Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Automatic Reconstruction of Images from Shredded Paper

Author:
Jack Chorley

Supervisor:
Dr. Mark Wheelhouse

Second Marker:
Prof. Sophia Drossopoulou

June 17, 2019

Abstract

Shredding and tearing private material is seen as a secure way of disposing of information. Confidential documents, banking slips, passports and credit cards are all generally destroyed by cutting them into pieces with the assumption that this process cannot be reversed. This assumption of security and privacy relies on the premise that the difficulty and / or time for a human to reassemble the original image outweighs the benefit of having said document. If a computer program were able to take these pieces and reconstruct the original item within a reasonable amount of time with a high accuracy, it could make the process of shredding futile This has never been tested and evaluated in the public domain, but has been experimented with by governments. Here we attempt to model this scenario, build an image reconstruction pipeline, and then evaluate the feasibility and accuracy to determine the risk that software of this kind has on current prevention techniques. The model will be generic enough to handle real-life scenarios, such as missing a few pieces or extra pieces being mixed in, irregular edges such as those from ripping apart paper rather than straight-line cutting, and low resolution pieces that may be cropped from photos. This opens up the platform for applications in criminal investigations to recover evidence, personal restoration of damaged photos and documents, or perhaps reassembling an old treasure map.

Acknowledgements

I'd like to thank my supervisor Dr. Mark Wheelhouse for always making time in his already busy schedule to provide invaluable guidance, fresh ideas and enthusiasm at every opportunity; and Prof. Sophia Drossopoulou for her feedback and great advice on writing academic papers.

I would also like to thank Dr. Tony Field for 3 excellent years of tutorial support, beer, and for attempting to mask his look of fear when I told him I hadn't started writing my dissertation yet.

Finally, I'd like to thank my family for their unrelenting support throughout all of my education, providing me the opportunity to follow my dreams; and my closest friends for the BBQs, distractions and "stimulating intellectual conversation".

Contents

List of Figures

Chapter 1

Introduction

Image reconstruction is well researched and explored, but previous studies tend to target how to rebuild an image when you have noisy or missing data. Often the focus is on maximising information gain in order to estimate what the missing pieces might look like, [2], or how to interpolate around noise to give a higher fidelity image [3]. However, there are very few examples of reconstructing images when you have all of the pieces available. DARPA opened a challenge in 2011 to design an algorithm to reconstruct a document from the complete set of shredded pieces of paper [4]. It was both an experiment to see if it could be used to reveal war zone secrets from destroyed documents, but also as a security measure to ensure that shredding was still an effective way to protect information. Within 33 days an algorithm had been developed by a small team in San Francisco [5] to suggest which pieces should go together, and a human would follow them and fix mistakes, but it was never made public and the accuracy wasn't declared. A more public example [6] looked a scrambled images, rather than shredded, but only allowed rows and columns to be swapped; essentially the same as shredding along both axes. It used an annealing algorithm to merge columns or rows into each other and is particularly effective at the start when progress is easier to make, but slows rapidly when large chunks are formed. It is also limited by straight line cuts along either axis, and requires all of the pieces. Around the same time an optimal algorithm was found [7] for rebuilding images that had been shredded along just one axis. This, again, does not solve the wider problem of rebuilding an entire image from the irregular shapes created by torn or chopped up documents. This could be invaluable in criminal investigations [8], war zone document recovery, and for personal use to repair damaged photos.

1.1 Overview

The objective of this project will be to build a three stage pipeline. The first stage will take an input of a complete set of image pieces and perform preprocessing on each to prepare them for analysis. The second stage will analyse and characterise each of the pieces, and make predictions on which pieces fit together. The final stage will take the predictions and reconstruct the image, hopefully producing the original. This will then be extended to support missing pieces and the detection of pieces that don't belong.

The pieces can have straight or curved edges, or a mixture of both, be rotated to any orientation, and will allow for "partial cuts", also referred to as "nested cuts". A full cut would be defined as splitting the original image from one edge to another using one continuous cut — the start and end edges may be the same edge if the cut is curved. A partial cut, on the other hand, would be a cut from one arbitrary point in the image to another arbitrary point in the image, such that it creates a new piece. For example, if you imagine cutting an image in half, and then taking one of the two new pieces and cutting it in half again, this is now only a partial cut through the full original image, as it does not span from one edge to another; rather from an edge to the centre of the original image. While a subtle difference in definition, it produces a very different problem as we can no longer assume that edges align perfectly.

The first stage of the pipeline has a relatively simple task definition. It should take each piece and turn it from a hard to use matrix, into a set of well defined objects that the second stage can harness. This will involve finding the shape of the image, the corner coordinates, the edges between each corner, and anything else that may be relevant or helpful.

In the second stage of the pipeline, each piece will each be characterised by a function of an edge to its score. These functions must be invariant to rotation to support any orientation, and invariant to edge length to support partial cuts, so that you can find the point where two edges fit together even if they are different lengths. This characterisation score will then be used to find similar pieces and suggest which to join together. Ideally the characterisation algorithm will work like a hashing function and give each edge a unique fingerprint, where similar edges have similar fingerprints. Paired with a distance function, this will form the predictor.

The final stage will take these suggestions, eliminate the incorrect ones, and rebuild the full image with no prior knowledge. It will need to figure out which of the pieces fit together the best using a score, as not all of the predicted joins will be correct. If the pipeline is successfully implemented, the end user should have no concept of the three parts. They should all work together inside a black box that takes an input of pieces and outputs a singular reconstruction of the original image.

Chapter 2

Background

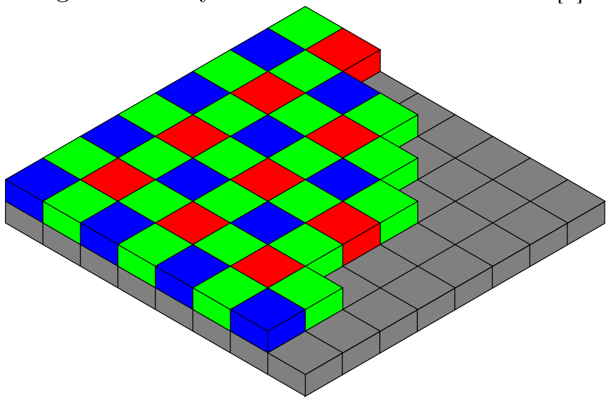
2.1 Digital Image Representation

As this project focuses on digital image reconstruction, and aims to replace the manual approach requiring a human to place pieces together, it is important to understand how these images are represented and how best to replicate human vision.

2.1.1 Camera Sensors

When a camera takes a photo, it is creating an encoding of the scene it was looking at. This is stored as a discrete matrix of colour intensities. The encoder is a sensor array where each sensor detects light intensity of different wavelengths. Normally there are three different sensor types for red, green and blue respectively. Commonly this array uses a Bayer formation [1].

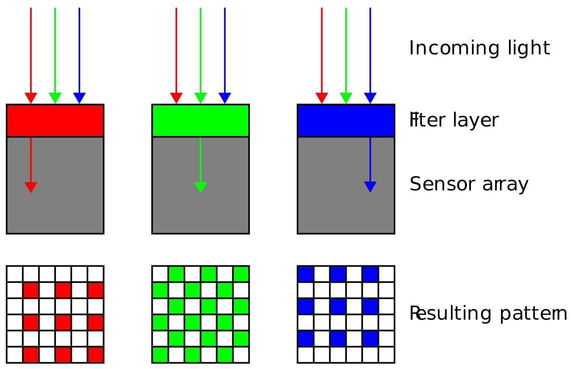
Figure 2.1: Bayer Formation Camera Sensor [1]



It is comprised of red, green and blue sensors, in a 1:2:1 ratio, which together make the full camera sensor. Green is the dominant sensor as the cone cells in our eyes are most sensitive to green light, so this best mimics human vision. The number of pixels that the final image contains is usually equal to the number of individual colour sensors, so a 12 megapixel (12MP) image was likely taken on a sensor with 12 million sub-sensors. When green light hits the green sensor, it is absorbed and an electrical signal is sent with the corresponding intensity; the same applies for the other two colours. Through this, we get 3 different matrices representing the 3 colour intensities at each point, but each is full of gaps as, for example, only 50% of the sensor will detect the colour green.

To resolve this, we then interpolate all of the gaps in each matrix using the surrounding values as a good estimate. We can now merge the 3 matrices into an RGB (Red, Green, Blue) matrix, and we have produced a digital image.

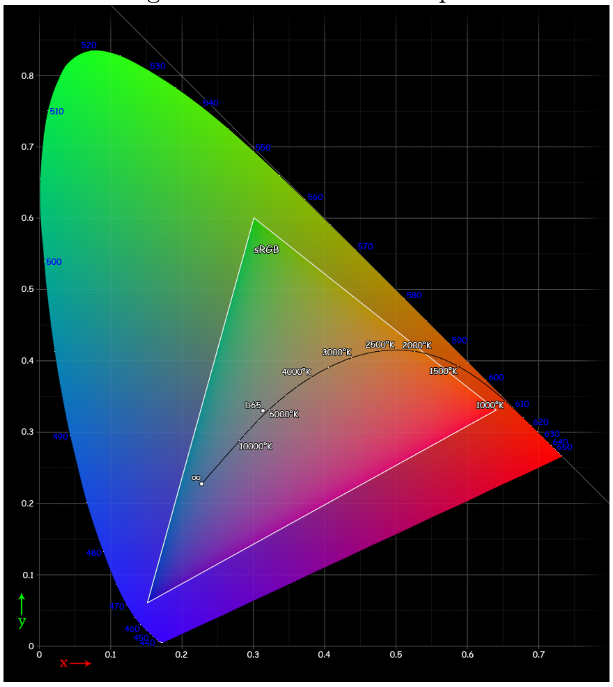
Figure 2.2: Bayer Formation Sensor Outputs [1]



2.1.2 Colour Representation

To actually represent the intensity of each pixel, we need to encode how intense a certain colour is. In general, we use 8 bits per colour, with an optional alpha (opacity) channel. This produces a 24 bit RGB range, or 32 bit RGBA range. As an example in RGB, the colour 0,0,0 would be black, 255,0,0 would be red, and 255,255,255 would be white. It is also important to note that not all colours that are visible to the human eye can be encoded due to only having 8 bits per channel, and because most screens are not capable of displaying them. In 1996, HP and Microsoft developed the sRGB colour space [9] that represented all of the colours that the 24 bit encoding could display. Today there are many more encodings that allow for far more colours, but this is still a good basis to work from.

Figure 2.3: sRGB Colour Space



2.1.3 Greyscale

In some cases, we want to represent an image in greyscale, such that each pixel has a single intensity between 0 and 255. Many computer vision techniques are built solely for use on greyscale images so it is important to be able to generate this representation. There are several different methods

[10], with the most obvious being taking an average of all 3 intensities:

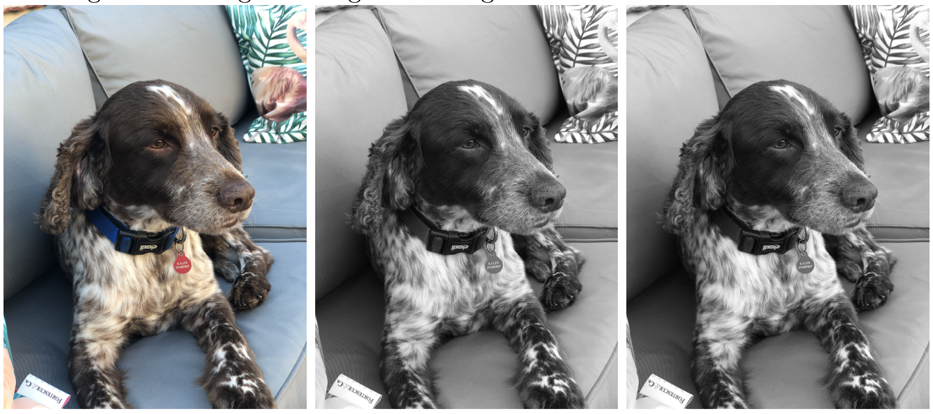
$$I = \frac{R + G + B}{3}$$

This is efficient and simple, but as explained above, it does not account for how our eyes see the world. Common photo editing tools use a more accurate version where, like the sensor, green takes priority. The coefficients vary between applications, but these are commonly used:

$$I = 0.3R + 0.59G + 0.11B$$

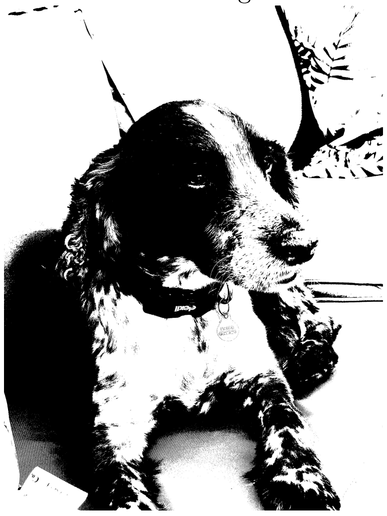
Both are shown below — the differences are minimal for most images, but it's important to understand the difference when rebuilding images that would normally be rebuilt using the human eye. Notice the blue collar is more pronounced on the average greyscale.

Figure 2.4: Original Image vs Average Blur vs Colour Corrected Blur



In some cases, you may also just want black and white. This can be done via thresholding; often on the alpha channel, or on one of the greyscale intensities used above.

Figure 2.5: Thresholding where I > 100



2.2 Convolution

Once you have an image in a matrix format, convolution [11] becomes an appealing technique to apply filters onto the image. Convolution takes another matrix, normally smaller than the original image, called the kernel. This is translated over each pixel in the image, and the sum of the products are taken to create an effect determined by the kernel chosen. Given image I and kernel K, convolution can then be defined as:

$$(K * I)[x, y] = \sum_{i=-\infty}^{+\infty} \sum_{j=-\infty}^{+\infty} K[i, j] \cdot I[x - i, y - j]$$

The bounds of the summations are usually set to the size of the kernel, but for the definition we can assume all matrices are infinitely padded with 0. If we weren't to assume this padding, then the output image would be smaller than the input image, as the kernel must fit entirely inside. There are many ways to avoid this issue, with the most common being:

- Pad the image with 0s
- Pad the image with the same value that surrounds the edge
- Allow the output image to be cropped

2.3 Gaussian Blurring

Turning an image into a greyscale version is one common requirement before using certain computer vision algorithms; another is blurring. Images, by our own creation, create a very sharp picture but when detecting features, lines and corners we want to reduce this sharpness such that only the key points are detected as most detectors will see a sharp image as noisy. Blurring is simple to perform with convolution, with the most basic example being an averaging filter:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This will add all of the surrounding pixels and normalise them, thus taking the average and bluring the image. While this is easy to implement, it does not provide a particularly good blur. It treats all pixels equally regardless of their distance to the centre, when in reality the neighbours in the immediate proximity to the centre should be weighted higher. This is especially a problem when using a larger kernel. This is where a 2D Gaussian function [12] becomes useful; if we use it to fill our kernel with values, the points closest to the centre (or equivalently, closest to the centre of the gaussian distribution) will have the highest weighting. The sigma value we choose for the distribution will be the "blurriness" of the output image, where a higher sigma value will generate a stronger blur. This is because the decay of the values becomes shallower as sigma increases, so points far from the centre are included in the blur with far higher weights. Notice we can ignore the constant factor normally present in a Gaussian distribution as we normalise our matrix anyway.

$$G(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

An extra benefit of the Gaussian kernel is its separable property. We know that the 2D Gaussian equation is the product of two 1D equations. We can then use this to generate two matrices from each equation where the product is equal to the 2D matrix used above. Convolution, by definition, is commutive and so we can apply the two filters separately in either order and get the correct output, and it will be the same as applying the NxN version of the filter. When using the NxN matrix, we have to iterate over every pixel in the image, and for each pixel we must perform an operation on the N^2 elements on the kernel. On the other hand, if we use the 1xN and Nx1 matrices, we again need to loop over each pixel, but we only need to perform N operations each time, repeating the process twice for each kernel. We have therefore reduced the complexity from $O(I^2K^2)$ to $O(I^2K)$.

$$G_x(x) = e^{-\frac{x^2}{2\sigma^2}}$$

$$G_y(y) = e^{-\frac{y^2}{2\sigma^2}}$$

$$K_1 = \frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix}$$

$$K_2 = \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix}$$

$$s.t. \quad K = K_1 \cdot K_2$$

Figure 2.6: Original image vs Average blur

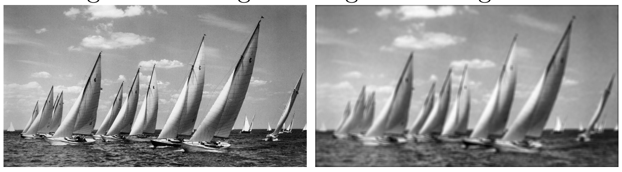
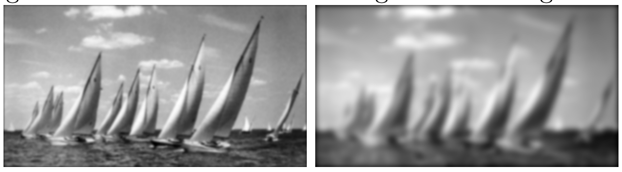


Figure 2.7: Guassian Blur with Sigma = 3 vs Sigma = 9



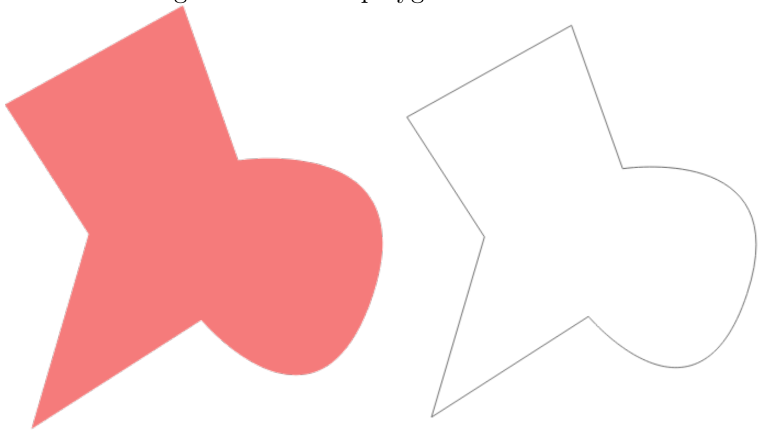
2.4 Edge Detection

Edge detection can be performed in numerous ways depending on the case at hand, and the accuracy needed. It also depends on the definition of an edge.

