EPiC Series in Computing

Volume 64, 2019, Pages 165–173

Proceedings of 28th International Conference
on Software Engineering and Data Engineering

Utilizing Optical Character Recognition and Border Detection Algorithms to Identify Trading Cards

Brodie Boldt, Christopher Cooper, Ryan Fox, Jared Parks and Erin Keith

University of Nevada, Reno, Reno, NV, US,
Department of Computer Science and Engineering
bboldt@nevada.unr.edu, christophercooper@nevada.unr.edu,
ryanfox@nevada.unr.edu, japarks@nevada.unr.edu,
erinkeith.unr@gmail.com

Abstract

Magic: The Gathering is a popular physical trading card game played by millions of people around the world. To keep track of their cards, players typically store them in some sort of physical protective case, which can become cumbersome to sort through as the number of cards can reach up to the thousands. By utilizing and improving optical character recognition software, the TCG Digitizer allows users to efficiently store their entire inventory of Magic: The Gathering trading cards in a digital database. With an emphasis on quick and accurate scanning, the final product provides an intuitive digital solution for storing Magic: The Gathering cards for both collectors and card owners who want to easily store their collection of cards on a computer.

1 Introduction

Optical Character Recognition software (OCR) is a powerful technology that has many practical applications. This technology was utilized as the main feature in order to create TCG Digitizer, allowing it to efficiently scan physical images of Magic: The Gathering trading cards [1] and translate them into a digital entry in a localized database. Magic: The Gathering is an incredibly popular physical trading card game, or TCG for short, that is played around the world by millions of players. Thousands of unique cards have been made since its original creation in 1993, which has created the dilemma of

F. Harris, S. Dascalu, S. Sharma and R. Wu (eds.), SEDE 2019 (EPiC Series in Computing, vol. 64), pp. 165–173

sorting through collections with hundreds of copies of cards. For shop owners that sell these cards, this amount can go into the tens of thousands, which makes physical cataloging tedious and prone to error.

There were several features that needed to be accomplished to make the project successful:

- The ability to use a simple 1080p camera to rapidly take pictures of cards for scanning
- Use an Optical Character Recognition software to obtain data from the scanned picture
- Compare the captured text from the picture with a background database to find a match
- Store each card entry from the user in a database for fast and easy access

The OCR technology that was chosen for use in the software was Tesseract 3.05 [2], which is sponsored by Google LLC. This was combined with blob detection functionality from AForge.NET [3] in order to trace an outline around the cards for easier reading. After an image of a card has been taken, it is split into different rectangular shapes that are drawn around two specific areas of the card: the name and the description. Once these shapes are drawn, Tesseract goes in an ordered-step process to try to identify the card. First, it will scan over the name of the card and compare it to the backend database to check for a match. If this is not sufficient, it will search for a match with the card's description. If both checks are not enough, the card will be tagged as having a possible error and the user is notified to make the appropriate corrections.

The background database system used for handling card storage was PostgreSQL [4]. PostgreSQL was used to store every card entry that the user scanned into their inventory and kept track of different users in the system with varying privilege levels, ranging from admin to guest.

2 Constraints

Making sure all of these technologies functioned properly was not an easy task. There were several issues with card scanning and correctly parsing the data to ensure that it could be properly added to the inventory. The biggest issues identified were:

- Cards had to be perfectly straight when a picture was taken to ensure correct image cropping
- Tesseract was highly inaccurate and produced correct transcriptions only 29% of the time
- The non-standard font used by card was hard for Tesseract to discern between certain characters
- Colors of the cards could interfere with reading
- ASCII control characters that were present in strings were improperly handled by Tesseract.

These problems resulted in a lower than desired success rate and needed to be addressed in order for the software to be viable. Section 2 discusses these obstacles in more detail and how they impacted the overall effectiveness of the application and Section 3 contains the solutions to these challenges.

2.1 Cards placed at an angle

A misaligned card could produce inaccurate results due to blurriness or from angled text. While it was possible to force the user to manually align the card perfectly within a box, this was not considered ideal. A card holder would have also solved this problem, but it could also damage the card if

mishandled. The only solution that minimized the risk of damaging cards during a scan was simply placing the card on a flat, contrasting surface and placing the camera above it. To solve the issue of odd angles, an algorithm was developed to detect them at these angles, which is discussed further in section 3.2.

