.render()*).

Figure 4.1: Scanned Image of Outlet Symbol (enlarged)

4.3 Outline Symbols

Outlining is a way to define where one symbol ends and where another starts. This differentiation doesn't work when symbols touch each other, but in those cases some features inside the symbol will likely still be recognizable. Figure 4.1 shows an enlarged view of a typical electrical outlet symbol from a scanned PDF image. For comparison, figure 4.2 shows a similar symbol from a vector PDF file. Such differences from one file to another would make template recognition difficult. Increasing magnification beyond a certain point does not produce any additional pertinent information and only amplifies the differences.

What is actually important is only the outline of each shape. Potrace [12] is very effective at finding outlines that more smoothly represent a shape when enlarged. For this project, smooth curves were not the goal – just accurate outlines. Figure 4.3 shows in red the overall outline of a typical outlet symbol as well as in black the smaller outlines inside it which were all produced by Potrace [12] (using Potrace's optional setting: $alphamax = 0$ to create polygons with only straight segments instead of more complicated curves produced by the default value of $alphamax = 1$)

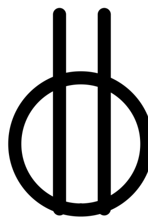

Figure 4.2: Vector Image of Outlet Symbol (also enlarged)

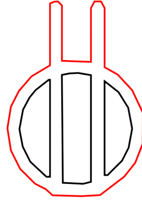

Figure 4.3: Outlines of Above Images (applies to both scan and vector)

Initially, the Javascript port of Potrace by Kilobtye [6] was used without modification to process the image of each page. It first converts any colored or gray pixels to either black or white based on a darkness threshold, then produces an SVG file with a path representing each outline of each independent black area in the image. Later a simpler and faster method (than even the $alphamax = 0$ option mentioned above) was developed for picking optimal points to represent a path. The original port of Potrace by Kilobtye [6] calculated a value called "*lon*" for each point along the current un-simplified path. This "*lon*" basically contained a pointer to the farthest coordinates along the same path that could be reached via 1 segment containing pixel jogs in up to 3 directions.

For reference, figure 4.4 (from page 5 of the Potrace documentation [12]) shows segments (a) and (d) which start to the right, then jog in only 2 directions – up and right (maybe multiple times, but no left or down jogs). Segments (b) and (c) again start to the right yet jog in 3 directions – up, right, and down (again no left jogs). Segment (e) starts up and jogs in 3 directions – right, down, left (even though it started up, there are no upward jogs). After establishing "*lon*" as described above, Kilobtye's port of Potrace then shortened or "clipped" most segments to avoid "strange behavior around the corners" per page 6 of the documentation [12], determined the minimum number of segments required to simplify that path, and compared all possible polygons having that number of segments to find the best fit.

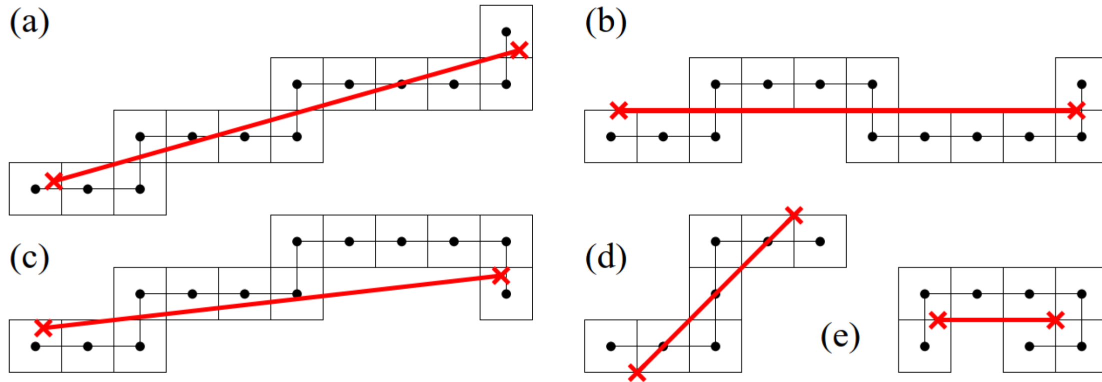

Figure 4.4: Segments “Jogging” in 2 or 3 Directions

The simpler method presented in this report instead modifies “*lon*” to contain a pointer to the farthest coordinates along the same path that could be reached via 1 segment containing pixel jogs in up to 2 (instead of 3) directions. Notice how segments (b), (c), and (e) from figure 4.4 jog in more than 2 directions and would thus be shortened. Then one polygon (possibly not the absolute best fit, but probably close enough) can be found with adjustment at the occasional corner instead of having to “clip” most segments – and without having to spend the time testing for the best fit.