2.4.1 Border Edge Detection

If we define an edge to be the outside border of an image, then we can examine each pixel to decide which are edge pieces. Let's assume we have been given an image of a filled polygon with a transparent background and we want to find the edge pixels of this image. We have the matrix of our individual colour intensities, and we have an extra alpha channel to represent transparency. We know that each of the edge pixels has at least one transparent pixel in a neighbouring tile. By iterating over each pixel, and testing for a neighbouring pixel with an alpha value of 0, we can easily find the edge pixels. At most, 4 neighbouring tiles (each cardinal direction) for every pixel in the image will need to be tested, but you only need to find one instance of a transparent pixel before you can move onto the next point. In the worst case on an nxm image, this is $O(nm)$.

Figure 2.8: Filled polygon and its border



2.4.2 Canny Edge Detection

If we now change our definition of an edge to a feature within the image that represents an edge in the real word, like the edge of a building in a photo, then we come onto a very different problem. This is useful when looking for defining features within an image, especially if you are trying to orient two separate images. If you know there is part of a building in both, you can attempt to use edges to align the two images correctly, or as a metric of how likely two images are to be connected. Canny Edge Detection [13] is an algorithm for effective recognition of edges within images. Edge detection is one of the tecniques where the image needs to be blurred before detection can happen. Some detector kernels are the combination of a blur and gradient detector so prior blurring isn't always neccessary.

An edge in a greyscale image tends to be represented by a sharp transition from white to black. If we use a detector kernel to find this feature, we can then use the response to find edges. A Sobel filter [14] is a commonly applied kernel to this problem. It has two versions, one for each axis, and each returns the change in intensity, or gradient, at a given location when convolved on the image. It is the result of the convolution of two other kernels: the first is an averaging kernel very similar to the Gaussian blur, such that the image is blurred when the edge detector uses it. Points closer to the center are given a higher weight. The second is based around the central difference formula, where it takes the difference of the points either side of the center and uses this to estimate the gradient. If the pixels on the left and right side are roughly equal, then the convolution on the center point will cancel out to near 0. If one side is greater than the other, then the absolute value of the output will be a larger number. The value will be negative or positive depending on the direction of the gradient, which we can use to estimate the angle. Below are the two Sobel filters.

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

For an image I, we can then calculate from the two gradients the overall magnitude of the edge, and it's approximate angle.

$$g_x = S_x * I$$

$$g_y = S_y * I$$

$$M = \sqrt{g_y^2 + g_x^2}$$

$$\theta = \tan^{-1} \frac{g_y}{g_x}$$

This is a very effective way of finding edges, but it often overperforms and detects far more edges than you would like. There are a few ways of fixing this, such as using a larger sigma value when blurring the image such that less significant edges disappear, but Canny proposed a thresholding method which works a little more reliably. To start, pick a value T such that we will reject all values where $M < T$. This should This will create a rough lower threshold to remove any noise from the image, but before adding in a better threshold another problem needs to be dealt with.

When we use the Sobel filter, the edges it generates can sometimes be a few pixels thick as the gradient may change over more than a 1 pixel range. We want to suppress the non-maxima values of M to ensure that the edges we use are 1 pixel thick. Take each value of M remaining from the initial threshold and ensure that the pixels perpendicular to the direction of its gradient (from θ) are smaller than M. If not, drop this point from the list of potential edge points. We now ensure that all proposed edges are one pixel thick. Finally, pick a value τ such that just the strongest edges remain, and create your output image.

$$I_o = M > \tau$$

2.5 Harris Corner Detector

Finding edges in an image can be useful, but often the applications are limited unless you know where edges meet, and where they start and end. A corner detector can be used to scan an image for points that are likely a corner, and if used on an image that has already filtered for the edges, it can be incredibly effective. The Harris Corner Detector [15] is commonly used for this task. It often uses both Sobel filters to look for strong gradient changes but in both axes rather than just one. If there is a gradient change in one axis, it tends to imply that it is an edge, but both axes implies that two lines with different orientations have met at a corner. To measure the likelihood of a corner, first calculate the energy of the pixel using the two Sobel filters where I_x, I_y are the filters in each axis, and I is the combination of the two:

$$E(u, v) = \sum_{x,y} [I(x + u, y + v) - I(u, v)]^2$$

This takes the intensity at the given pixel u, v, and takes the difference of the intensity of the pixels within the window, finally summing the squares for the energy. As with the Sobel filter and Gaussian blur kernel, we can make a small modification to this equation to allow for a window function, such as a Gaussian distribution to ensure points closer to the centre make the biggest change to the energy.

$$E(u, v) = \sum_{x,y} w(x, y) \cdot [I(x + u, y + v) - I(x, y)]^2$$

For now, assume $w(x, y) = 1$ for all points. If the Taylor series for $I(x + u, y + v)$ is calculated, limiting to just the first partial differential, we can begin to change the equation into a matrix which will become particularly useful later.

$$I(x + u, y + v) \approx I(x, y) + uI_x + vI_y$$

$$E(u, v) \approx \sum_{x,y} [I(x, y) + uI_x + vI_y - I(x, y)]^2 \quad \text{ass. u, v are small}$$

$$E(u, v) \approx \sum_{x,y} [u^2 I_x^2 + 2uvI(x, y) + v^2 I_y^2]$$

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} \sum_{x,y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

$$M = \sum_{x,y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

With M, it is possible to start analysing corners. Using the two Sobel filters, we get a picture of the gradients in both directions. If the distribution of the gradients within the Harris detector window were to be plotted, there should be some large values in both the X and Y axes. If we use the two eigenvalues of M as a way to represent both axes, we can very easily decide whether a corner is present or not. Both eigenvalues will be large when a corner is detected, whereas only one will be large when an edge is found, and both will be small on a flat image. Finally, create a value R to act as the response value to decide whether a value is a corner.

$$R = \det M - k(\operatorname{trace} M)^2$$

k is usually between 0.04 and 0.06. At this point, R can be used to generate a response across the image, but it is particularly sensitive and normally produces duplicates for each corner due to the size of the window. In order to resolve the sensitivity, a constant must be decided in order to threshold which values of R should be discarded. Depending on the window size and image, this value can differ greatly, but the distribution of values stays roughly the same at a different scale. With this in mind, a good threshold would be $a \cdot \max R$ where a is between 0.005 and 0.02.

At this point, all corners should have been correctly identified, but some will have been detected several times. For example, the actual corner could be at $(100, 100)$, but you received large values of R for $(99, 100)$, $(100, 100)$ and $(100, 101)$. From here, it is possible to cluster the points using a clustering algorithm, leaving you with a single estimate for each corner.

2.6 Searching

When given a grid of tiles, such as an image with different intensities for each pixel, it's often necessary to navigate from one point to another, potentially with obstacles blocking the direct path. In order to find a path, usually the optimal path, an algorithm should be used to avoid an exhaustive search.

2.6.1 A* Search

If you know both the source and the destination, A* pathfinding [16] can be used to find the optimal path. You start at the source and iterate over each of the points you can move to within one step. For each, calculate a score; use the distance travelled so far added to the estimated distance to the destination (using a heuristic suited to the application). Store this value with the point itself in a priority queue ordered with the shortest distance first, call it the open queue. Add the source to a set of points that have already been visited and had all of their possible steps exhausted. Call this the closed set. On the next iteration, pop off the first element in the open queue and again find all the points that can be reached in one step that are not already in the closed set. Give each their own heuristic score and add them to the queue. Finally add this point to the closed set. Repeat until the destination has been found.

It is important to note that while A* is technically an algorithm to find the optimal path, this is not true if the heuristic is incorrect. The heuristic is assumed to never overestimate, as if it does a point could be pushed to the back of the queue incorrectly and never popped back off leading to a non-optimal route. An underestimating heuristic is less of a concern as it will still lead to an optimal path. Eventually the actual distance travelled will become greater than the underestimate allowing other items to pop off the queue. From this, we can see that a larger underestimate will approach the optimal path far slower.

Searching is an expensive operation, and while A* performs far more efficiently than undirected searches or brute force, it is still important to make optimisations where possible. Optimising the algorithm and data structures used such as using a priority queue as the open set, while useful, will not improve the performance as much as reducing the search space. The set of next steps should be limited to the greatest extent possible as most of the processing work is performed on each of them. Walls, barricades and any tile that is known to be unreachable or unusable should be removed from the space before running the algorithm for best results.

2.7 Image Joins and Distances

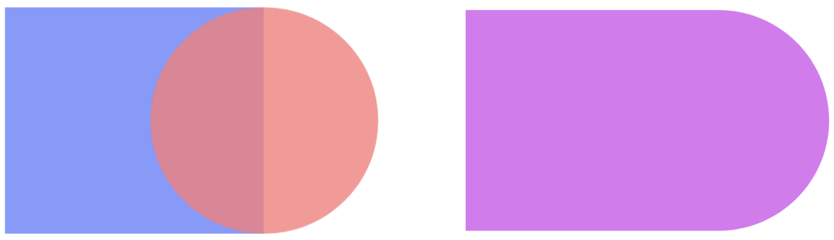
As images can be represented as matrices, it's possible to perform vectorised bitwise operations on them to join them together in different ways. This can be used both in the traditional way of joining two images in a performant manner, but also as a way to examine how two images interact with each other. For the purpose of the following examples, and for this project, we only care about the shape of the images so will assume that they are represented as a 2D bitmap, where a 1 represents a filled pixel and a 0 represents a transparent pixel / gap in the shape. All of these operations are particularly useful as the individual comparisons run in $O(1)$ and can easily be parallelised on your CPU.

2.7.1 Union

In order to join two images together, you can treat it as a set union. Both images will be basic matrices, and we can pad them on the right and bottom to match their shapes. By padding on the right and bottom, the shapes remain in the same coordinate space relative to the origin, which is traditionally in the top left hand corner. This allows us to match the shapes without moving the content. Then you can perform a bitwise OR on the two matrices joining the two images. This works as anywhere that has a 0 in both shapes will remain a 0, but all other values will become a 1, giving the union of the two shapes.

Below we have a blue rectangle and a red circle to show two different matrices. When we perform a set union on them, the image becomes one solid shape. This will be useful for reconstructing an image from pieces.

Figure 2.9: Union of Images

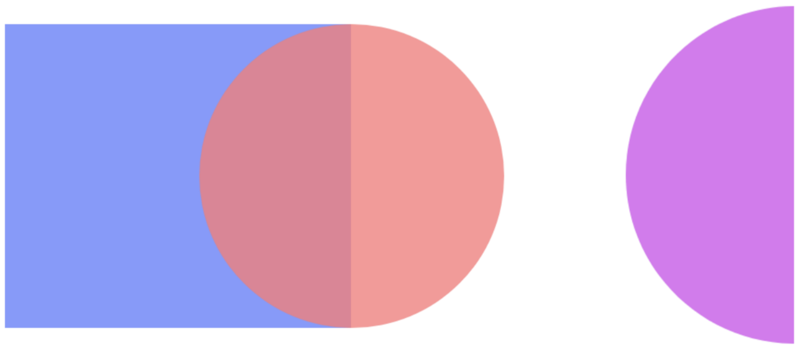


2.7.2 Intersection

Another way of joining images is by using an intersection. This is again based on set intersection. We pad the images on the bottom and right sides to match their shapes, and then perform a bitwise AND operation on the two images. This leaves us with the parts of the images that overlap, as the only places where a value of 1 will appear in the resulting matrix will be where the two images had a 1 in the same location, thus overlapping.

Below we have a blue rectangle and a red circle to show two different matrices. When we perform a set intersection on them, we can see the parts that overlap. We can perform an extra step after to sum the resulting matrix, which gives us a measure of how well two pieces fit together, or how much they overlap.

Figure 2.10: Intersection of Images



2.7.3 Hamming Distance

While the Hamming distance [17] isn't a form of image join, it is based almost entirely on the same premise. It is used to determine the distance between two patterns, normally represented as a bitmap. If the bits are the same (00 or 11) then it is given a score of 0, but if they differ (10 or 01) then it is given a score of 1. You move down the two bitmaps summing the score. The smaller the score, the closer the two patterns are to each other. We again take two matrices, this time of any dimension, that are of the same shape. In order to accelerate the above calculation, you can perform a bitwise XOR operator which can be parallelised easily.

Chapter 3

Implementation

The core functionality of this application will be to take a set of images which have been produced by cutting up a larger image, and attempt to reconstruct the original picture without any prior knowledge. It should be assumed that the pieces can have any number of edges and are not limited just to straight line cuts. They could have curved edges too. Pieces can be cut up further, and we refer to these nested cuts as “partial cuts” as they cut only partially through the original image.

The focus will be on full colour images, where all of the pieces are provided, though the designed application should not force these requirements upon the user. Rather, it should be able to handle, even if not particularly well, cases where the image is mostly text, where pieces are missing and where extra pieces are added in.

The application will be split up into a 3 stage pipeline. The first stage will import each of the pieces, convert them into a suitable format, and edges and corners will be extracted. The second stage will examine the edges and give them a characterisation so predictions can be made on which to join together. This middle stage will have an extension for partial cut support. The final stage will be taking these join predictions and eliminating the false positives by searching the solution space. It will then attempt to rebuild the image and output it to the user.