2.2 Errors from using Tesseract-OCR

Even when cards were aligned correctly in the frame and the text was clear in the image, errors were still possible. Google Tesseract 3.05 was chosen as the software's OCR technology due to its ease of use and availability as a NuGet package in Windows Forms, which the application was built in. Tesseract tends to see characters in backgrounds that are not perfectly uniform and in the case of Magic: The Gathering cards, the backgrounds of text spaces have inconsistent patterns. Figure 1 is an example of a typical card name.

Figure 1: Example of text background on some Magic cards

Inconsistencies such as those in the image above would cause Tesseract to return bad strings that, when tested against the database of card names, resulted in wasted time correcting the mistakes.

2.3 Non-distinct text characters

Cards in Magic: The Gathering use a custom font made by Wizards of the Coast [5] that is relatively easy for people to read. Optical Character Recognition, however, has some difficulty correctly reading this font.

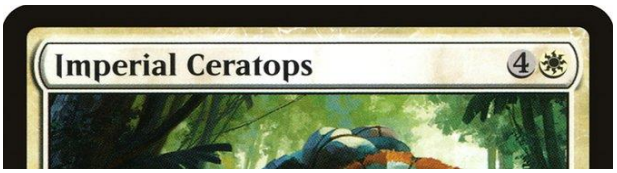

Figure 2: Example of a Magic: The Gathering card name

Figure 2 is a prime example of this problem. The characters 'I' and 'l' are difficult to distinguish from one-another. Additionally, 't', while clearly discernible from the other two, looks like the character 'l' to Tesseract. During accuracy analysis, it was found that cards that didn't contain these characters were successfully transcribed upwards of 71% of the time, while cards that did contain them had a success rate of approximately 26%. Addressing this issue became a high priority.

2.4 Variable card color

The background of a card in Magic: The Gathering can come in many different colors. This led to issues where cards could be more difficult to read depending on their background color. In particular, darker backgrounds usually resulted in lower accuracy than lighter backgrounds. This made it difficult to properly recognize any image of a card that was not filtered or modified. The seven card colors that were tested during development were red, green, blue, black, white, gray, and gold.

3 Methods

Various methods were used to increase the overall accuracy of Tesseract and the software's ability to overcome incorrect transcriptions. These methods can be simplified into four categories: image filtering, image transformations, text manipulation, and database spelling forgiveness.

3.1 Color correction

The initial testing of Tesseract's accuracy when transcribing text from an image, was done using various image filtering techniques similar to how other sources have implemented image detection [6]. The first attempt used a custom contrast and grayscale filter that altered the pixels of the image one-by-one by averaging the RGB values. The contrast filter was achieved by increasing an RGB average if it was above a median threshold and decreasing it if it was below that median. This method did not increase accuracy dramatically, which hovered around 29% at the time.

The second color correction attempt used a binary filter called Bradley Local Thresholding, which used Gaussian filter techniques to remove noise, similar to solutions from other sources [7] [8] and an image inverting filter provided by the AForge API [9]. It was found that the accuracy increased dramatically when Tesseract was detecting white text with a black background and had only two colors to process. To reduce blurring and a loss of contrast, the image was converted to grayscale first and then converted to black and white. Using the Bradley Local Thresholding filter without converting to grayscale resulted in pixel noise around high contrast areas, including text.

Tesseract functioned properly with the image in this state, but tests showed that inverting the image so that the text was white on a black background resulted in better accuracy. The effects of these filters can be seen below in Figure 3, where the final picture is the result after all three filters. Following rigorous testing and improvements, the final build processed the image using all three filters and increased accuracy to approximately 50%. All color types previously mentioned in section 2.4 were successfully converted to grayscale with no issues. It should be noted that this method was not tested on holographic cards.

Figure 3: Diagram of the filters used while processing an image. A color image is converted using grayscale, Bradley local thresholding, and inverting filters to produce the image on the bottom right.

3.2 Edge / Blob detection / Quadrilateral Transformation

To avoid the issue of the camera improperly focusing at certain distances, a card detection algorithm was developed rather than statically cropping the image that was taken. This also allowed the card to be placed anywhere so long as it was in view of the camera as well as at angles that were not perpendicular to the camera.