4.4 Organize Similar Symbols

One simple attempt at organization was to sort the symbols by size then iterate through the sorted list of symbols comparing each point in the current symbol with that of only the previous symbol. If the square of the differences was less than an arbitrary threshold, then they were considered to match. This was fast and worked well in a lot of cases, but unfortunately many symbols are similar sizes and the sort would intersperse them with other symbols which would require a more lengthy comparison to multiple previous symbols (not merely the one immediately previous). Another issue was matching differing rotations of the same symbol.

This prompted the idea to divide each symbol into four quadrants marked by the first instances of maximum and minimum X and Y coordinate values encountered along the outline path of that particular symbol. Symbols were then rotated in 90-degree increments to have a consistent orientation. First, the width of each symbol was compared to its height. If width was greater, it would remain un-rotated (0 degrees) unless its center of gravity (discussed later) indicated it should be rotated 180 degrees. If height was greater, it would be rotated either 90 degrees or 270 degrees depending again on its center of gravity.

The following formula: $\frac{\text{(mean of X values)} - \frac{\text{(min X+max X)}}{2}}{\text{(max X}-\text{min X)}}$ was used to calculate the "center of gravity" in the X dimension. A similar calculation was performed in the Y dimension. Of these, whichever dimension had the greater absolute value was given dominance. Its sign (negative or positive) then governed the decisions mentioned above (0 vs. 180 degrees) or (90 vs. 270 degrees).

Figure 4.5 shows an example. The first quadrant is shown in green starting at the first instance that the minimum Y value is found when traversing that path in a clockwise direction. In this case, it happens to contain six points that approximate the original pixel path. The next quadrant is shown in red starting at the first instance of the minimum X value and happens to contain seven points. The first instance of the maximum Y value marks the beginning of the purple quadrant containing eight points. The last quadrant has sixteen points and is shown in orange starting with the first maximum X value.

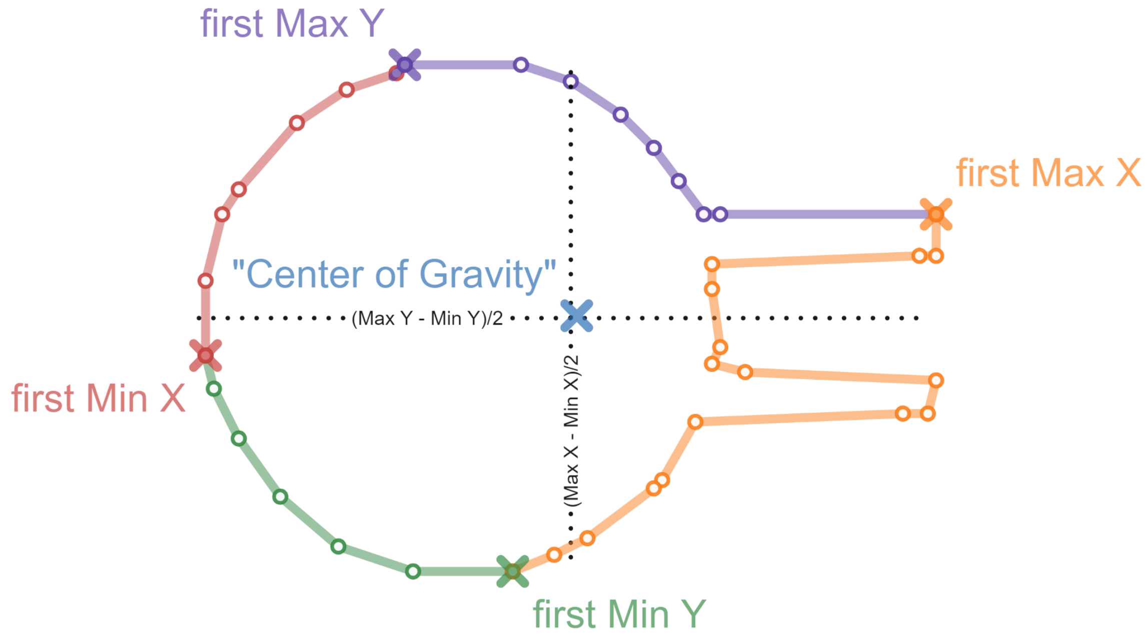

Figure 4.5: Quadrants & "Center of Gravity"