3.1 Piece Analysis

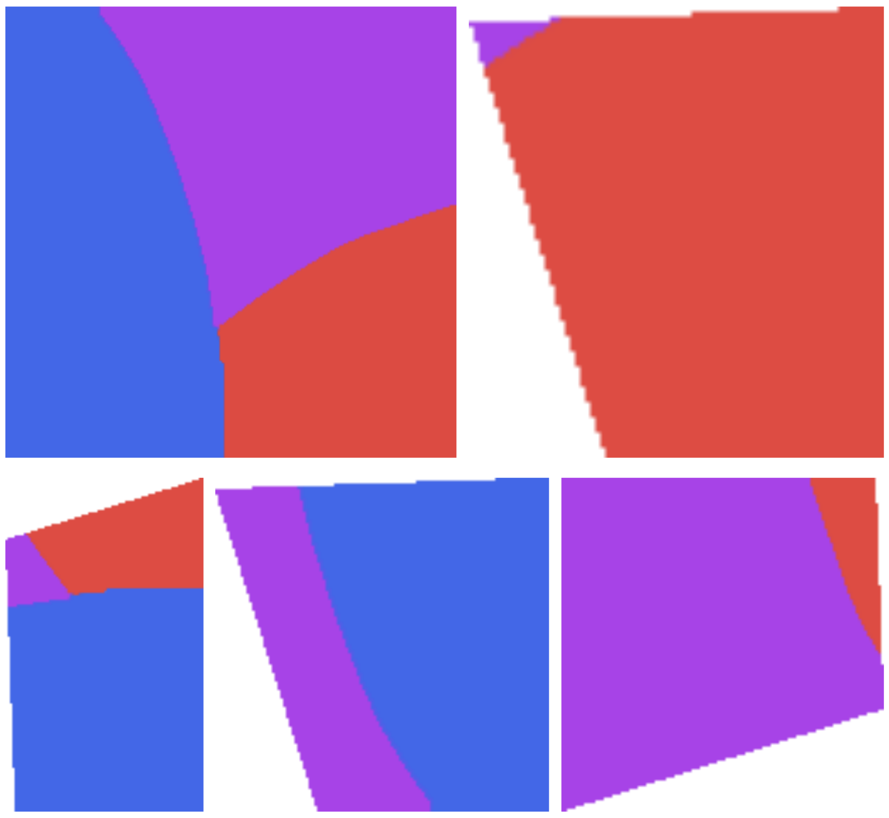
In order for us to find pieces that fit together, they must be imported and converted into a format that allows us to perform characterisations on each one. We assume each image is passed to us perfectly cropped against a transparent background, though this could easily be changed to support jpegs against a pink background as in the standard practise for formats that don’t support transparency. This project does not focus on how these images are prepared for our use, as this is outside of the scope. There has been plenty of research into automatically finding the bounding area of an image and cropping around it, such as “scanner” apps on mobile phones, so this technology could easily be paired alongside this.

The program takes a folder as its input, and reads in every image inside it as a new piece. For the current iteration, we assume that all of the images are pngs with a transparent background. This will allow us to find the shape of the image later. It then takes a second folder to put the output log and image into.

It is read in as a 3 dimensional array. The first two dimensions are the x and y axis of image, and at each pixel an array of size 4 is used to represent the red, green, blue and alpha channels respectively. At this point, we have n rectangular images, each with a mixture of solid and transparent pixels. They may have straight or curved edges, any number of corners, and be at any orientation. We must now begin analysing each piece to identify these traits ready for characterisation.

In order to explain each stage, we shall use the following test image which has been split into 4 pieces using full cuts, and then each of these 4 pieces has been rotated randomly by a multiple of 90 degrees (for simplicity, not because rotations are limited to 90 degree intervals).

Figure 3.1: Test image with 4 pieces

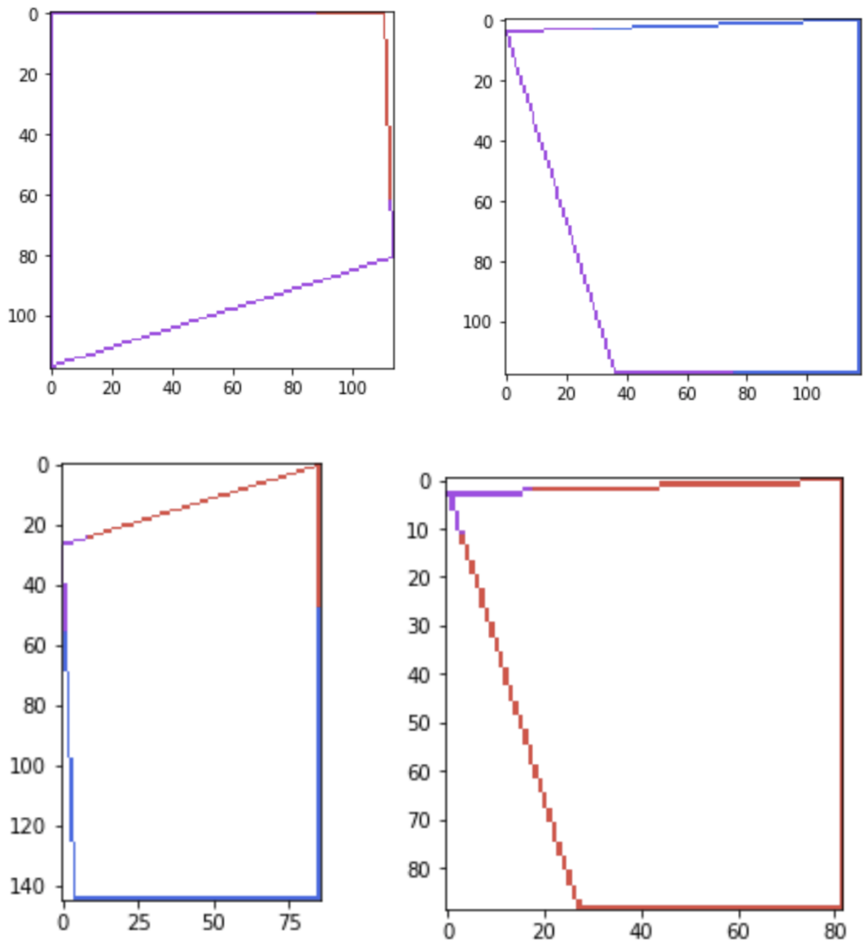


3.1.1 Perimeter Detection

Once the images have been imported into their matrix form, we need to start by figuring out where their perimeters are, and thus what shape they actually hold. As we have the assumption of one image per piece, there should be one continuous perimeter around the entire shape. If we can find this, we can then perform corner and edge detection on it to extract the features we need to join the pieces back up.

In order to make this as flexible as possible, there is no limitation on the shape, length, curvature or colour of each edge, and therefore of the border. This means we can only make one rule about the perimeter pixels — they must be adjacent to a transparent pixel in either the vertical or horizontal axes, or on the edge of the bounding box of the image. With this in mind, we can loop over the pixels in the image and apply this comparator. While this is quite an expensive operation, it can be performed to each pixel independently so could be vectorised and parallelised easily if it becomes a bottleneck. We produce a copy of the image with just the perimeter, and all of the colour data is left unchanged so we can use these matrices later.

Figure 3.2: 4 pieces with their perimeter



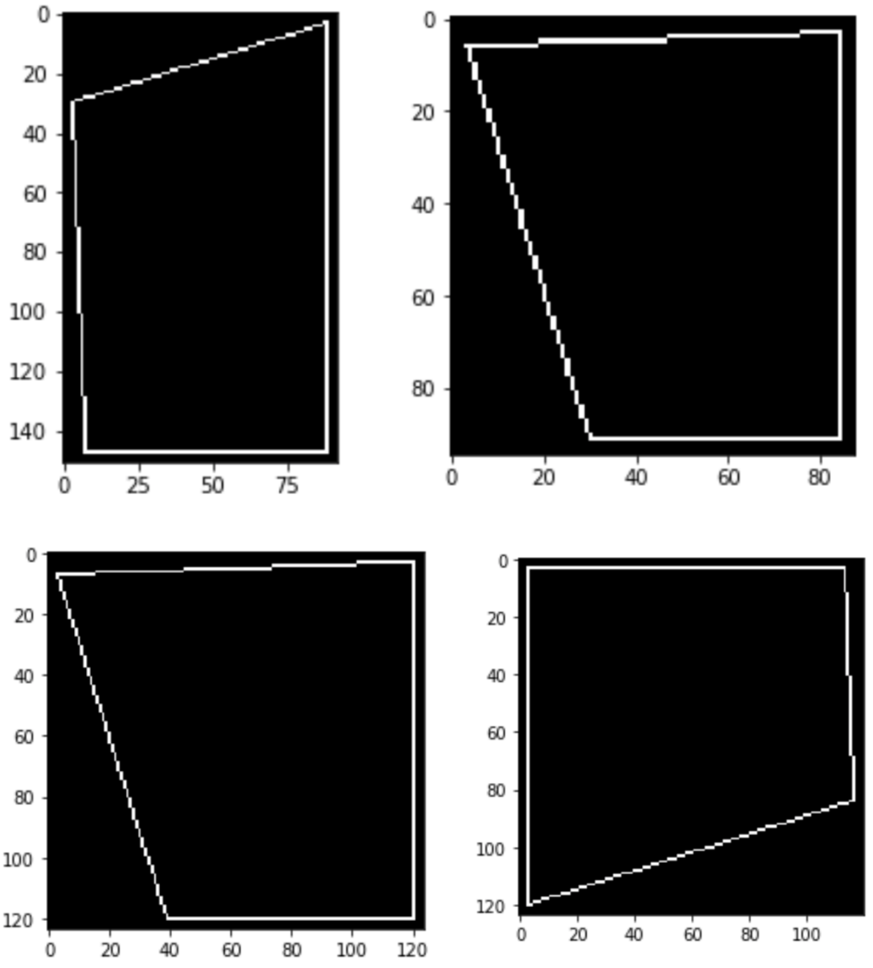
3.1.2 Corner Detection

Detecting the corners of an image is a challenge, especially when its is stored as a matrix. Our eyes detect corners where there is a significant change in gradient, but this is hard to detect when sloped lines in an image are represented by a repeating pattern of horizontal and vertical lines, as can clearly be seen in the images above. For this reason, the idea of following the perimeter and detecting changes in the gradient is hard to build accurately.

Instead, computer vision techniques can be used to find the corners. The Harris corner detector is the best known technique, and is performed by convolving the image matrix to find points where the gradient changes in both axes. This implies that there must be two lines that meet.

In order to use the Harris corner detector, we first perform some pre-processing. First, the image is turned to black and white. This isn't a hard requirement, but if we make the edges more prominent, the edge detector that the corner detector relies on will be more accurate. Next, we use a window to convolve on the image matrix, and we need to ensure that the window fits. From experimenting, a 5x5 window size gave accurate results so we need to ensure that there is a 2 pixel padding on all sides of the image such that the centre of the window falls inside. Our images are now ready to have the corner detector ran on them.

Figure 3.3: 4 pieces with their perimeter



Using OpenCV, the open-source computer vision library, we can easily apply a performant version of the Harris Corner Detector onto each of the pieces. This version takes 3 parameters:

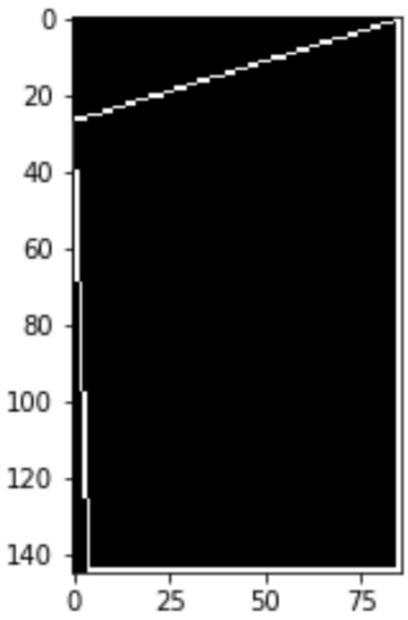
- The window / neighbourhood size which generates the eigenvalues for the corner detection
- The aperture size of the Sobel filter derivatives used in each axis to detect gradient change
- The free parameter k from the original paper, which normally sits around 0.04

Our implementation uses a window size of 3 to keep complexity low and an aperture size of 5 as mentioned above. This aperture size works well as our lines are exactly 1 pixel thick, so a 5x5 Sobel filter will fit neatly on top to detect the changes. Finally, we use the recommended value of $k = 0.04$.

This gives us a weight for every pixel in the image, where it is only large if both of the sobel filters detected a significant change, and thus detected a potential corner. We now need to threshold the weights to select candidates. There is no constant value that works well for thresholding, as the size of the image, severity of the corners, and other similar factors can change how high the values are. However, as any non-corner pixel will give a *very* low value, we find that the entire 99th percentile contains the corners. Our threshold, therefore, can be defined as $T > 0.01 * weights.max()$

A side effect of the Harris detector is that it will give a high reading for any point within the 5x5 window that is next to a corner, so each corner is flagged multiple times by slightly different coordinates. We need to cluster these and select just one coordinate to work with, suppressing the rest. As we are only going to be out by around 3 pixels, it is not particularly important which we pick. We can still reconstruct an image almost perfectly if the edges are out by a pixel or two. To start, the padding is reversed so that the corners are relative to $(0, 0)$. Below are the corners that were detected for this image.

Figure 3.4: 4 pieces with their perimeter



```
[  0  84]
[  0  85]
[  1  82]
[  1  83]
[  1  85]
[  2  85]
[  3  85]
[ 26   1]
[ 27   0]
[142   4]
[142  85]
[143   4]
[143  85]
[144   4]
[144   5]
[144   6]
[144  83]
[144  84]
[144  85]
```

Using a basic hash to sort the points into clusters, we can suppress the duplicates. We can take each point and give it a value of $x + 31y$. Once sorted, the clusters become clear. You can see there are 4 distinct corners.

```
[ 27   0]
[ 26   1]
[142   4]
[143   4]
[144   4]
[144   5]
[144   6]
[  1  82]
[  1  83]
[  0  84]
[  0  85]
[  1  85]
[  2  85]
[  3  85]
[144  83]
[144  84]
[142  85]
[143  85]
[144  85]
```

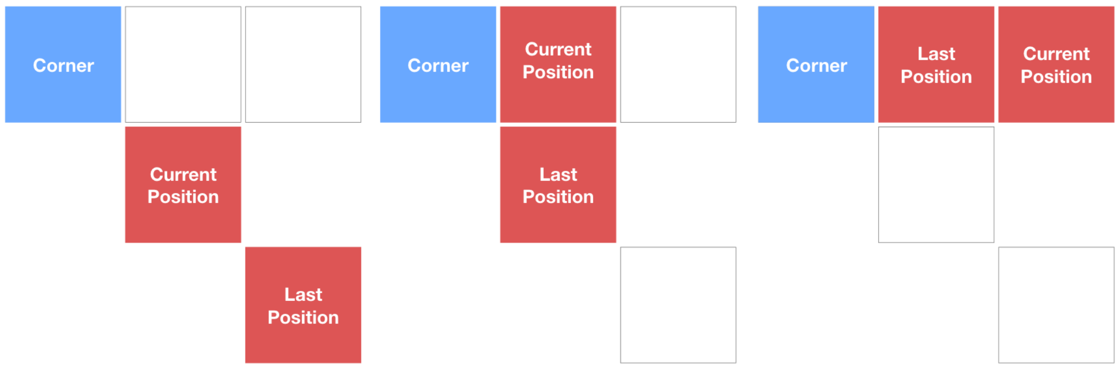
We know that the spread around a corner relies on the 5x5 window, so can use this to estimate the maximum distance between two points that are marking the same corner. If the maximum distance from the corner in either axis is 5, then the diagonal distance will be $\sqrt{10}$. We can then take the first point, and remove any that are within a distance of $\sqrt{10}$. This leaves us with the following points, which correctly identify the corners within a tight tolerance of only a couple of pixels:

```
[ 27   0]
[142   4]
[  1  82]
[144  83]
```