For the software to correctly detect a card, it must be completely within the camera frame and there must be a contrast between the card edges and the background surface it is resting on. When a picture is taken, it is duplicated, converted to black and white and inverted. All algorithms are currently analyzing the black and white version of the image and adjusting both images to provide the color version. Initial card detection uses the AForge blob detection algorithm [10] with its blob size range limited to being greater than 25% and smaller than 97% of the original image's width and height. The algorithm returns a list of rectangles (rectangles consist of x, y, width, height information relative to an image) which meet those requirements and the largest rectangle will be used for further card edge detection.

Card edge detection was achieved with a custom algorithm that finds the card's border, calculates the formula for the line of that edge, and finds the x, y coordinates for each corner. This algorithm was contained within a class to allow for different referenceable coordinate types (AForge IntPoints and System.Drawing PointF) and detection failure flags. These flags were triggered if the corner angles were out of an acceptable range of 90° ± 10°. The algorithm starts at the edge of the rectangle found during blob detection and walks along pixel by pixel until it detects a white pixel. A visual representation of the algorithm's operations can be seen in Figure 4.

Figure 4: Corner x, y coordinates are calculated using edge detection algorithm.

The x, y coordinates are stored, and another pass is made at a different position looking for that same edge. In cases where the blob detection is off, and the first pixel is detected as white the algorithm will start 10 pixels outside of the rectangle and work its way inward as long as it is still within the dimensions of the original image. The default number of passes per edge is three, but the class allow for the number of passes to be specified during class construction to allow for additional accuracy if desired. After multiple x, y edge coordinates are found for each edge, the algorithm finds the slope (m)

between each neighboring point pairs using equation (3.2.1), averages the results of each pair and excludes any dramatic outliers.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{3.2.1}$$

Outliers can occur if a pass exceeds the card's dimensions or if an edge detection pass detects another edge by being passed the card's corner while the card is askew in the image. Using the slope of each line and one of each edge's x, y coordinates each y-intercept (b) is calculated using equation (3.2.2).

$$b = y_n - x_n m \tag{3.2.2}$$

Using the slope and y-intercept of each corner's intersecting lines, the x, y coordinates of that intersection is derived from equations (3.2.3) and (3.2.4) respectively and rounded to the nearest integer.

$$x_{corner} = \frac{b_h - b_v}{m_v - m_h} \tag{3.2.3}$$

$$y_{corner} = m_h x_{corner} + b_h \tag{3.2.4}$$

To adhere to the previously mentioned corner angle range, and to confirm the detected object is a fairly rectangular structure, each corner angle (θ_{hv}) is calculated with equation (3.2.5) and a failure flag is set if the angle is out of the acceptable range.

$$\theta_{hv} = tan^{-1}(|\frac{m_h - m_v}{1 + (m_h m_v)}|) * \frac{180}{\pi} \tag{3.2.5}$$

If the class succeeds in finding the four corner points the program passes those points and the previous blob rectangle to the Quadrilateral Transformation filter provided by AForge [9]. The Quadrilateral Transformation filter takes four points from an image and will stretch the area within those points to fit the specified image size. This filter then transforms an askew non-rectangular card image into fixed, square, and level image. This allows for easy text image cropping based on a fixed ratio relative to the image's size. An example of this transformation can be seen in Figure 5.

Figure 5: Image within virtual corners is stretched to fit image dimensions.

A cropped image of the name header and, if additional accuracy is needed, the main descriptive text also is taken to be transcribed by the Tesseract OCR. All text images are taken from the images that have undergone black and white transformations and not the original color image.

3.3 Font Training / String Manipulation / Fuzzy Searches

During the early development process, seemingly correctly transcribed text would return nothing from the database when queried. It was discovered that due to a flaw in the Tesseract library characters like "\t", "\n" and nonstandard Unicode characters were being returned instead of ignored. The problem was so widespread that at least 25% of all text transcriptions contained at least one incorrect placed ASCII control character. Punctuation also became an issue as any dot or smudge on the image was interpreted as a period or comma. Due to most of the name header strings containing minimal punctuation, all punctuation, special characters, control characters and duplicate spaces were removed from all strings returned from Tesseract. The side effect of this was the permanent misspelling of any and all name headers containing punctuation.