This method of dividing the outline into quarters ensures that symbols that are rotated can be easily compared using defined starting points for each quadrant. The symbol in 4.4 happens to be wider than it is tall, so the next choice is whether to rotate it 180 degrees or leave it un-rotated. Although the "center of gravity" happens to be very close to midway between the max and min values, it is slightly off – and more so in the X dimension than in the Y dimension. Since that larger value happens to be positive, this symbol would not need to be rotated. If the value was negative, it would be rotated 180 degrees. A symbol that was taller than it was wide would be rotated either 90 or 270 degrees depending on its "center of gravity". In this way, every symbol can be pre-rotated by the appropriate 90-degree increment before making comparisons. If other angles are expected to be frequently encountered, it might be worth extending this concept to smaller rotation increments (such as 45 degrees).

Then a 5-dimensional sort organizes the outlines by size (the area the symbol covers rounded to the nearest pixel) as well as the number of coordinates in each of its quadrants. Differences of 1 or less are ignored so that similar outlines end up getting grouped together in the sorted list. A clustering algorithm would likely do a better job of grouping symbols. As discussed in chapter 7, this possibility will be explored more fully in the future.

4.5 Count Each Type And Display Results

Excluding the first outline, all but the last two sets of relative coordinates for each outline are compared to those of the previous outline. If the sum of the differences does not exceed an arbitrary threshold of 30, the symbol is considered a match, and a counter is incremented. Any time the threshold is exceeded, the previous symbol is added to a table along with the current count, the counter is reset, and the program starts comparing the coordinates of the next outline.

The table is displayed in a sidebar with a row for each unique symbol and three columns. One contains an SVG path (scaled to fit in the height of the row). Another column has the quantity for that symbol. The remaining column contains a text box to add a description if desired. For each instance where that symbol was found in the original PDF file, a reference to that symbol's one SVG path is used along with information for translating it and rotating it to the correct position and angle on the page.

Chapter 5

Testing

Manual testing was performed throughout the development process. The first test confirmed that a desktop application could be built from a JavaScript web app. Mozilla's viewer (4 MB in size) was packaged with NW-builder [13] as an example. The resulting .exe file (with several supporting files totaling over 325 MB in size for the win64 platform) opened without errors on the Windows 10 PC. It viewed PDF files and appeared to function the same or slightly slower than it did with Live Server in a browser.

Most subsequent testing was performed simply with Live Server to save time – with the idea that the code could be built into a desktop application at any time if desired. It was later discovered that there can be differences between a browser application and the resulting desktop build. The zooming function for example needs additional code for the desktop application. Also, packages need to be added in different ways (import vs. require). As discussed in section 4.2, modifying Mozilla's viewer to fit this project proved to be more difficult than creating a simple viewer from scratch.

After creating a simpler viewer, the next test was a visual comparison. SVG outlines were placed in a semi-transparent layer in front of the PNG rendering of each original PDF. This made it easy to ensure that everything lined up where it was supposed to. Testing followed the addition of each component or feature to ensure that all simplifications, translations, or rotations functioned as intended and did not "break" previously working parts.

It was easy to see when symbols were correctly sorted by size. It was also obvious if a "similar" symbol was used in place of another which did not exactly match. After the quadrant system was implemented, it was helpful to display a single quadrant at a time to ensure that all symbols were treated the same through all rotation angles. Whenever a problem was detected, it was helpful to create a small sample PDF file with symbols of a certain shape or orientation to test that specific condition.

The metrics used in final testing were accuracy, speed, and compression. Accuracy is measured as the percentage of correctly included symbols (a higher percentage is preferable) and incorrectly included symbols (a lower percentage is preferable) in each count. Speed is recorded as time in seconds and seconds per symbol (lower is preferable for both). Compression is calculated as $\frac{\text{original file size}}{\text{processed file size}}$. (value of 1 indicates no change to file size, a higher value is preferable).