3.1.3 Edge Detection

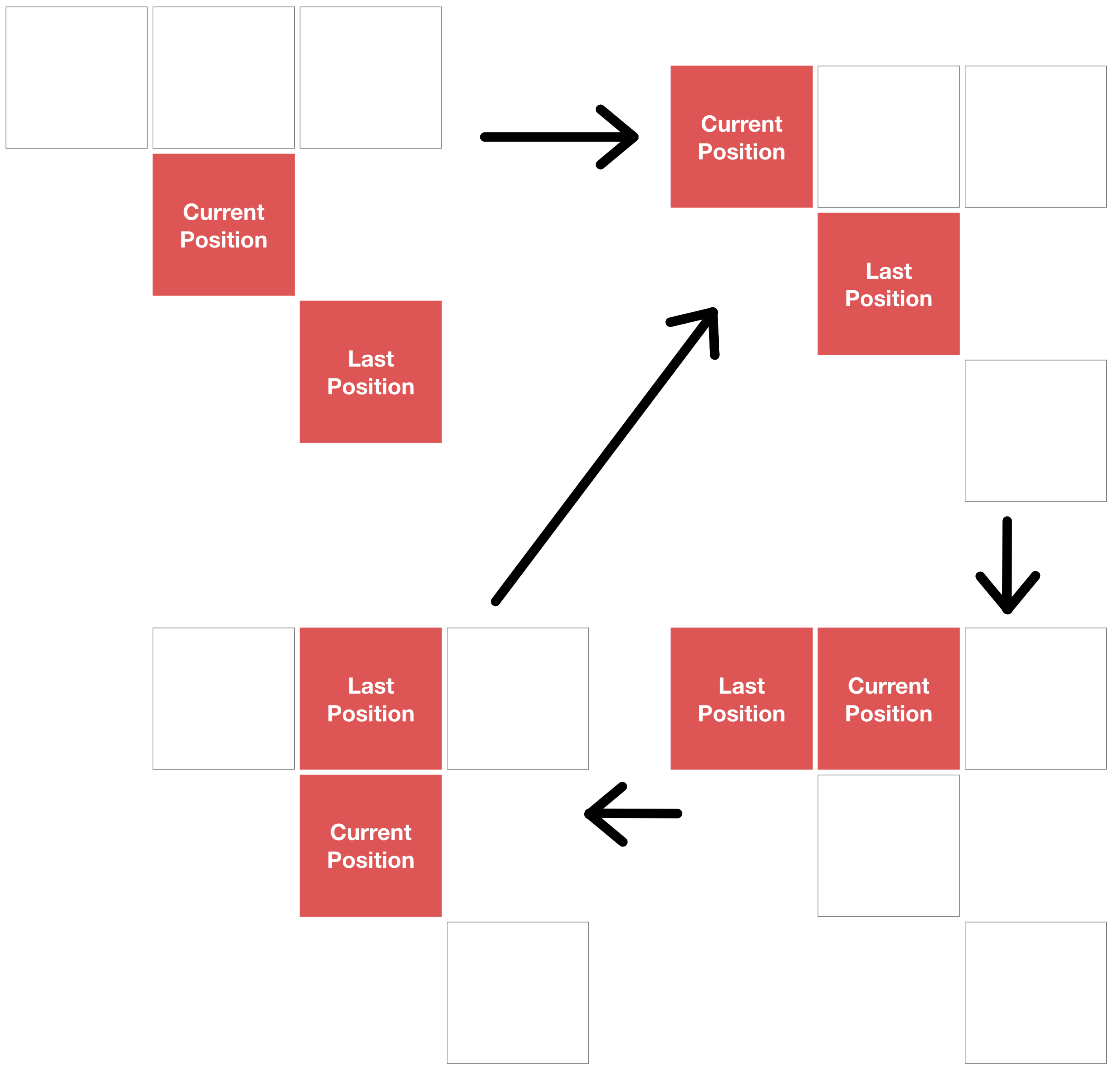
Once the corners have been identified, it's necessary to find the edges that lie between them so that we can divide up the perimeter into the parts we will use to reassemble the full image. On the surface, this seemed like a trivial task — simply walk the perimeter until you find a corner, and count this as an edge. Repeat until you return to the start, ensuring that you only move forwards by keeping track of the last position. This made the assumption that there was a perfect 1 pixel path around the image, and by following it you would touch every border pixel exactly once. As it turned out, it was possible to walk all the way around a perimeter without ever touching a corner due to the rule dictating what was an edge piece. As any pixel with a transparent neighbour became a border, acute corners had the corner pixel extended from a complete perimeter loop. This allowed the path-walker to walk straight past it.

Figure 3.5: Missing the corner



Further, in the case that it did touch the corner, it was possible for it to loop indefinitely as it got stuck in the corner. This was because we only tracked the last position, not a history of every pixel visited. The only solution to this was to keep a list of every pixel visited, and check it's contents on every move. This suddenly made a simple walking algorithm far more intensive on both the memory and performance overhead. It also doesn't fix the previous issue.

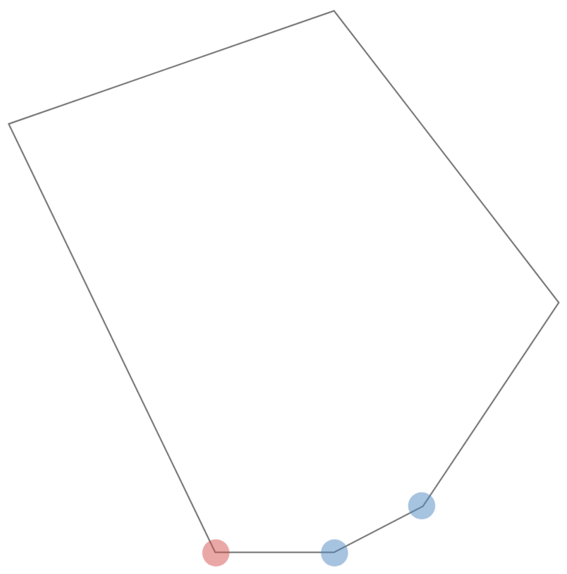
Figure 3.6: Looping in the corner



An undirected path-following was clearly not the appropriate solution. A simple idea might be to assume the line is a perfectly straight set of pixels from one corner to the other. In this case some basic geometry could define the edge. This, however, would limit us to just perfectly straight edges, and would rely on the corner detector being near-perfect. This seems like an unnecessary constraint to impose on the solution when there are better alternatives. Instead a directed search was used. A* search is often used for pathfinding, and seemed very suitable to this problem. It relies on a cost function, and a heuristic function. The cost dictates how expensive moving to a pixel is. The border pixels were given a cost of 1, and the rest were given a value of $MAXINT$ to ensure they would never be used. The heuristic should be an estimate of the distance from the current position to the target (the corner we are trying to get to). This must not be an overestimate or the algorithm will fail, but too much of an underestimate will cause the performance to suffer as it thinks it is closer than it is. The cartesian distance works well here as the actual distance will always be greater than or equal to it. With these set, we can provide a start and end position, and it will expand the frontier from the start point along multiple paths finding the shortest route, returning both the path and the distance. This has resolved both of the issues explained above. A* is often seen as quite an expensive algorithm as it is usually ran on a grid and can expand the frontier along thousands of paths to find the shortest one, however we are limiting the search to a closed loop which means the frontier is often only expanded down one or two paths making this a far more reasonable approach.

While this solves the challenge of joining up two corners, it does not tell us which corners to connect together. We must ensure that each corner is only joined to the correct two corners that branch from it. An easy mistake to make would be assuming that the closest two corners are the two we wish to connect to, but as shown below, the red corner should not join to the two (closest) blue corners.

Figure 3.7: Closest corners aren't the correct corners



The naive approach, therefore, would be to try and connect every corner to every other corner. Then for each, examine each path to ensure it does not contain a third corner, and select the two paths that remain as the connections . As explained previously, our A* search is far more performant than a traditional grid search, so this isn't a terrible idea, but improvements can be made.

Will will create 3 definitions called the "level of connection":

- Unconnected (A corner that has no connection to any other corner)
- Partially Connected (A corner that has connected to one other, but still needs to make a further connection)
- Fully Connected (A corner that has been connected to both of its neighbours)

We will also define a function which upgrades the level of connection by one. Unconnected will become Partially Connected, and Partially Connected will become Fully Connected.

To start, every corner is Unconnected. Take one at random and perform the following:

- Use A* to find a path to every other Unconnected and Partially Connected corner
- Remove all paths that contain another corner
- Keep track of the two remaining paths and set those as our connections
- Mark current corner as Fully Connected
- Upgrade the two corners we connected to
- Repeat for until there are no Unconnected corners left

Now we are left with Partially Connected and Fully Connected corners. For each Partially Connected corner:

- Use A* to find a path to every other Partially Connected corner
- Remove all paths that contain another corner
- Keep track of the one remaining path and set that as our other connection
- Mark current corner as Fully Connected
- Upgrade the corner we connected to
- Repeat for until there are no Partially Connected corners left

We have now joined every corner to its appropriate neighbours in a far more efficient way. If we had tried connecting each corner to each other corner, we would have had a performance of $O(n(n-1)) = O(n^2 - n)$ for n corners, measured by counting the number of calls to the A* function. Instead, this algorithm finds at least one Fully Connected corner on each iteration (though sometimes as many as 3), and therefore takes at least one corner out of the calculation, except on the last run when it will find the two final Fully Connected corners. Roughly speaking, this makes the complexity $O((n-1)+(n-2)+(n-3)+\ldots+3+2) = O((n^2+1)/2)$ which is a significant improvement as n increases. On a 4 cornered shape, this reduces the A* calls from 12 to 5.

Running this on the example image and corners, we get the following output, which reflects the 4-sided shape, 5-call scenario:

```
Connected (27, 0) to (1, 82) and (142, 4)
Connected (144, 83) to (142, 4) and (1, 82)
```

3.2 Edge Characterisation for Joining

Each piece of the image has now been turned from a rectangular matrix of pixels into a set of descriptors that mark out the corners, edge pixels, and general shape of the image. We have laid the foundations for the image reconstruction to begin, so now we must take each shape and through some metric, figure out which pieces probabilistically fit together. We will design a set of functions which take an edge as an input and produce an output which we can compare with other edges to decide which pieces may fit together. By using the edges alone, the complexity is reduced from using the entire image which becomes particularly important when we start examining nested partial cuts. The objective of this characterisation was to find a good balance between performance and accuracy, so that this could reasonably be scaled to large numbers of pieces. To ensure support for any shape of edge, it should treat the paths purely as lists of pixels, and not assume that they are straight edges. We will start by assuming there are no nested cuts and build up to a algorithm that supports them too.

3.2.1 Naive Edge Characterisation

The initial target was to find a way to join up the 4 pieces from the test image used throughout this project. These 4 images have very distinct edges, which are predominantly one solid colour along them. The length of each the sides was also quite unique. Harnessing these two facts led to the first naive solution.

For each pixel along an edge, take its RGB components and find the one with the greatest value. Call this the dominant colour. Sum up the red, green and blue dominant pixels individually, and keep track of the total. This gives us a very rough estimate of how much red, green and blue is in each edge, along with the length. With this available, we can then find edges which have roughly the same distance, and compare their colour ratios. For this, we define a distance function that takes the sum of the absolute values of the difference between each of the RGB and length figures. If the distance is less than 10% of the length of the edges, we propose it as a potential match.

For the example images, this is surprisingly effective. It finds all of the correct joins, but there are a lot of false positives too as several of the edges are blue dominant and around the same length. This is where this algorithm fails — it does not account for the actual pattern of the edge, rather it simply says “this is blue and red”. This has no issues if a side is a solid colour, but if an edge is blue-red-blue, and another is red-blue-red, it could propose a match when our eyes would easily be able to tell the two pieces do not fit together at all. This leads us to the first improvement.

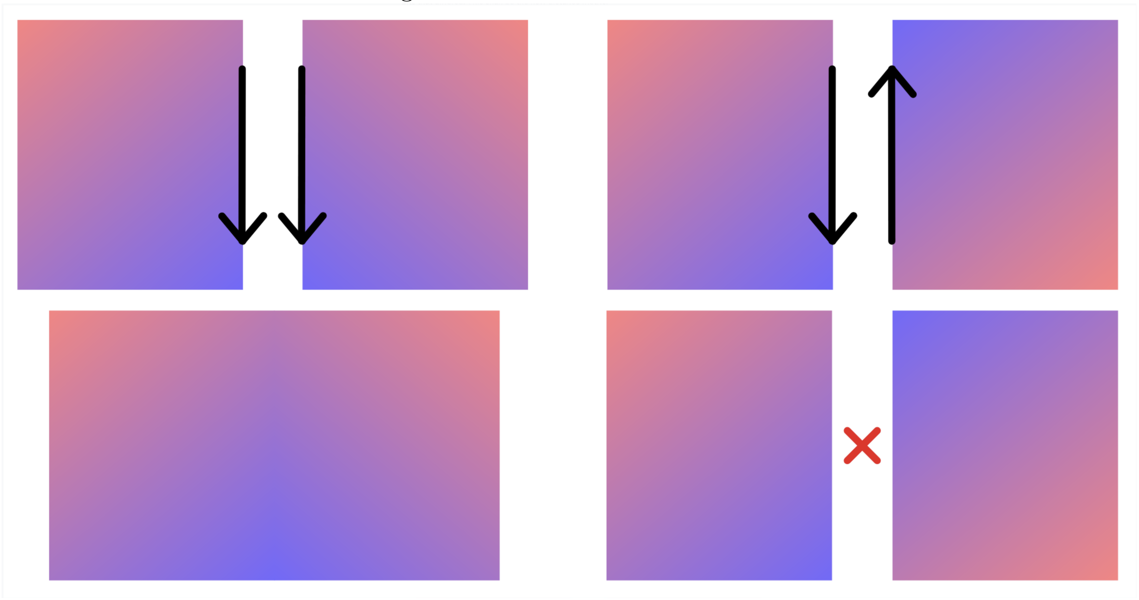
3.2.2 Improved Edge Characterisation

The main flaw in the previous attempt was the lack of pattern context. Knowing whether a side is red dominant is good, but it would be even better if we knew *where*. In order to expand to more complex edges, we will introduce a bitmap system where every bit encodes the RGB dominance for a subsection of the edge. We will use a modified bitmap where each index contains either 100, 010 or 001 for R,G and B respectively. The function that generates this bitmap will have a new parameter n which defines the size of each subsection. A value of 5, for example, would split each edge into 5 pixel chunks and the bitmap would have a set of bits for every one of these. Essentially, we are using exactly the same algorithm as before, but chopping up each edge such that when put back together, we have more context — we will be able to determine the pattern of the pixels. This should not be much more expensive than the previous implementation as we still only count each pixel once, but there is additional overhead chopping the edge up, and memory overhead having to store a characteristic for each subsection rather than just the entire side. If n is too large, its worth noting that we will get little performance gain over the naive attempt as we will again lose any patten information. If n is 1, it will be the same as comparing each pixel, which gives us a lot of flexibility. This may lead to performance issues on particularly large edges where there will now be hundred or thousands of (albeit inexpensive) comparisons.

We now define a slightly different distance function. For a start, we only compare edges who’s bitmaps have the same size. Next we apply a slightly modified Hamming distance function to the bitmaps. For this, we can harness more bitwise operators on arrays to maximise our performance. Taking the sum of the XOR of the two bitmaps will give us a count of the number of bits that differed. This shall be the new distance metric.

Inadvertently, this has introduced a new problem. When we joined the corners, we did not give a direction to the path. This means when we compare the two bitmaps, one of them might be the wrong way round and not actually be a valid join. This can not easily be solved at this stage, so will be part of the false positive elimination in part 3, but worth acknowledging here. At this stage, we are simply outputting suggested joins, of which the output space should contain all of the correct joins. A human would then be able to filter the invalid ones, though the third stage of the pipeline will perform this task in place of human intervention. To ensure all correct joins are found, edges will be tested against every suitable edge in both the forwards and backwards direction. Below is an example. On the left, the two bitmaps match and as they are in compatible directions, the join is possible. On the right, the two bitmaps match, but they are not in compatible directions; you would need to flip the image on the right for the edges to align which would cause the two pieces to sit on top of each other. This is clearly not a valid join, and we will eliminate this later.

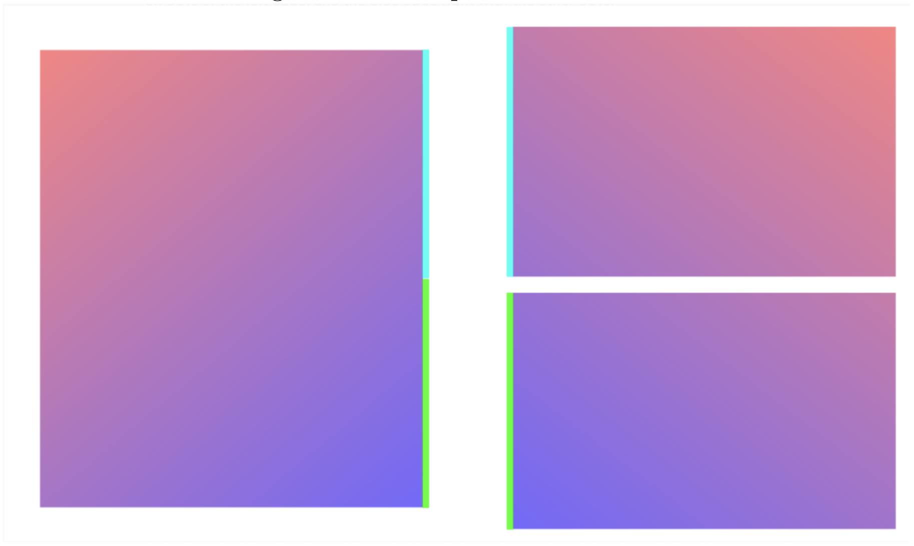
Figure 3.8: Valid vs Invalid Joins



3.2.3 Handling Partial Nested Cuts

The improved version of the edge characterisation algorithm is a good start, and now has better performance than the previous version, and can eliminate some of the false positives with the pattern context. Its limiting factor is its ability to only work on full cuts - it assumes that every edge will perfectly fit next to another, or to none at all. When we consider nested cuts; cuts which are made on the pieces themselves to turn them into further, smaller pieces rather than on the whole image, this algorithm falls apart. As shown below, we can no longer rely on edges being the same length, nor matching up corner to corner. The green edge on the right will match to 50% of the left edge, and the blue edge on the right with the other 50%.