Attempts were made to train tesseract to increase accuracy using the font type on the cards. This font type contained certain characters that looked very similar. For example, 'i', 'j', 'l', and 't'. These characters were commonly swapped with one another during transcription, resulting in text that was slightly misspelled. This became problematic when searching a database where a misspelled search query resulted in null data from the database, even if the string was off by only a single character. Tesseract's text training features were not sufficient to resolve these issues. As a result, another solution was developed to correctly identify cards regardless of the limitations of the OCR transcription's accuracy. With the accuracy increased due to card edge detection and image correction filters, the PostgreSQL database was made more robust by allowing search functionality to handle misspelled words and missing punctuation.

Manual optimization was achieved using the pg_trgm module for Postgres [11], which determined the similarity of two individual strings. Using the GiST operator class, they were able to make searchable indexes based on the header name, main text, and flavor text. These indices would be searched using the similarity function and the, potentially misspelled, Tesseract transcription. The queries response would be a list of entries sorted in order of highest probability of a match. If the probability of the first entry exceeded 70% then the returned name header would be accepted, and the card would be added to the queue. With only one "fuzzy" search, the acceptance rate increased to approximately 63%. The highest probability match on the cards that did not exceed 70% were correct in many cases, but a secondary test was implemented to ensure a higher rate of accuracy.

The card images that didn't find an acceptable match were processed to retrieve the image of the main text and the resulting Tesseract transcription from it. An additional similarity query was made that determined the probability that the misspelled name and main body text belonged to a particular card entry. Due to the excessive length and various special characters present in the main text, the probability threshold was decreased to 50%. This was acceptable due to the uniqueness and size of the main text. The most likely choice after this search generally results in the correct identification of the card. The resulting accuracy after all filters, edge detection, and "fuzzy" Postgres queries was approximately 97%.

3.4 Analysis of accuracy improvements

Analysis of each individual technique on their own and an analysis of all techniques combined showed the software's ability to correctly identify a card had increased from an initial accuracy rate of 29% when Tesseract was first implemented to approximately 97% when it used image filtering, card detection algorithms, and database query manipulation.

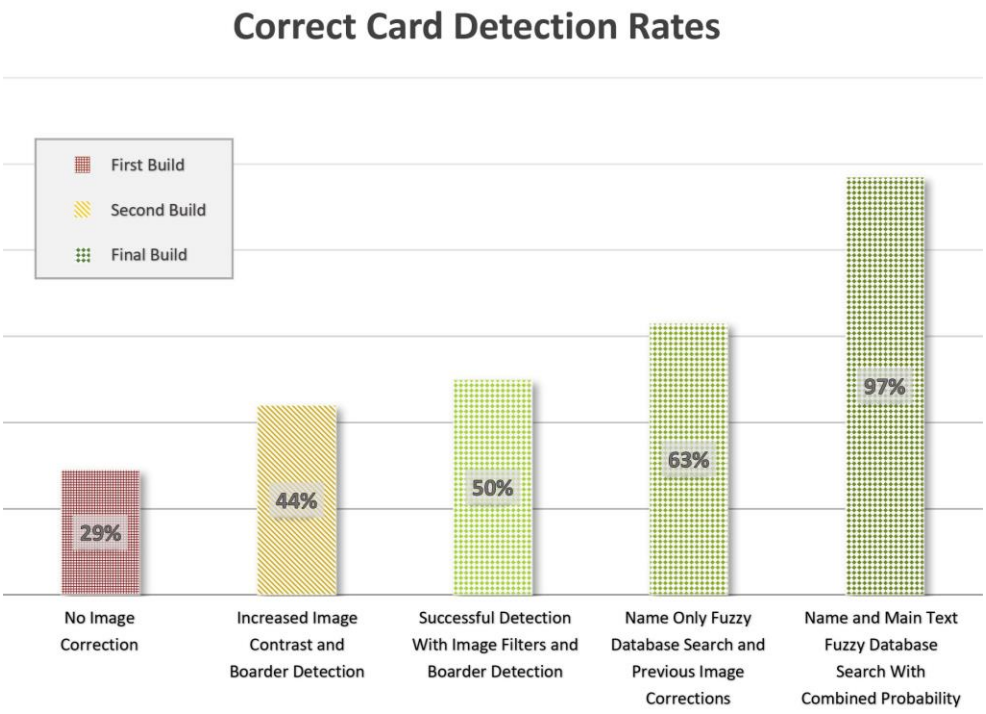

Figure 6: Various techniques gradually increased the software's detection accuracy to its current rate of approximately 97% successful detection.