Comparisons were performed against BlueBeam Revu Version 2017.0.40 (hereafter referred to as BB) and On Center OST Version 3.97.2.2 (hereafter referred to as OC) processing 3 files. The first file (439 kb in size) is one sheet of a typical electrical drawing. It is 34 inches wide and 22 inches high as produced by the US Army Corps of Engineers in 2016, and happens to contain 77 standard outlet symbols. For simplicity, time on the following tables is rounded to the nearest second and does not include the initial program loading time (about 6 seconds each for BB and OC) nor the time for a user to carefully drag a rectangle over one of the outlet symbols to be counted – neither of which is needed for the Auto-Count solution presented in this report (hereafter referred to as AC).

	Auto-Count	**BlueBeam**	**OnCenter**
Total Quantity of Shapes	3,595	77	105
% of Outlets in Largest Grouping	17	100	100
Percentage in Wrong Category	0	0	36
Time in Seconds	42	8	9
Seconds per Symbol	0.012	0.104	0.086
Compression	0.65	0.93	0.85

Table 5.1: Comparison: Typical Drawing (439 kb size, with 77 outlet symbols)

Results for the first file are shown in table 5.1. *BB* happened to find all 77 outlets in eight seconds and did not incorrectly count any of the other symbols as outlets. It added 35 kb to the file size (compression factor of 0.93). *OC* happened to find all 77 outlets in nine seconds, but incorrectly also included 28 symbols of a different type (or 36% in the wrong category). It added 75 kb to the file size (compression of 0.85). *AC* used 42 seconds, but in that time it processed all 3,595 shapes on the drawing including four outlines for each of the 77 outlets per figure 4.3). *AC* attempted to group all these shapes by similarity – the largest group of outlet outlines only including thirteen shapes (or 17% of the 77 outlets).

Chapter 7 includes ideas for improvement, since an estimator would have to add the total of this largest group and several smaller groups to find the total number of outlets. *AC* included so many groups that it did not have the problem of incorrectly including any different symbols in any of these groups. An efficient saving method has not yet been developed for *AC*, but using the browser's default print to PDF setting, 232 kb was added to the file size (compression of 0.65). This actually includes both an image of the original PDF as well as vector paths for all the outlines. Presumably, it could be saved with just the vector paths to save space. It might be worth experimenting with saving in different formats such as .svg or even .html as long as it could still be reliably rendered on different devices.

	Auto-Count	**BlueBeam**	**OnCenter**
Total Quantity of Shapes	3,564	71	53
% of Outlets in Largest Grouping	27	73	43
Percentage in Wrong Category	0	19	26
Time in Seconds	15	18	5
Seconds per Symbol	0.004	0.254	0.094
Compression	3.08	0.95	0.98

Table 5.2: Comparison: Scan (2,135 kb size, with 77 outlet symbols)

The second file is a 2,135 kb scanned image of the first file with results in table 5.2. *BB* and *OC* perform worse than they did on the first file, while *AC* performs better on this scanned image than it did on the first file. *BB* took eighteen seconds to find 56 outlets (or 73% of the 77 outlets). It incorrectly included fifteen symbols of a different type (or 19% in the wrong category). It added 107 kb to the file size (compression of 0.95). *OC* took five seconds, but only found 33 outlets (or 43% of the 77 outlets). It incorrectly included twenty symbols of a different type (or 26% in the wrong category). It added 27 kb to the file size (compression of 0.98).

AC took fifteen seconds to process all 3,564 shapes (it identified 31 fewer independent outlines than in the original file). This time, its largest group of outlet outlines included 21 shapes (or 27% of the 77 outlets). Again, an estimator would have to add the totals of several groups to find the total number of outlets. Here, even though it could likely be saved in a more efficient way, the browser's default print to PDF setting reduced the file size by 1,442 kb (or a compression of 3.08).

	Auto-Count	**BlueBeam**	**OnCenter**
Total Quantity of Shapes	4,000	779	505
% of Outlets in Largest Grouping	100	78	51
Percentage in Wrong Category	0	0	0
Time in Seconds	3	6	11
Seconds per Symbol	0.001	0.008	0.022
Compression	1.36	0.45	0.69

Table 5.3: Comparison: Benchmark (859 kb size, with 1,000 outlet symbols only)

For the third file, an 8.5 by 11 inch page with exactly 1,000 outlet symbols (250 facing each of four directions – up, down, left, and right) was created with no other symbols. This ensures that each of the three systems process the same number of symbols. Results are shown in table 5.3. Here all systems performed better than on the scanned file, and there were no other different types of symbols to incorrectly include. *BB* took six seconds to find 779 outlets (or 78% of the 1,000 outlets). It added 1,055 kb to the file size (compression of 0.45). *OC* took eleven seconds, but only found 505 outlets (or 51% of the 1,000 outlets). It added 377 kb to the file size (compression of 0.69).