Figure 3.9: Example of Partial Cuts



To handle this scenario, we need to extend our current implementation to not just find the edges that match, but where they match best so that they can be placed at any point along the side. Fortunately, our bitmap approach from before can easily be adapted to serve this purpose, and we ensured that we used vectorised bitwise operators which will be a major benefit as our number of comparisons will increase dramatically. Our new algorithm is as follows:

- Split each edge into n sub-edges of size s
- For each edge, create a bitmap to represent whether each subsection is red, green or blue dominant
- Slide another shorter edge along it, and get the Hamming distance of the bitmaps at each increment
- Return the coordinates and distance score for the point where the distance is minimised. This will be the best place to join the two edges

This has now allowed support for partial cuts, but there are a few things to bear in mind due to the greatly increased number of comparisons.

- If s is 1 pixel, then we will need to slide each edge along each pixel of every other edge. We should always find the best match, but at the cost of performance which will suffer exponentially with the number of pieces
- If s is small, say 5 pixels, then we will cut our comparisons down by a factor of 5, and still get an accurate estimation of where the match will be
- If s is too big, the bitmaps may not overlap neatly, so the accuracy will drop dramatically and matches will be lost

3.2.4 Full Color Edge Characterisation

Partial cut support is now implemented, though using a few test images shows it is struggling to find the joins, and it generates a *lot* of false positives. This is due to the poor choice of pixel characterisation we are using. The RGB dominance was fine for solid edges like the test image, but say we have an edge with an intricate blue and purple pattern — this would look no different than a solid blue edge as purple is still marked as blue dominant. Further, the intensity and brightness of a colour is important too. If we take two RGB pixels, $r10 : g0 : b0$ and $r234 : g200 : b50$, then our dominance metric would mark them both as red.

We need to create a new function that will generate a better, if not unique, signature for an edge. Let c be a 3D coordinate representing the red, green and blue channels of the colour. Let the distance between two points be the Euclidean norm: $\sqrt{(r1-r2)^2+(g1-g2)^2+(b1-b2)^2}$. Let's now expand this to our subsection model by grouping sets of pixels together. So long as the grouping size s is small, we can let our c value be the sum of the RGB values of the pixels in the set. This is because the colours within the group should not vary greatly. If they do, a smaller s value should be used. We can now no longer use the bitwise operators to find the distance, instead we should now take the two sets of grouped edges and take the distances between each of the pixel sets with the Euclidian norm described above. Finally sum and divide by the length of the shortest edge to normalise the distance. Going back to the pixel example before, the distance using the dominance metric would be 0, but using the new metric we have a far more appropriate distance of 22. This is now an accurate, partial-cut-supporting pipeline, and we can begin reconstructing the image.

3.3 Image Reconstruction

The final part of the pipeline is taking the join predictions and attempting to reconstruct the image. We have already highlighted some of the challenges, and we need to ensure that each is handled appropriately. The first is a large number of false positives, where the joins are absolutely valid, but not correct so will lead to a totally incorrect final image. As joins are made, other joins will become impossible which will give us an idea of which are the correct ones. Through a process of backtracking and join elimination on the output space, it should be possible to find the optimal, correct set of connections that produce the original image. The second issue is the invalid joins formed by the lack of edge direction which cause the pieces to entirely overlap. Whenever we make a join, we should ensure that the overlap is sensible. Due to our corners not being entirely accurate, some overlap should be expected, but a sensible limit of, say, 10% of the shape volume would be appropriate.

3.3.1 Preparing for Joins

In order to search the solution space, we are going to need to perform a lot of joins, potentially backtrack, and try again. This is going to be an intensive and somewhat complex operation so we need to ensure that sensible performance measures are taken. The main functions will be rotation of the image, rotation of coordinates and overlap detection. These can easily be done on the existing image matrices, but is unnecessarily expensive as they hold all of the colour data, which is irrelevant for the tasks we wish to perform. Instead, a quick pre-processing step shall be made to convert each of the images into a binary matrix, where a 1 will be used for the pixels, and 0 for the transparent area surrounding it. This will be a lightweight representation purely of the shape of the image. We can test and search with these matrices, and then replay the desired operations on the actual images at the end, reducing some of the overhead.

Next we should consider how a join is to be performed. A fundamental part of the pipeline is support for rotated pieces, so we can not guarantee that they are already aligned and ready to join together. We will be fed instructions such as "join image 1 along $(25, 100)$ to $(25, 300)$ with image 2 along $(15, 50)$ to $(215, 50)$". With this, we are implicitly told the orientation that the two pieces need to be at. With some basic trigonometry we can use the following formula to rotate both of the edges into the same orientation. For simplicity and consistency, we shall rotate all of the edges into the vertical axis before joining. The first coordinate will be oriented at the top of the edge in the vertical axis, and the second coordinate at the bottom. Image rotation can be done through matrix multiplication, but we shall use a library where this has been optimised.

```
// Find the angle that the edge slopes away from the vertical axis at
theta = atan2(x2 - x1, y2 - y1)
// Rotate the image by that amount (negated for clockwise rotation)
rotateImage(-theta)
```

When images are rotated, they often no longer fit within their bounds. This will then make using their coordinates very difficult, and their matrices will be broken and unusable for overlap values. To avoid this, images will be padded before rotation to give ample room for rotation at any angle. If we consider the worst case where a tall, thin rectangle is rotated 90 degrees, then the width should be padded by 50% of the height before rotation for safety. Padding is a particularly cheap operation, so setting a rule of padding top, bottom, left and right by 50% of the largest of the width or height will ensure all rotations work.

Now that the images can be rotated, we need a function that can rotate points too so that we can update our coordinate system. Each edge needs to be updated every time the image is rotated to ensure we can still use it. First, the 50% padding applied earlier needs to be used to translate the points. This is one benefit of using a consistent 50% padding as it makes this calculation incredibly simple. We will write a function that takes point $p = (px, py)$, rotational origin $o = (ox, oy)$, the angle, and the padding as its inputs, and translates p by the padding before rotating it around o by a given number of radians. For most of our use cases, o will be the midpoint of the padded image.

```
px = px + padding
py = py + padding

// Again, negate the angle for clockwise rotation
qx = ox + math.cos(-angle) * (px - ox) - math.sin(-angle) * (py - oy)
qy = oy + math.sin(-angle) * (px - ox) + math.cos(-angle) * (py - oy)
```

Finally, we will define a function that joins two images that have been appropriately padded and then rotated. This assumes that the edges are along the vertical axis, and the right hand image has been given extra padding on the left to ensure the edges align perfectly inside the same coordinate space. Overlap can be measured very easily using bitwise operators, as we can bitwise AND the two matrices together and sum the result to find the number of pixels that overlap. In order for us to do this, the two matrices must be of exactly the same shape. By padding zeroes only on the bottom and the right, we can adjust the size of the smaller matrix without adjusting the position of the shape.

3.3.2 Searching for the Reconstruction

We can now use all of the above to generate a function to take a join suggestion, join the two pieces together, and output the overlap for us to analyse. First we take the two pieces and turn them into the simple single-bit matrices. Then for each, find the largest of the width and height and half it, then pad each side by that amount as the rotation buffer. Take the edge for each piece and use it to rotate the image such that this edge is aligned vertically. Rotate the coordinates of the edge by the same angle. Find which of the two pieces has the highest coordinate, and pad it down from the top such that it aligns with the other. Find the image that needs to be on the right, and pad it on the left by the width of the shape plus the left padding of the other image. We now have the two images perfectly aligned ready to connect. Use the join and overlap function to pad them to the same size, join the two matrices and return the overlap of the two pieces. If the overlap is greater than 10% of the area, we reject the join and try the next one, otherwise count it as a success. Loop over the rest of the suggested joins and remove any that involve both of the two edges as these are no longer viable, and then find any others that involve one of the edges, and translate the coordinates to the new coordinate system of the joined image. This has now completed a full join and prepared for another to take place, so we can begin to put this into a search system.

In order to perform a successful search for the “best” set of joins, we need to define what we mean by “best”. We have already mentioned that too much overlap between pieces suggests that a join is not valid, and we might find that by joining two pieces together, the third piece we join now doesn’t fit very well. This makes the overlap metric one of the best guides. It is worth noting here that we may have extra pieces or missing pieces thrown which will challenge the algorithm. We need to take this into account, for example if we have 10 pieces, join 3 together with a very small overlap, and can then not make any more joins due to the 10% overlap rule, this would score highly with just the overlap metric, but it might be the case that the correct solution involved all of the pieces. Therefore we should use the number of pieces involved in the reconstruction as a second metric, aiming to maximise it while minimising the total overlap. Finally, it may be possible to put all of the pieces together perfectly in more than one arrangement, so we need a metric to measure the likeliness that the final arrangement is sensible. We can assume that most of the images we reconstruct were originally rectangular, so a measure of how rectangular the final output is would be sensible. As we have the coordinates of all of the edges, we can simply find the top-left most and top-right most corners, align this edge to the horizontal axis, trim any padding, and then find out how much whitespace remains. We can minimise this value, as a perfect rectangle will have no whitespace. This will ensure that extra pieces thrown in to trick the constructor should be suppressed, and missing pieces will still be supported as the most rectangular shape will still have the least whitespace in most scenarios, with just one space missing.

Below are some examples of how these 3 metrics can work together to remove the false positives and find the best result. The first shows how one bad join can cause other joins to have severe overlap, so the first metric will attempt to fix this. The second shows that sometimes it can't use all of the pieces, so maximising the number will likely bring us closer to the actual solution. The third and final example shows that there are several ways to put the image together correctly, but the more rectangular it is, the more likely it is to be correct based on our assumptions.

Figure 3.10: Example of Incorrect Joins Causing Overlap

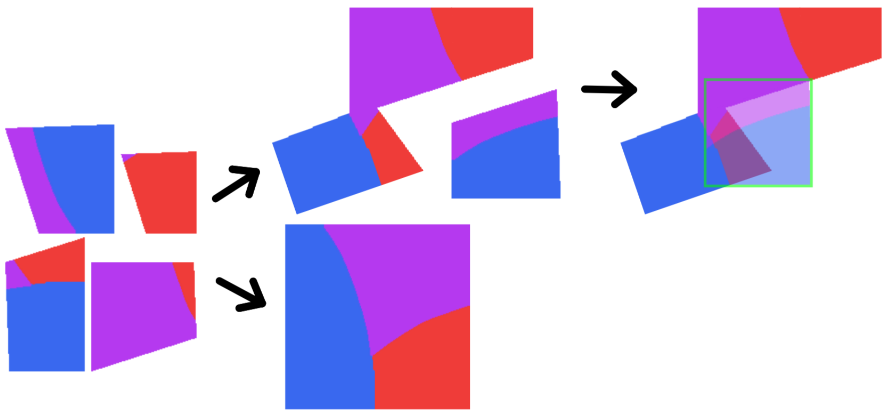


Figure 3.11: Example of Incorrect Joins Causing Reduced Piece Usage

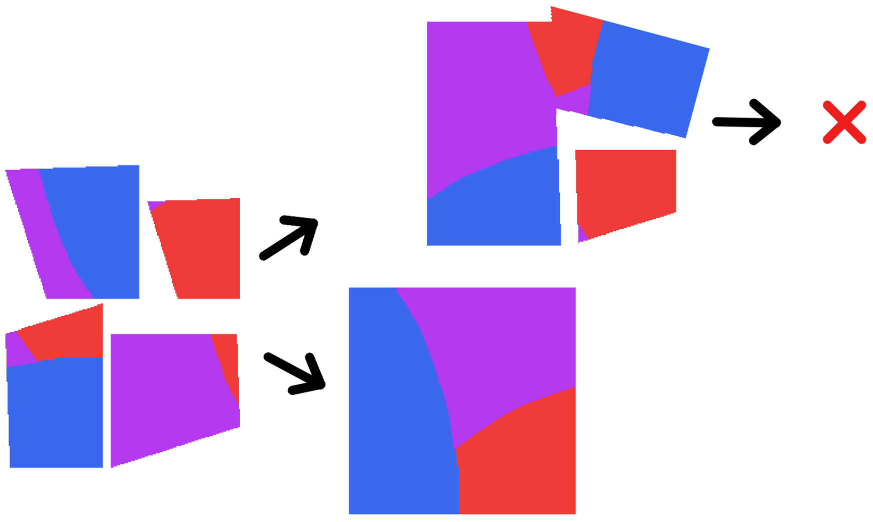
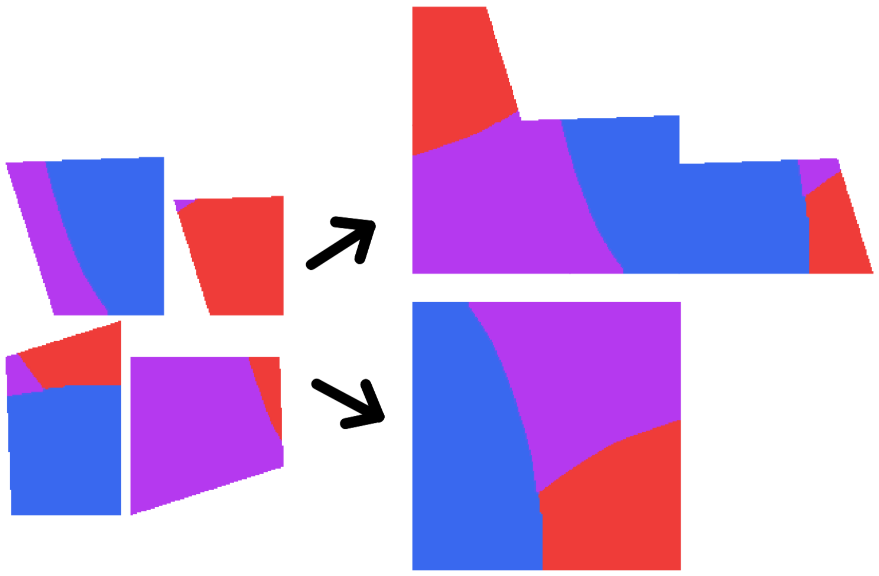


Figure 3.12: Example of Incorrect Joins Causing Wrong Shape



Turning this into an optimisation problem, we can now search for the best reconstruction. One method would be an exhaustive search of the input space to generate all of the possible permutations, and scoring each. Alternatively, we could generate n random reconstructions and again pick the best based on the metrics. This would not always find the most optimal solution, but if n is suitably large, we can probably get a good level of correctness at a far lower performance cost. The exhaustive search is far more expensive as the rectangular check is only possible at the final stage so it would require the whole image to be constructed, given a score, and would then backtrack and try other possible joins. The reconstruction algorithm will be evaluated in the next section to see if this is a viable option, but it seems sensible to assume that it is not. For this reason, the random implementation was built, with an initial n value of 15. In order to randomly choose the joins, it takes the list of joins and shuffles them. It will then pop the first element from the list, make the join, and then pop the next element off. If the join is impossible, it will move on, otherwise it will perform the join. This loops until there are no joins left, and a score and image is output. The list of joins is then shuffled again and repeated all n times. As part of the evaluation, different n values will be tested to find the most suitable. This concludes the implementation of the entire pipeline which now takes a set of image of any shape, and attempts to reconstruct the original image with no prior knowledge. It supports curved edges, nested cuts, missing pieces and extra pieces too.

Chapter 4

Evaluation

In order to evaluate the effectiveness of the image reconstruction solution, we need to define a set of success metrics. We will test the middle and last section of the pipeline separately, starting with the predictor and then the reconstructor. The first stage of the pipeline relies on basic premises and algorithms such as the Harris corner detector which have already been well evaluated, so will only be mentioned when relevant.

Focussing on the predictor, we will want to measure the accuracy. To measure this, we will chop an image up and make a note of the joins required to put it back together. We will compare it to the output of the join predictor and count the percentage of correct joins found. We will not include false positives in this calculation as we shall assume a perfect reconstructor that will eliminate these. Therefore if it successfully finds all of the joins but also has 10 extra incorrect ones, it is still 100% accurate by this metric. This will give us an idea of how well each of the characterisation algorithms work and compare against each other. Having said this, the number of false positives is still important and will be counted as more of these will lead to worse performance from the reconstructor. We can again compare the algorithms to minimise this metric. Finally, the performance of the algorithms will be compared by timing each function. The speed at which the process runs is very important as the system is meant to make it easier for humans to reassemble images. If a human could perform the task in less time, or if this application takes 10 minutes to produce an incorrect output, it will have failed at its objectives.