As demonstrated in Figure 6, each iteration of development gradually improved performance. The final iteration where Fuzzy Database searching was applied to both the name and the main text of the card and probabilities were combined showed the biggest leap of improvement from the previous version. While 100% success would have been the best outcome, it was determined that 97% is more than acceptable given the nature of the software. In the event that a card fails to properly scan, the user can re-scan the card or make a manual correction to the entry before finalizing the card input.

4 Conclusions & Future Work

Through analysis, the accuracy data showed that the Name and Main Text "Fuzzy" Postgres searches and combined probability produced the largest jump in success rates and would likely, on its own, increase the consistency of the software, while the other techniques increased the efficiency and reduced the input needed from the user. While the accuracy of the image recognition had indeed increased, it came at the cost of increased processing time for each image. Further optimizations to the image recognition could increase the accuracy even more but would ultimately increase the overall processing time. Support for multithreading was planned as a solution to this for future development as it would help decrease overall processing time.

Multithreading would be implemented in such a way that after running the filters and border detection a single thread could process the header text, and another could process the main text. While earlier configurations had a lower accuracy rating, they could still be useful as they had faster processing times. If a thread running the earlier non-fuzzy search configuration is successful in identifying the name of the card it can return a success signal before the other threads finish, reducing process time.

The version of Tesseract that was used for TCG Digitizer was also not the most recent version. The TCG Digitizer used Tesseract version 3.05, while the most recent stable build is version 4.0. Upgrading to the newest version would provide faster processing time and increased accuracy with the use of LSTM neural networks [12]. A 100% success rate for each card would be the ideal scenario to achieve. Doing so would require additional improvements to the software design by using multiple threads and updating to the latest version of Tesseract.

References

[1] Wizards of the Coast, "Magic: The Gathering," Wizards of the Coast, 1993. [Online]. Available: www.magic.wizards.com. [Accessed August 2019].

[2] R. Smith, "Github, Tesseract-Ocr," Google LLC, [Online]. Available: https://github.com/tesseract-ocr/tesseract. [Accessed May 2019].

[3] A. Kirillov, "AForge.NET :: Computer Vision, Artificial Intelligence, Robotics," AForge.NET, [Online]. Available: http://www.aforgenet.com/. [Accessed May 2019].

[4] The PostgreSQL Global Development Groupd, "PostgreSQL: The world's most advanced open source database," [Online]. Available: https://www.postgresql.org/. [Accessed August 2019].

[5] Wizards Corporate, "Wizards of the Coast," Wizards Corporate, [Online]. Available: company.wizards.com/. [Accessed August 2019].

[6] Minh Nguyen, Wai Yeap, Steffan Hooper, "Design of a New Trading Card for Table-top," *2016 International Conference on Image and Vision Computing New Zealand (IVCNZ),* pp. 4-6, 2016.

[7] A. M. Davis, C. Arunvinodh and A. Menon NP, "Automatic License Plate Detection using vertical Edge Detection Method," *2015 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS),* pp. 2-4, 2015.

[8] Dr. T. K. Rao, K. Y. Chowdary, I. K. Chowdary, K. P. Kumar and C. Ramesh, "Optical Character recognition from Printed Text Images," *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, vol. 5, no. 2, pp. 597-604, 2006.

[9] A. Kirillov, "AForge.Imaging.Filters Namespace, Imaging Filters Classes," AForge.NET, [Online]. Available: http://www.aforgenet.com/framework/docs/html/cdf93487-0659-e371-fed9-3b216efb6954.htm. [Accessed May 2019].

[10] A. Kirillov, "AForge.Imaging Namespace, Imaging Classes," AForge.NET, [Online]. Available: http://www.aforgenet.com/framework/docs/html/d087503e-77da-dc47-0e33-788275035a90.htm. [Accessed August 2019].

[11] The PostgreSQL Global Development Group, "PostgreSQL 11.5 Documentation," The PostgreSQL Global Development Group, 1996-2019. [Online]. Available: https://www.postgresql.org/files/documentation/pdf/11/postgresql-11-US.pdf. [Accessed August 2019].

[12] R. Smith, "4.0 Accuracy and Performance · tesseract-ocr/tesseract Wiki · GitHub," Google LLC, [Online]. Available: https://github.com/tesseract-ocr/tesseract/wiki/4.0-Accuracy-and-Performance. [Accessed August 2019].