AC took three seconds to process all 4,000 shapes (four outlines for each of the 1,000 outlets per figure 4.3). The largest group of outlet outlines included all 2,000 of the “D” shaped inner outlines (two for each outlet or 100% of the 1,000 outlets). The file size was reduced by 229 kb (or a compression of 1.36).

The results bring to light a few interesting points. For some reason, *BB* is much slower on the second file (a scanned image of the first file). *BB* and *OC* both happen to lose accuracy on the second file compared to their results on the first file, while *AC* performs better on the second than on the first. *BB* and *OC* add to the file size in all three cases, while *AC* is able to reduce the file size in two out of three cases.

For the first file, although AC increased the file size by more than the competitors, it also processed many more symbols. To be fair, we could look at kb per symbol processed. Measuring this way, BB used 0.5 kb/shape (35 kb divided by 77 shapes) and OC used 0.7 kb/shape (75 kb divided by 105 shapes). Even if we assume that each shape has four outlines, AC only used 0.3 kb/shape (232 kb divided by 3,595 outlines times 4 outlines per shape)

Overall, these tests show that the solution presented in this report isolates unconnected symbols at a much faster rate than either of the competing systems. In the three test cases, it did not wrongly categorize any symbols. It resulted in more subgroups than desired, which seems to indicate that it is too sensitive to minor differences. It also shows that file size can be reduced.

Chapter 6

Schedule

This project was completed during the ECU spring 2021 semester starting January 19, 2021. Figure 6.1 shows the major activities and phases through completion the end of April, including an introduction with the initial plan (chapter 1), a brief overview of leading competitors (chapter 2), research (chapter 3), design (chapter 4 including the development environment in 4.1 and each of the key functions – rendering 4.2, outlining 4.3, organizing 4.4, and counting 4.5), testing (chapter 5), scheduling in this chapter, and conclusions (chapter 7).

As with most projects the actual schedule differs from what was originally proposed. At first, the plan was to learn about portable document format (PDF) with the hope of making direct modifications to input files. Due to their reference table and compression methods, it soon became evident that PDF files cannot be edited as easily as some other formats. Whenever objects are added to or removed from a PDF, any reference to the starting point of a subsequent object must be adjusted to point to its new location.