To test the reconstructor, we want to measure the correctness of the image output. This can easily be done by comparing it to the actual image and seeing if it matches. We can test over a set of images to see what percentage are correct using the same set of images used for the first part. For the random search, we will vary n to see how many random permutations will generally produce a correct result. This way we can find the smallest n that provides a satisfying reliability and a minimal search performance overhead. Finally we will test the overhead of a single image reconstruction, and how it varies with the number of joins. As an extension, the reconstructor will be tested with missing and extra pieces. While this was never directly implemented, all design decisions ensured that it was still possible for it to function in these scenarios.

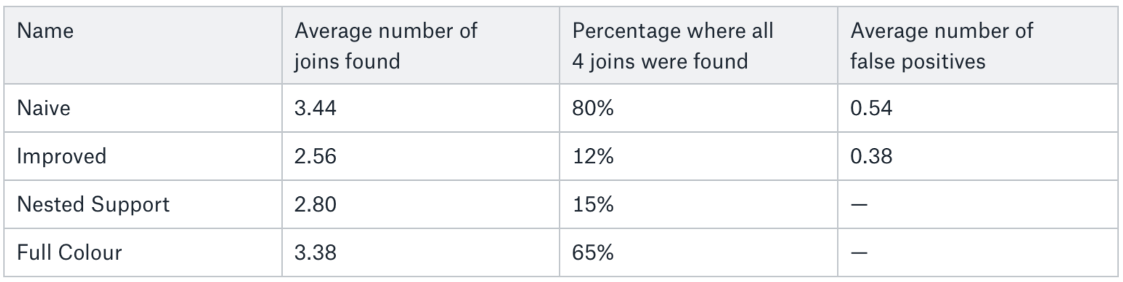
4.1 Preparation

In order to begin evaluating the program, we need to create a set of test images. For this purpose, a tool has been created to chop an image into 4 pieces, one vertical slice and one horizontal slice; both randomly generated. Although 4 pieces doesn't necessarily simulate a real scenario, this dataset will be particularly useful for comparing the algorithms to each other where their relative performance will remain unchanged. For testing the actual performance of the pipeline, more realistic examples will be used, though the algorithm works on a piece by piece basis in isolation, and if it works for 4 pieces, it should work for any number of pieces with the same level of detail just with slower performance. 4 piece slices also allows us to test many images in a reasonable amount of time. We have taken a mixed set of 100 coloured images from Unsplash [18], half being 800x600, and the other being 600x800, and chopped them up. Each set of 4 pieces is accompanied with a text file that encodes where they were sliced for comparison with the predictors.

4.2 Comparing Predictor Accuracy

The first test will be to evaluate the 4 different predictors. As the first two do not support nested partial cuts, the test dataset will not contain any. All 100 images will be put through the pipeline and the output will be searched to see if it can find the matches from the text file for each image. As the corner detector isn't perfect, we will allow ± 5 pixels on the edges it chooses to join together. We care about how well the predictor can find the correct cuts, and how many false positives it generates as this will make the reconstructor work much harder. Below is a table documenting the results.

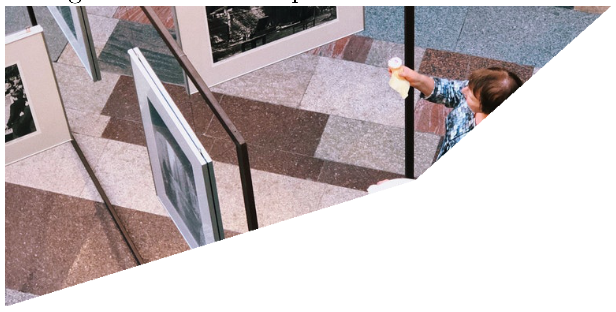
Figure 4.1: Algorithm Test Results



Name	Average number of joins found	Percentage where all 4 joins were found	Average number of false positives
Naive	3.44	80%	0.54
Improved	2.56	12%	0.38
Nested Support	2.80	15%	—
Full Colour	3.38	65%	—

The most noticeable statistic here is the initially surprising accuracy of the naive solution, and the fact it performs better than the improved version. There are a few things to break down here to make sense of it. Firstly, the naive accuracy appears to be 3.44, or 86%, but when examining the images on which it performed poorly, it became apparent that the flaw was in the corner detection. Some images had very shallow corners that the detector could not pick up, which meant edges were incorrectly defined and near impossible to match up anyway. If we remove these examples, we find that the naive solution has an even better 93.4% accuracy. Note that the improved algorithm also gets a boosted accuracy value of roughly the same figure. This makes sense as there was never a problem with the naive algorithm on full cut images. The problem was with partial cuts where it is unable to find any of these types of joins. The improved algorithm doesn't necessarily improve the characteristic function, but rather the technique used which allowed us to support more complex cuts. The naive solution can heavily rely on edge length which is a great indicator of a match, and it uses it down to the pixel level. When the bitmaps start being used, the accuracy here is lost, which explains a drop in the number of joins found with the "improved" version. There is also an edge case in the improved version where the bitmaps for the correct join can be of different sizes, where one has an extra index as the corners aren't pixel perfect so the edges aren't identical lengths. This was implicitly resolved in the nested support addition as bitmaps slide along each other, but nonetheless can cause edges to be missed. These attributes together cause it to perform worse in testing.

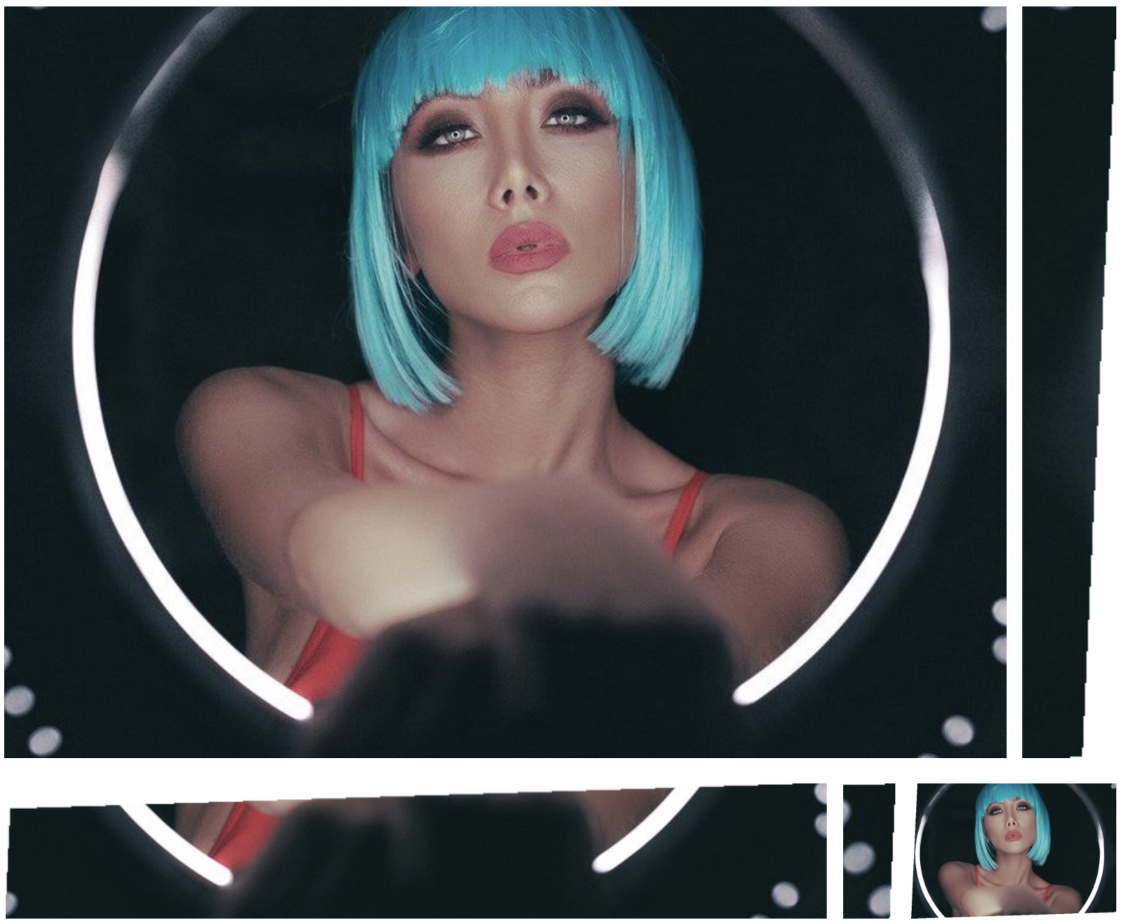
Figure 4.2: An Example Of a Shallow Corner



A particularly important metric is the number of tests that returned all of the correct joins. The reconstructor assumes that within its input space, the correct configuration exists, so all of the correct joins must be detected at this stage for any hope of success by the end. The bitmap length issue described above is very present here, as only 12% of the test images returned all of the edges. Most examples only had 2 or 3 of the 4. The naive solution, however, did particularly well and was successful for 80 of the 100 images, including those with the corner detector issue. This could be seen as a limiting factor of the reconstructor, where perhaps future work should attempt to find joins that it missed based on the ones it has; or a flawed characterisation algorithm. With such an open ended problem, and images of varying shapes, sizes and colours, perfect cut detection is likely impossible, so the former idea of finding the missing cuts within the ones you have sounds like a better future improvement. 80% is already a strong statistic, but it should be possible to get higher using different characteristics.

An interesting case was the number of false positives generated by the naive solution. Although it looks quite high with 0.54 false positives per image, they were not uniformly spread. In fact, the median number of false positives was 0. One image in particular, shown below, generated 19 false positives, while the majority had none. The reason for so many false positives is somewhat clear. The naive algorithm has a heavy focus on the length of the sides, and uses this as a strong indicator as to which pieces fit together. This image was chopped into near perfect rectangles, which meant every pieces had two sides of almost the same length. This paired with the fact that all of the edges were the same dark blue / black colour meant that the algorithm had very little to go on, thus throwing out so many predictions. It is worth noting, however, that it did still find all of the correct edges too so the reconstructor should still have been able to output the correct image.

Figure 4.3: The Rectangular Cuts that Caused 19 False Positives

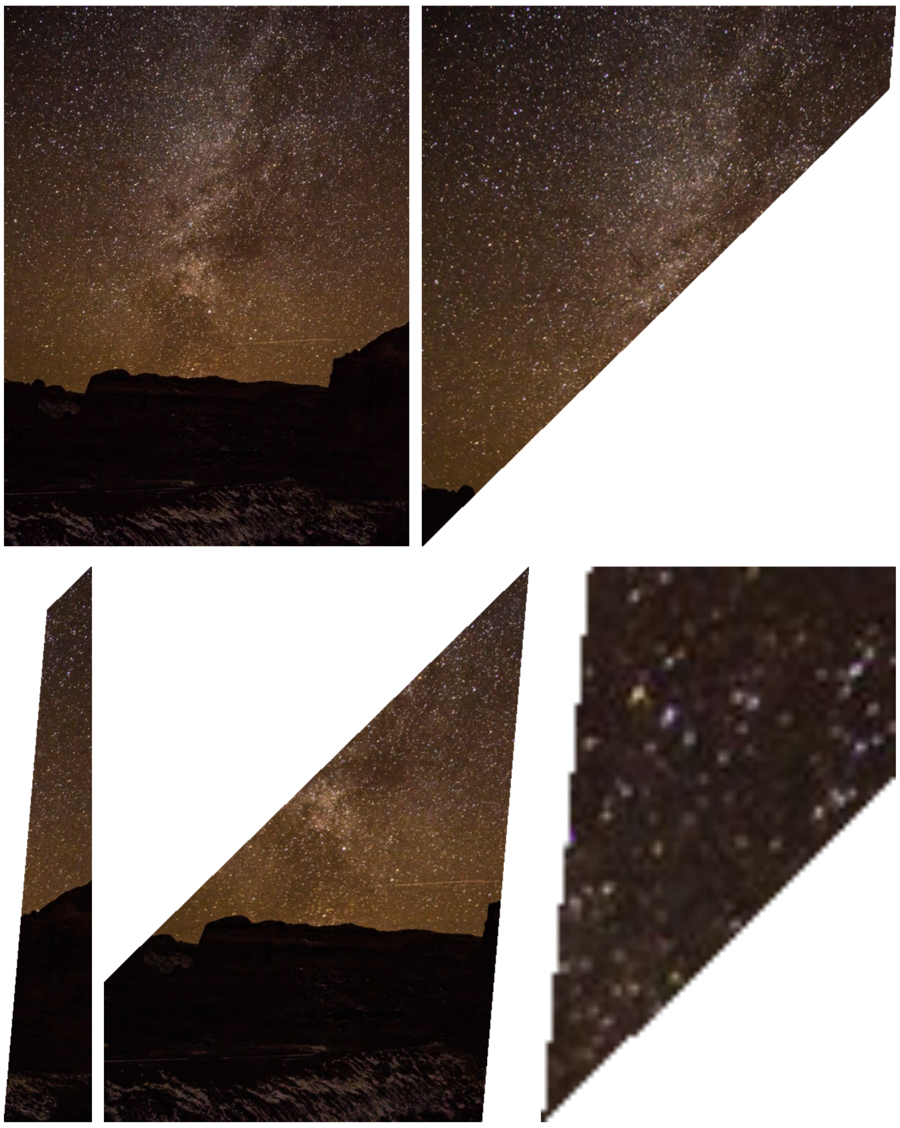


Although some algorithms produce less false positives, it should be considered alongside the number of correct joins found too. If it found less false positives but also found less true positives, this suggests that the algorithm is not particularly good at finding edges, rather than it being good at eliminating incorrect results. This is the case with the improved algorithm. The two partial-cut supporting algorithms work a little differently to the rest, and are a lot more lenient on accepted distances to ensure we capture all of the correct cuts. This comes with the major downside of a very large number of false positives. On average, for every correct join there are 15 incorrect joins. However, many of these are impossible joins where the image would have to be flipped. These will easily be eliminated later. This metric was redacted from the table as the variance was too great for the average to be useful. Some produced 10 false positives, others could produce hundreds.

You'll notice that the algorithm with nested support performs slightly better than the improved iteration. This seems unlikely as it was ran on the same batch of images, there were no partial cuts for it to improve on, and they should be based on the same distance function. As mentioned earlier, the bitmaps can sometimes be slightly different lengths due to corner inaccuracies, but now that the edges can slide along each other, those that were eliminated by the improved algorithm for being differing lengths are now considered and found. It is only a small improvement, but an improvement nonetheless. It does perform far worse when considering false positives though, generating around 15 times as many depending on the value of n for the size of the bitmap and the percentage used for the distance threshold. This is due to a larger distance threshold used as sliding the bitmaps creates more inaccuracies, thus requiring a more lenient tolerance.

Finally, the naive and full colour algorithms have very similar accuracy qualities. On the face of it, this shows that we can get the accuracy of the naive algorithm while also supporting partial cuts. The fact that no partial cuts have been used in the dataset should not affect this too much as the full colour implementation assumes every cut is a partial cut and finds all of its results through the same sliding mechanism. There are some differences between the two though. Firstly, the number of false positives is higher than any other with the full colour due to high tolerances and the sliding window. This does not affect the accuracy as the average number of joins is still similar, but will affect the performance of the reconstructor. The distribution of the number of correct edges discovered is also a little different. The naive solution was particularly good at finding all 4, but occasionally found very few, giving an average of 3.4. The full colour on the other hand almost always found 3 - 4 of the cuts, giving a similar average with a tighter distribution. This makes the latter a more consistent implementation, but due to the reconstructor requiring all 4 to put the image back together, the naive solution will produce more correct images at the end as it produces all of the cuts more often. This also isn't to say that if one performs well, then the other will automatically. There are a few examples of images that perform much better with the full colour algorithm due to the dominant colour being the same on most edges, letting the naive algorithm down. The below example was used and naive produced 3 predictions, one of which was incorrect. The full colour found all 4 correct cuts (with a lot of false positives too). You can see it is a dark coloured image with an almost entirely red-dominant set of edges. The full colour characteristic can find the light and dark parts of the edge and produce all of the edges, whereas the naive struggles.