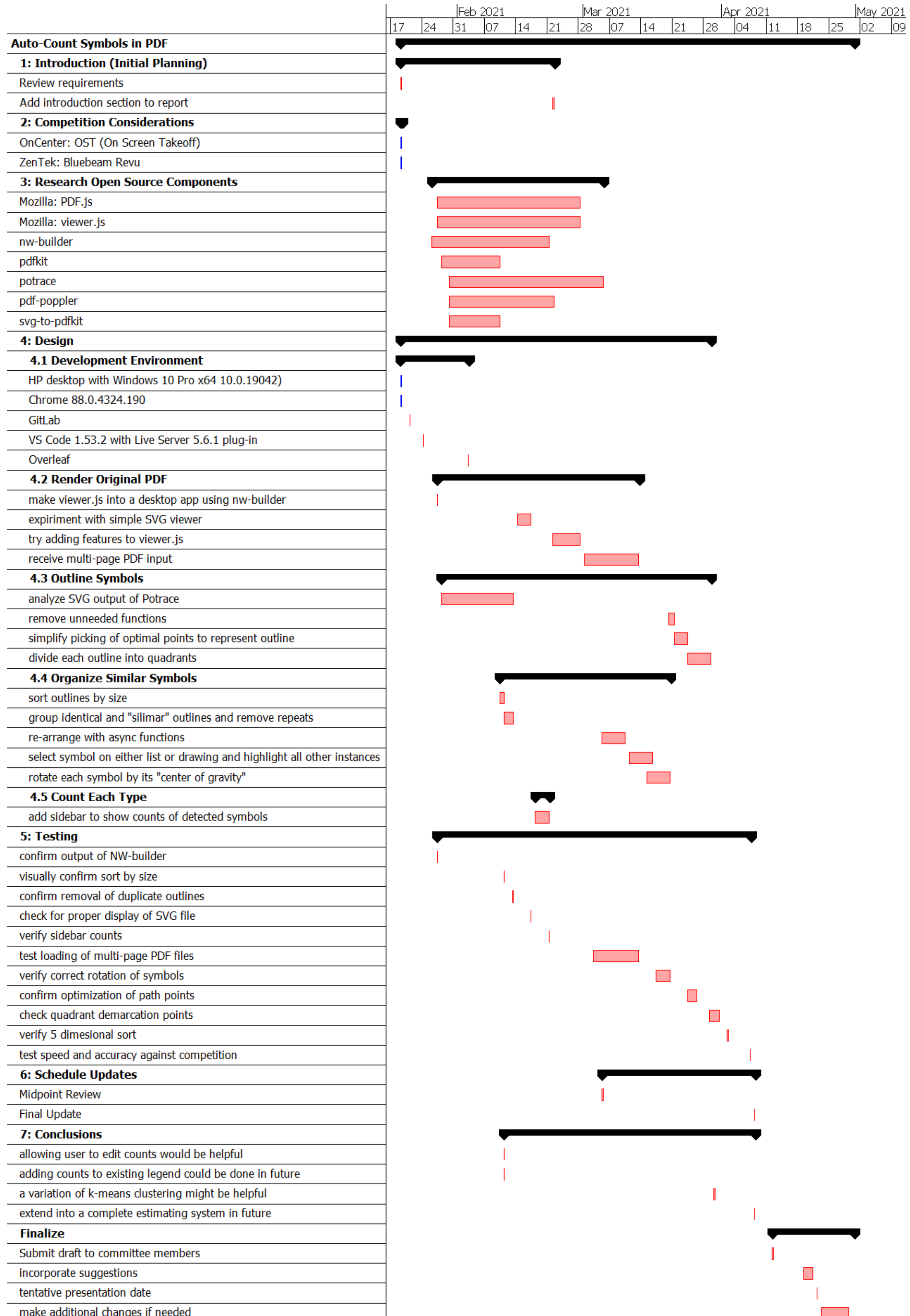

Figure 6.1: Schedule

A task that took more time to develop than anticipated was the rotation of symbols in a consistent way for comparison. At first, it was thought to be a relatively easy matter of merely rotating entire symbols. First, the value midway between its maximum and minimum X and Y values was subtracted from each coordinate to find its relative coordinates centered around the point (0,0). Then for a 90-degree rotation, new relative coordinates of $(Y, -X)$ were used in place of the original relative (X, Y) coordinates. Alternatively, new relative coordinates of $(-X, -Y)$ produced 180-degree rotation, and new relative coordinates of $(-Y, X)$ resulted in 270-degree rotation. So far this was simple as expected.

The problem was that the first point for tracing the outline of each symbol originally started at the top. Rotating the entire symbol made some paths start at the right side, some at the bottom, and some at the left side of the symbol depending on the amount of rotation. The first coordinate of a path starting at the top of a symbol would not match with the first coordinate of a path starting at the right side of an identical symbol. The quadrant system described in section 4.4 took more time than expected. Fortunately, it was able to overcome this problem and allow rotated symbols to be compared point for point – each from a consistent starting location.

Chapter 7

Conclusions and Future Work

The proposed system shows promise. It is useful in its present stage of development, but there is room for future improvements. Even estimators currently using competitive auto-count software could likely save time. Instead of dragging a box over one symbol then waiting, dragging a box over another symbol etc., the estimator could have all symbols quickly identified at one time. At this point, he or she would instead need to add up quantities from several sub-groupings to find each desired total.

Given the effort that Mozilla has expended developing a method for rendering PDFs, that aspect of the solution is solid. An independent copy of Mozilla's current release is included in the code for this project. Although it seems adequate at this point, it is possible that some types of PDF files may not render accurately with their current system. If so, it is expected that they will make improvements in their future versions. At that time, this solution will still continue to function as-is, but could then be updated to include their revisions.

The original tracing algorithms developed by Selinger [12] and ported to JavaScript by Kilobtye [6] has been used for many years without updates and are not expected to require maintenance. The modified outlining algorithm used in this solution appears to function correctly on the shapes that it has been tested on thus far. It is possible that some shapes may produce irregular results. It also appears that the occasional quadrant is missing a few necessary coordinates for a complete and accurate representation of the original shape. This issue is being tested further.

In place of the sorting method implemented, a clustering algorithm would likely be a better way of finding matching symbols. Traditional k-means clustering requires the desired number of groups as an input. Since the number of different types of symbols for any given PDF is unknown, some adjustments would be needed. In Visual Research, Im and others describe a "window-based" modification of k-means clustering [4] which seems promising.