Figure 4.4: An Example Image that Performed Best with Full Colour

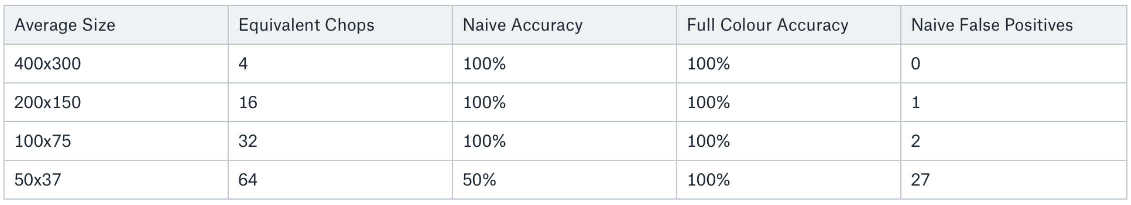


4.3 Varying the Number of Pieces

Next we will vary the number of pieces and measure the accuracy. As each image is treated as an isolated entity, more pieces alone should not make any difference to the accuracy, only its performance and number of false positives. What *will* make a difference is the level of detail reduction that often comes with more pieces. For example, if we use four 400x400 pieces or sixteen 400x400 pieces, we should see no change in accuracy. But if we take an 800x800 image and chop it into four 400x400 pieces, or sixteen 200x200 pieces, we should see an accuracy reduction as we have less pixels on each edge to analyse. Further, if we wish to test how the number of pieces affects the accuracy, we should not actually vary the number of slices we make to an image, as if we generate more cuts, and therefore more pieces, all of the pieces will be the different to the previous iteration so we introduce a new variable into our test. Rather we should take the same 4 pieces, and gradually reduce the level of detail by resizing each piece. This will emulate cutting smaller pieces from the same size image, while keeping the content in the pieces the same for a fair test.

We will take image 83 from the 100 image test set as the first test subject as the naive, nested cut and full colour algorithms all found all 4 of the cuts. Due to the results of our prior analysis, we shall be ignoring the output of the improved algorithm as it performs worse than the naive equivalent, and shall ignore the nested cut support as the full colour is superior. We will then reduce the image size by contiguous increments and plot the 3 algorithms accuracy. We will then translate the image size to the equivalent number of cuts so we can find the estimated cut performance. Note that when resizing, we must use a bilinear or nearest-neighbour algorithm to keep sharp edges. Anti-aliasing does not play nicely with the corner detector. The original image was 800x600.

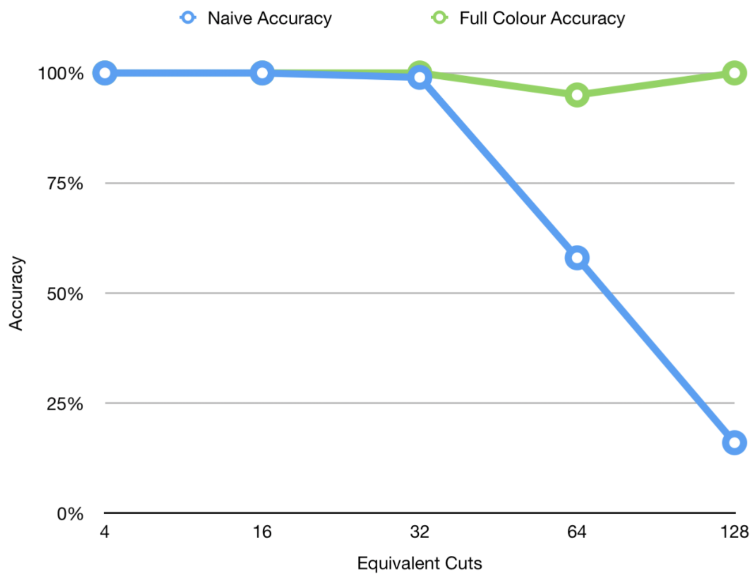
Figure 4.5: Algorithm Test Results With Variable Sizes

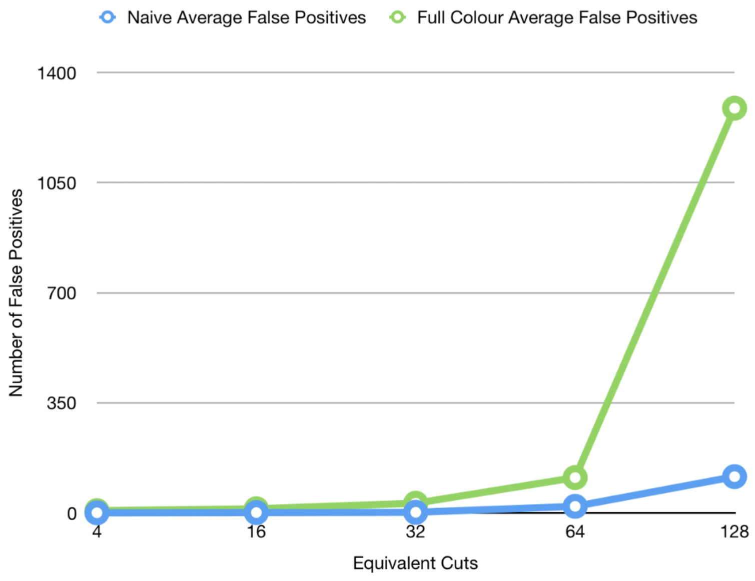


Average Size	Equivalent Chops	Naive Accuracy	Full Colour Accuracy	Naive False Positives
400x300	4	100%	100%	0
200x150	16	100%	100%	1
100x75	32	100%	100%	2
50x37	64	50%	100%	27

It appears that 64 cuts is the limit for this example. It was still successfully finding 2 of the 4 cuts with the naive algorithm, and the full colour algorithm was going strong, but the number of false positives was high for the naive, and unmanageable for the full colour (over 50). This was because the one piece was getting too small to accurately find the corners and edges due to the random nature of the slices. The top right corner became 19x26 pixels, so it was surprising that it was still managing to find the correct edges at all. To improve the quality of the results, the test was repeated on around 20 images (slightly less for some as tests were bailed on early if false positives grew too high) and the below graph was produced.

Figure 4.6: Graphs to Show Accuracy and False Positives Against Number of Pieces





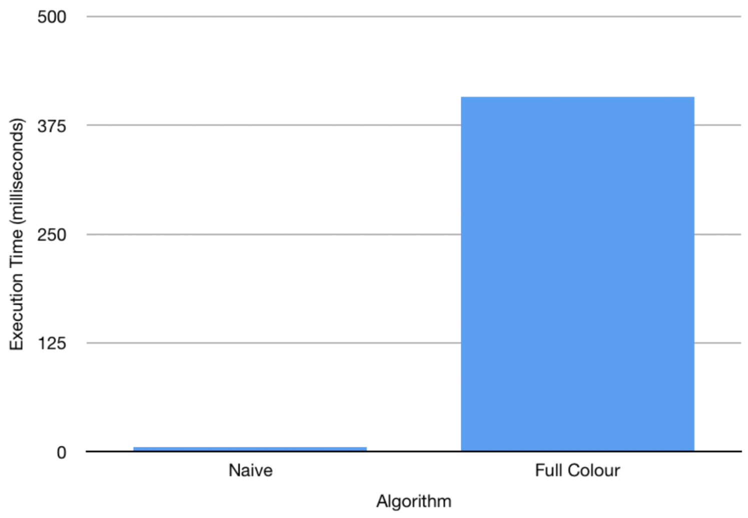
This confirms the suspicion that 64 cuts on an 800x600 image is the maximum manageable using the full colour algorithm, though 32 is a safer bet. This puts the estimated minimum piece size between 20x20 and 10x10 which is totally adequate for this problem.

Notice the dip at 64 cuts for the full colour algorithm before it goes back up to 100% on the next round. This is actually the accuracy naturally falling as it had done the previous round for the naive algorithm, but rather than carrying on with its trajectory, the entire algorithm fell apart at 128 cuts and guessed almost every possible permutation of edges, which meant the correct pieces lay in the solution set, along with an average of 1287 false positives. This is because the bitmap size was set to 5 pixels, and many of the pieces were around or even below these dimensions. This caused most of the bitmap to be empty, and when compared, it would compare an empty bitmap to another empty bitmap to give minimal distance.

4.4 Algorithm Performance

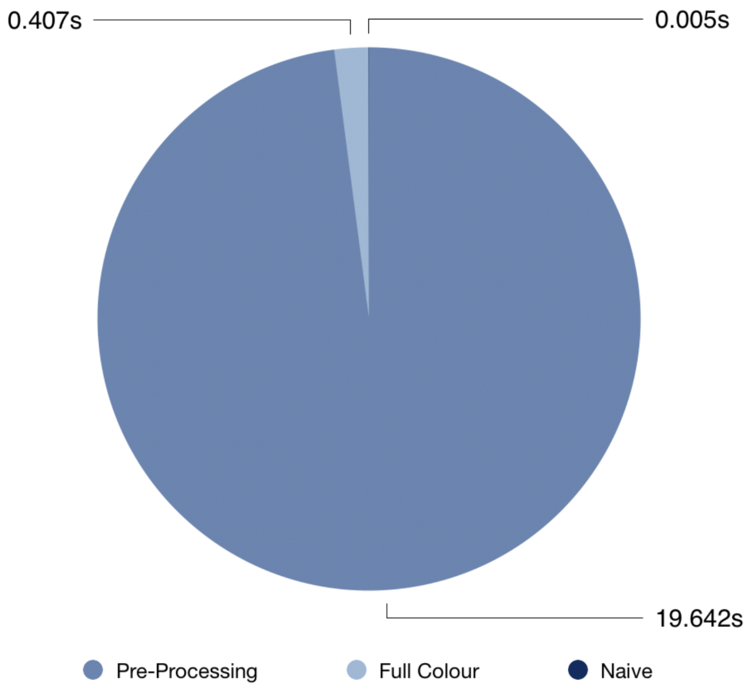
Throughout the pipeline, performance was heavily prioritised to ensure the program would run in a reasonable amount of time. We have narrowed down the two most accurate algorithms to the naive and full colour implementations, where the latter supports partial cuts, so we should benchmark both and compare their times. These experiments were ran on a 2017 MacBook Pro 15" with a 2.9 GHz (3.5GHz boost) Intel Core i7. All I/O calls were first removed from the implementations so they no longer printed to wrote to files. The timer was started when the first edge began characterisation, and stopped once the last suggestion would have been outputted. This was run on the 100 image dataset.

Figure 4.7: Algorithm Benchmarking



As expected, the full colour implementation is far more computationally expensive, by a factor of nearly 100. Full colour is the only algorithm not to represent the characteristics as individual bits, and rather than using bitwise operators, the Euclidean distance is used which involves square rooting each of the distance metrics. This is where this performance increase appears. While this may look like a concerning statistic, it is worth putting it into context. Below is the cumulative breakdown of the entire pipeline up to this point, including the pre-processing stage.

Figure 4.8: Algorithm Benchmarking in Context



The preprocessing on the 4 parts of an 800x600 image takes around 20 seconds. This includes finding the perimeter, using the corner detector, and searching for the corner connections; all of which are expensive operations that could not be optimised as easily as the predictors. Bear in mind that this scales with the number of pixels, not the number of pieces. This makes the half second processing time of the full colour algorithm almost negligible, and therefore the efforts made to keep the predictors fast can be counted as a success. The naive slice is barely visible on the graph.

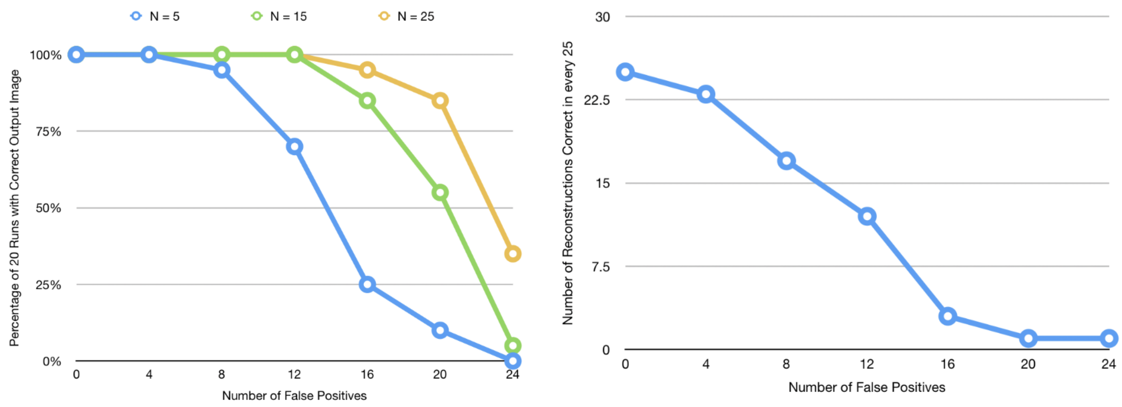
This has now fully analysed the predictor. The improved algorithm proved underwhelming, but still provided a stepping stone to the technique used with partial cuts. Unsurprisingly the full colour version outperformed its simpler counterpart, which left us just with the naive and full colour as good contenders. For images that we know have no partial cuts, the naive solution should be used as it has a better accuracy, a tiny number of false positives, and only takes a couple of milliseconds to produce an output. When partial cuts are present, the full colour join comes a very close second with similar accuracy, but often produces a lot more noise. Its performance is much more expensive, but negligible when compared to the whole pipeline, and it generally scales better to smaller image sizes.

Based on this, we can make some decisions about the implementation. Firstly, the two intermediate algorithms should be dropped. Secondly, we should choose which of the remaining two to use based on whether there are any partial cuts. Unfortunately this second point isn't possible in real scenarios, so instead we should pass the output of both predictors to two instances of the reconstructor. We will be given two output images at the end; if there were no partial cuts, then the first image should have a very high chance of being correct due to the high accuracy and little noise. The second image will likely be the same, but has a slightly lower chance of being correct. If there were partial cuts, then the first image will be incorrect, but the second will still perform well. This ensures that in either case, we give the user the best possible output.

4.5 Reconstruction Correctness

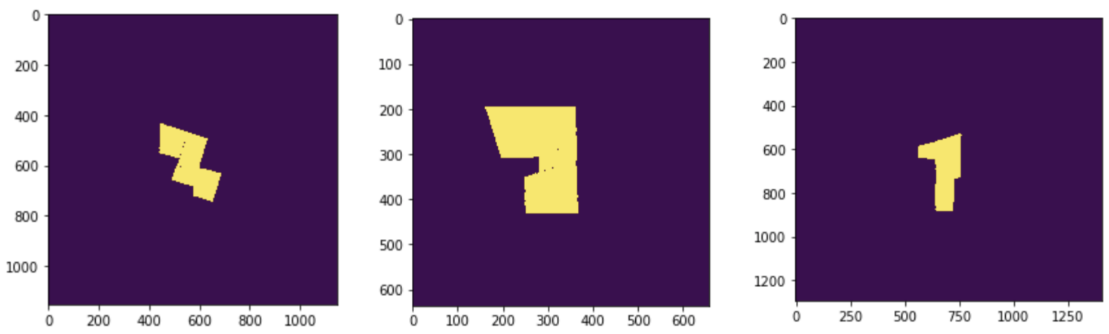
The reconstructor serves as the final part of the pipeline. It takes the list of suggestions, of which it assumes all of the correct joins are present. It will first shuffle the list randomly, and then perform the first join. It will then continue to make joins by adding one extra piece at a time. It will finish when there are no more joins to be made, either because there are no more predictions, no valid joins, or just joins between pieces that aren't part of the reconstruction. It will repeat this n times, scoring each of the outputs, and present the user with the best. To measure the correctness of the reconstructor, we should first start with a control. This will be a list of just the correct joins. Then we shall start incrementing the number of false positives to find the point that the reconstructor begins to struggle. We will start with an n value of 15, and then repeat for values of 5 and 25, and measure the performance implications of each. Each test will be repeated 20 times and the correctness will be a measure of how many produce the correct output. There is little point testing the reconstructor with a list of joins that do not contain all of the correct values as it is guaranteed to fail based on its assumptions.

Figure 4.9: Reconstructor Correctness



The first graph shows roughly the trend expected. As n gets larger, it is able to test more random permutations of the input space, and has a higher chance of finding the correct output. With the 3 indicators, the correct output is almost always chosen because the incorrect reconstructions are normally *very* incorrect. Some examples are shown below. The second graph shows a slightly different way of measuring the same data. For every 25 random permutations created from the input space, how many produce the correct reconstruction. As we add more and more false positives, the graph appears to exponentially decay, which should be expected as the number of permutations exponentially increases. By the time we get to 20 false positives, only 1 in every 25 is the correct image. This explains why on the first graph, $n = 5$ really begins to struggle at this point. It is unable to generate the correct solution at all on most of the attempts so it becomes impossible to give this as an output. Once the number of false positives increases to over 20, even the $n = 25$ version begins to falter. There are simply too many permutations to reliably produce the correct output. You can also see a pattern of diminishing returns forming as n increases linearly because it can not keep up with the exponential growth of the permutations. This confirms that the decision to not perform an exhaustive search was sensible, though perhaps a random search is too unreliable. A balance would be better, where an n value was still used, but the search was more directed. As shown in the evaluation of the previous part of the pipeline, it is not totally uncommon to have more than 25 false positives when dealing with the full colour algorithm.