Hierarchical clustering is another option. This would be helpful in cases where shapes touch or overlap, effectively joining their outer outlines. It would not be difficult to note which if any inner outlines are contained within each outline. The quantity, relative position, and size of any inner outline found within an outline could also be used as input for the clustering algorithm.

Another consideration is a modification of the arbitrary square of differences threshold for comparing symbols. It might be helpful to vary the threshold in proportion to either size and/or the number of coordinates being compared. This would make a lower threshold when comparing smaller, simpler shapes and a higher threshold when comparing larger, complex shapes.

Bluebeam Revu [14] and OnCenter OST [10] each offer a slider to adjust sensitivity. The proposed solution aims to minimize such user involvement. Ideally, the system would be tuned to operate truly automatically without user involvement. If the desired grouping cannot be achieved automatically, it might be worth adding sliders or controls to allow the user to adjust settings.

Even with the best grouping method, it would be helpful if a user could easily edit the groupings. In cases where different symbols represent effectively the same installation scope, the user should have a way to merge these groups. In other cases, a bid alternate might require otherwise identical symbols to be counted separately. Accordingly, the user should have a way to split a group.

Finally, it is important to see where this system fits into the overall estimating process. What is the best way to use the counts or convey them to the next phase of estimating? In order to easily send light fixture quantities to suppliers for pricing, the counts could be added as comments to the legend of the input PDF file. Counts could also be exported as a .csv file. A long-term goal would be to extend this into a full estimating system. This would require setting up a database with units for parts and labor hours to be associated with each symbol. In many cases, it would need to allow additional user interaction to “interpret” the intent of the drawings.

Bibliography

[1] 360iResearch. Construction estimating software market research report by product, by end-user, by deployment - global forecast to 2025 - cumulative impact of covid-19, Dec 2020. https://www.reportlinker.com/p05913945/Construction-Estimating-Software-Market-Research-Report-by-Product-by-End-User-by-Deployment-Global-Forecast-to-Cumulative-Impact-of-COVID-19.html.

[2] Borrmann, A., and Stoitchkov, D. Analysis of methods for automated symbol recognition in technical drawings. In *TUM Civil Engineering Bachelorthesis for Technical University of Munich* (2018). https://www.semanticscholar.org/paper/Analysis-of-Methods-for-Automated-Symbol-in-Borrmann-Stoitchkov/1e8986d8d9c8c6d5e5fb1724d2414c84b70c105c.

[3] Florian, A. Auto-count symbols in pdf. https://gitlab.cs.ecu.edu/floriana18/auto-count-symbols-in-pdf.

[4] Im, H. Y., hua Zhong, S., and Halberda, J. Grouping by proximity and the visual impression of approximate number in random dot arrays. *Vision Research 126* (2016), 291–307. Quantitative Approaches in Gestalt Perception http://dx.doi.org/10.1016/j.visres.2015.08.013.

[5] Khishigbaatar. Npm: pdf-poppler. https://www.npmjs.com/package/pdf-poppler.

[6] kilobtye. A javascript port of potrace. https://github.com/kilobtye/potrace.

[7] Lindquist, J. Convert svg to a pdf in node with pdfkit and svg.js. https://egghead.io/lessons/express-convert-svg-to-a-pdf-in-node-with-pdfkit-and-svg-js.

[8] Moreno-García, C. F., Elyan, E., and Jayne, C. New trends on digitisation of complex engineering drawings. *Neural Computing and Applications 31*, 6 (Jun 2019), 1695–1712. https://doi.org/10.1007/s00521-018-3583-1.

[9] Mozilla. Pdf.js. https://github.com/mozilla/pdf.js.

[10] OnCenter. Auto count. https://www.oncenter.com/university/video/auto-count.

[11] Rezvanifar, A., Cote, M., and Albu, A. B. Symbol spotting on digital architectural floor plans using a deep learning-based framework. https://arxiv.org/pdf/2006.00684v1.pdf.

[12] Selinger, P. Potrace: a polygon-based tracing algorithm. http://potrace.sourceforge.net/potrace.pdf.

[13] TraversyMedia. Create desktop apps with web technologies - nw.js. https://www.youtube.com/watch?v=5UsGnjPYxLU.

[14] ZenTek. Did you know series: Visual symbol search and count. https://youtu.be/aujjymvmg9Y.