Figure 4.10: Incorrect Joins are Very Incorrect



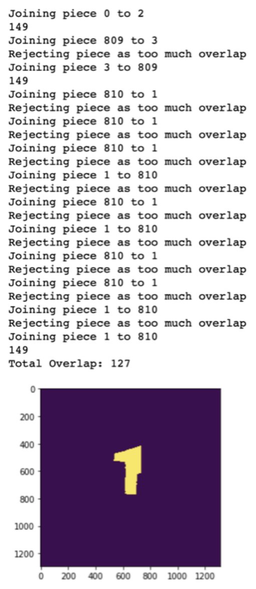
Another issue with increasing n is the performance cost. Rotating, translating and glueing together the images is incredibly expensive, and as more false positives are added, each reconstruction takes longer as it has to eliminate more joins that it discovers are impossible by the amount of overlap. Below is an example output from the reconstruction log when there were 24 false positives. Note that it tries to join the same two pieces several times but at different points along the edges, but notices that a lot of the joins are impossible before it finds the join between the two that works. Next to it is a graph measuring the performance of one reconstruction as the number of false positives increases. Again, all I/O was first removed to ensure the timer was only measuring the joining process.

Again, this graph looks roughly as expected. As the number of false positives increases, the reconstructor has to work harder. It tries more potential joins that end up being impossible, so time is wasted. Sometimes it gets lucky and manages to fully reconstruct the image with no backtracks, but other times it runs into all of the possible invalid joins. This means the variance of the time taken can be as much as $+ - 1.2s$. This is why the graph is a little bumpy. It appears to roughly grow exponentially which will be related to the exponential growth of the permutations. With all of the above data, it is fair to say that the constructor works very well when there are only a handful of false positives, and will likely pair well with the naive algorithm. However, the full colour algorithm simply produces too many false positives for them to pair well together.

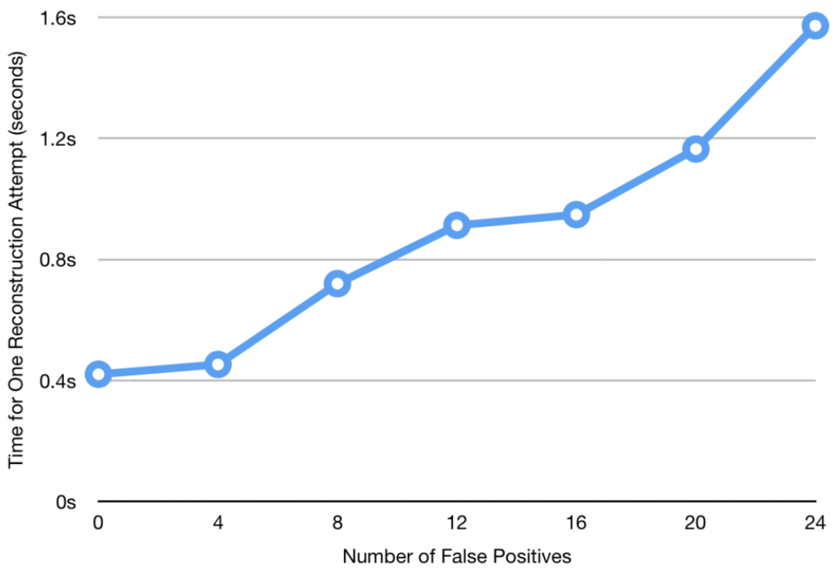
4.6 Tricking the Reconstructor

So far we have evaluated the reconstructor against perfect scenarios, where it has been given all of the pieces required to form the correct output. In a real life scenario, it may be the case that a piece is missing, or an extra piece has accidentally been mixed in. Note that if *just* the correct joins are passed to the reconstructor, it will be able to reassemble the image every time, even if pieces are missing or added, as it will always be able to join them up and only ever create one

Figure 4.11: Reconstructor Performance



```
Joining piece 0 to 2
149
Joining piece 809 to 3
Rejecting piece as too much overlap
Joining piece 3 to 809
149
Joining piece 810 to 1
Rejecting piece as too much overlap
Joining piece 810 to 1
Rejecting piece as too much overlap
Joining piece 810 to 1
Rejecting piece as too much overlap
Joining piece 1 to 810
Rejecting piece as too much overlap
Joining piece 810 to 1
Rejecting piece as too much overlap
Joining piece 1 to 810
Rejecting piece as too much overlap
Joining piece 810 to 1
Rejecting piece as too much overlap
Joining piece 810 to 1
Rejecting piece as too much overlap
Joining piece 1 to 810
Rejecting piece as too much overlap
Joining piece 1 to 810
149
Total Overlap: 127
```



permutation. For these tests we will add a random assortment of false positives such that other valid permutations become possible. Using the statistics from the previous test, 8 random false positives seems sensible to not affect the base accuracy. Starting with the missing pieces case, we introduced the "rectangular" indicator to the reconstructor as most full reassembled images will be a rectangle, and incorrect permutations will not be. This was shown to be the case in previous tests. When a piece is missing, we find that this is still mostly true. While the final image will never be a perfect rectangle, the "most rectangular" image will usually still be the correct output as it is the full rectangle with a small piece removed. There is a possible edge case where the pieces are able to form a more rectangular shape on their own, but this is unlikely. If more than one piece is removed, it is far harder to rely on this assumption. To test this hypothesis, a 6 piece image was used with an n value of 25. 1, 2 and 3 pieces were then removed as labelled below. This was repeated 20 times.

Figure 4.12: 6 Piece Image

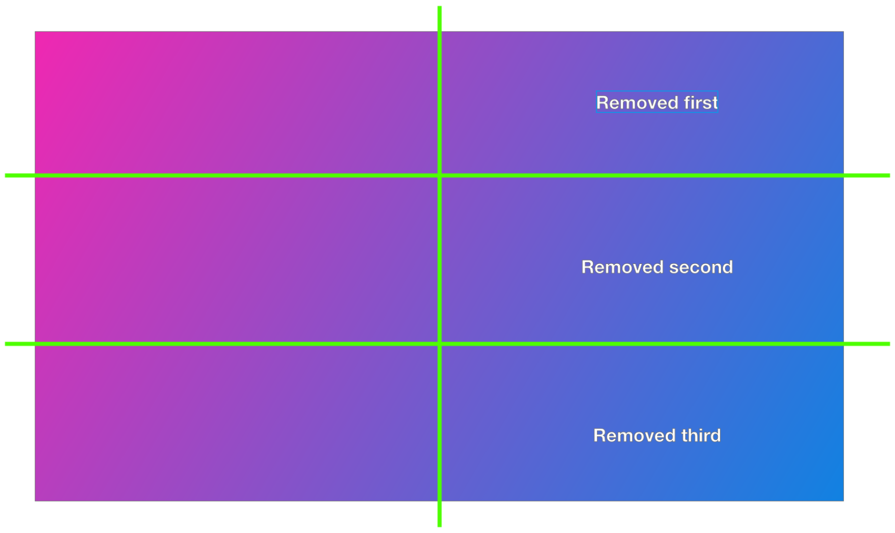
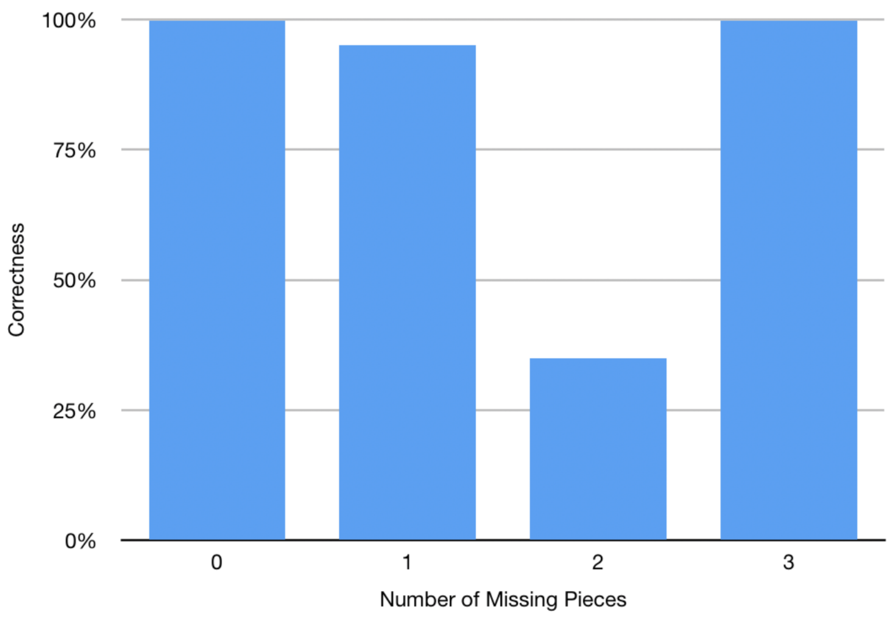


Figure 4.13: Missing Pieces vs Correctness



This produced an interesting graph. When the first piece was removed, 19 of the 20 iterations still found the correct reconstruction as it was the most rectangular image. When two were removed, however, the false positives sometimes were able to create a "better" image as it was now not very rectangular. When the third pieces was removed, we ended up creating a perfect rectangle again so the accuracy shot back up. This shows that removing more than one piece gives an inconsistent reliability. It would be fair to acknowledge support for one missing piece, but not for any more.

Finally, adding in extra pieces that aren't part of the actual image is implicitly supported. If they do not have the same colour profile, then the predictor won't find matches between them and any other piece. This means it will never get as far as the reconstructor. However, in the case that one does slip through, there are two scenarios. In the first, we assume that it is a piece that fits somewhere in the middle of the image. The overlap indicator will find that this piece fits worse than other correct pieces, and will therefore output better reconstructions to the user. In the second case, the extra piece sits on the outside of the image. This is unfortunately not handled by the current optimiser, as it will give bonus points for using an extra piece, and as it fits snugly, the overlap metric will accept it to. The final indicator is the rectangular value, which will severely decline, but this will not cause the reconstructor to reject this piece as it is built to use as many pieces as possible. Instead, it will assign this construction a worse score than other, potentially incorrect builds, causing the program to produce incorrect outputs regardless.

Chapter 5

Conclusion

As demonstrated, there is still work to be done for this to be a real threat to shredded documents. It relies on harsh assumptions requiring the pieces to be perfectly cropped with transparent backgrounds, with consistent lighting. This makes taking photos of the pieces an unlikely possibility, so they will instead need to be scanned on a closed-lid scanning bed or similar tool. This is not an unreasonable ask for the potential reward though. On the other hand, the accuracy is high so long as the image quality is too, and it scales to a large number of pieces and low resolutions if needed. Efforts were made to keep performance costs low, so images are reconstructed in a reasonable amount of time, certainly quicker than it would take a human when considering larger numbers of pieces. The major flaw in the current application is its reliance on needing all of the joins to be found in order to reconstruct the image. If this could be resolved, the real-life potential would profoundly increase. If the pieces do not include nested cuts, such as documents that have been put through a traditional shredder, then the naive algorithm will work well and pairs with the reconstructor reliably to the extent that it could well be usable in the real world, but the full colour algorithm produces too many false positives so the same cannot be said. Finally, the evidence shown towards support for missing and additional pieces is quite convincing, and again increases the real world possibilities as it's losing a piece here and there is expected.

5.1 Future Work

5.1.1 Automatically Find Missing Joins in the Reconstructor

In many cases, the predictors miss 1 of the joins, and immediately the reconstructor is unable to do its job. On reflection, it should be possible to fill in the blanks, quite literally, when putting the image back together. If the final image has a gap that can fit the final piece that was missing a join, the reconstructor should insert it automatically. If the number of pieces is large enough, it might be possible to determine more than 1 missing join. This may be harder to do if support for missing and extra pieces is still required.

5.1.2 Experiment with OCR

The current solution focuses on image based content, such as photos, but could easily be extended to text based documents too. It should be possible to find the orientation of a piece by running OCR at multiple angles and detecting the largest response. This can then be used as an extra join indicator as we know which way round the pieces fit together. If no rotation can be found, it can fall back to the algorithms currently implemented.

5.1.3 Experiment with New Join Algorithms

The current characteristic functions focus on colours and edges. This already produces a convincing accuracy, but it may be possible to improve upon it by either trying different algorithms completely or by bolting on additional indicators. A good starting point might be shape detection. If we can introduce a value that represented how "rounded" an edge is, it would be possible to join it with other edges with a similar roundness. Currently we rely on the reconstructors overlap metric to eliminate edges that don't join because of their shape, but the accuracy and performance would improve if this could be done at the predictor level. Further, the content of the entire image may also be a good indicator. Puzzles are often solved by humans by taking pieces of similar colours, grouping them together, and then solving sub-puzzles. You could also run more computer vision techniques, such as line feature detectors to find straight lines that are present in the image content. From this you can align actual edges in the content as part of the reconstruction algorithm. All of this would help in the battle against false positives which is the main flaw in the stage 2 to stage 3 pipeline connection.

Bibliography

[1] "Bayer filter," Nov 2018, accessed 01-02-2019. [Online]. Available: https://en.wikipedia.org/wiki/Bayer_filter

[2] J. Skilling and R. K. Bryan, "Maximum entropy image reconstruction - general algorithm," accessed 01-02-2019. [Online]. Available: http://adsabs.harvard.edu/full/1984MNRAS.211..111S

[3] S. Gull and G. Daniell, "Image reconstruction from incomplete and noisy data," accessed 01-02-2019. [Online]. Available: https://www.nature.com/articles/272686a0

[4] "Darpa impossible shredder challenge...solved," Jun 2018, accessed 01-02-2019. [Online]. Available: https://www.semshred.com/darpa-impossible-shredder-challenge-solved/

[5] "Team claims 50,000 for decoding shredded messages," Dec 2011, accessed 01-02-2019. [Online]. Available: http://futureoftech-discuss.nbcnews.com/_news/2011/12/02/9171208-team-claims-50000-for-decoding-shredded-messages

[6] Nayuki, "Project nayuki," accessed 01-02-2019. [Online]. Available: https://www.nayuki.io/page/image-unshredder-by-annealing

[7] L. K. R. Lactuan and J. P. Pabico, "Unshredding of shredded documents: Computational framework and implementation," *CoRR*, vol. abs/1506.07440, 2015, accessed 01-02-2019. [Online]. Available: http://arxiv.org/abs/1506.07440

[8] J. Delk, "Fbi is reconstructing shredded documents obtained during cohen raid," May 2018, accessed 01-02-2019. [Online]. Available: https://thehill.com/blogs/blog-briefing-room/389944-fbi-is-reconstructing-shredded-documents-obtained-during-cohen-raid

[9] "Srgb," Feb 2019, accessed 01-02-2019. [Online]. Available: https://en.wikipedia.org/wiki/SRGB

[10] T. Helland, "Seven grayscale conversion algorithms (with pseudocode and vb6 source code)," Jun 2012, accessed 01-02-2019. [Online]. Available: http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/

[11] S. H.L., "2d convolution in image processing," Nov 2018, accessed 01-02-2019. [Online]. Available: https://www.allaboutcircuits.com/technical-articles/two-dimensional-convolution-in-image-processing/

[12] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Gaussian smoothing," accessed 01-02-2019. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

[13] R. Wang, "Canny edge detection," accessed 01-02-2019. [Online]. Available: http://fourier.eng.hmc.edu/e161/lectures/canny/node1.html

[14] "Sobel operator," Feb 2019, accessed 01-02-2019. [Online]. Available: https://en.wikipedia.org/wiki/Sobel_operator

[15] R. Collins, "Harris corner detector," accessed 01-02-2019. [Online]. Available: http://www.cse.psu.edu/~rtc12/CSE486/lecture06.pdf

[16] "A* search algorithm," Sep 2018, accessed 01-02-2019. [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/

[17] "Hamming distance," Jun 2019, accessed 06-06-2019. [Online]. Available: \hskip\z@\relax\hskip\z@\relaxhttps://en.wikipedia.org/wiki/Hamming_distance

[18] "Unsplash," Jun 2019, accessed 06-06-2019. [Online]. Available: https://unsplash.com/license