Solving Pictorial Jigsaw Puzzles using Image Features

Jean-Malo Delignon

School of Computer Science

University of Birmingham

This dissertation is submitted for the degree of

BSc. Mathematics and Computer Science

Supervisor: Ela Claridge

April 2018

Acknowledgements

I would like to thank my Supervisor Ela Claridge for her assistance and availability throughout this year. The answers to my many questions were always spot on and helped me tremendously to achieve this project. Finally, many thanks to my family and friends for their support during this year.

Abstract

We propose a greedy solver using Image features to solve Jigsaw puzzles. This work is focused on puzzle with rectangular shaped pieces, smooth edges and unknown dimension. The compatibility measure is based on the Mahalanobis distance between gradients at edges. We reconstruct the puzzle by finding the minimum spanning tree suing Kruskal's algorithm. Our solver is capable of perfect reconstruction for puzzles of up to 300 pieces of unknown dimension.

Table of contents

Chapter 1

Introduction

The aim of the work described in this report was to provide a fully automated solver for Jigsaw puzzles assuming no prior knowledge of the puzzle. Our solver therefore works with puzzles of unknown dimension and shapeless pieces. This design choice allows our solver to accurately solve multiple images in one as we demonstrate further in section 7. Multiple difficulties arise when designing a Puzzle solver of this kind. Correctly identifying edges is one of the biggest challenges to reconstruct reliable solutions. Shape identification is not possible here as our solver is designed to work with shapeless pieces. Previous work used several heuristics such as anchor points (Cho *et al.*, 2010) in the initial seed to improve reliability and reduce complexity of the algorithm. While our solver is not capable of 100% accuracy reliability for puzzles of more than 100 pieces it is, as we will show in this report, less complex than previous solutions such as the ones presented by (Gallagher, 2012) or (Sholomon *et al.*, 2013). Another major challenge is, naturally, creating a reliable compatibility measure to match pieces. We designed our solver to be as general as possible so the compatibility measure does not assume any particular feature of the initial image.

We refer the reader to (Gonzalez and Woods, 2008) section 4 and 5 if needed to understand the fundamentals of digital Image processing. Furthermore our reconstruction is based on building an un-directed weighted graph and the reader may want to increase their understanding of Graph Theory with the section 1 of (Diestel, 2016). The papers published by (Gallagher, 2012) and (Chung *et al.*, 1998) are the two papers we based our work on. Most importantly the compatibility measure defined by (Gallagher, 2012) is the one we use throughout this report.

1.1 Report Structure

Further Background Reading

We explain terms we refer to in this report and refer the reader to D for a literature review.

Analysis and Specification

In chapter 3, we discuss the specification of this project including user and software requirements.

Design

In chapter 4, an high-level account of the structure used for our solver is given. We also explain our design choices and compare them to the ones detailed in chapter 2.

Implementation and Testing

We explore our data structure in details in chapter 5, including pseudo code and detailed low level implementation of the crucial parts of our solver. We also provide an overview of our testing strategy with a detailed summary of specific tests.

Project Management

In chapter 6, we provide a time-line of the evolution of this project and discuss the various strategies put in place to consistently meet the requirements of this project.

Results and Evaluation

In chapter 7 we present the results and evaluation of this project in relation to the requirements we set out at the beginning of this project.

Discussion

Finally in chapter 8 we discuss the accuracy of our solver, its strengths and weaknesses in the context of previously available solver. We also propose improvements to increase computational complexity and accuracy.

Chapter 2

Further Background Reading

2.1 Background Information

2.1.1 NumPy

NumPy is the fundamental package for scientific computing with Python. It is is licensed under the BSD license[1] which imposes minimal restrictions. It contains amongst other things a powerful N-dimensional array object. NumPy arrays is the data structure we use to compute the compatibility scores and perform operations on images throughout this program. NumPy arrays have homogeneous type in contrast to Python lists which even when sharing the same type are pointers to memory location. Hence NumPy arrays can benefit from locality of reference which is shown in Figure 2.1[2]. Furthermore there are many native NumPy operations that we use in our program to perform arithmetic calculations such as mean or covariance which are implemented in C thanks to a wrapper of the Automatically Tuned Linear Algebra Software (ATLAS) library[3] which takes advantage of CPU vectorization otherwise impossible natively in Python.

2.1.2 NetworkX

NetworkX is the python library that we use to manipulate graphs in this project. It is licensed under a 3-clause BSD license[4] which, similarly to NumPy's license, imposes minimal restriction. NetworkX allows the creation of graph, digraphs, multigraphs and the implementation of various algorithm such as Kruskal's Algorithm for finding the minimum Spanning Tree which we use later in this report.

[1] http://www.NumPy.org/license.html#license
[2] http://jakevdp.github.io/images/array_vs_list.png
[3] http://math-atlas.sourceforge.net/
[4] https://raw.githubusercontent.com/networkx/networkx/master/LICENSE.txt

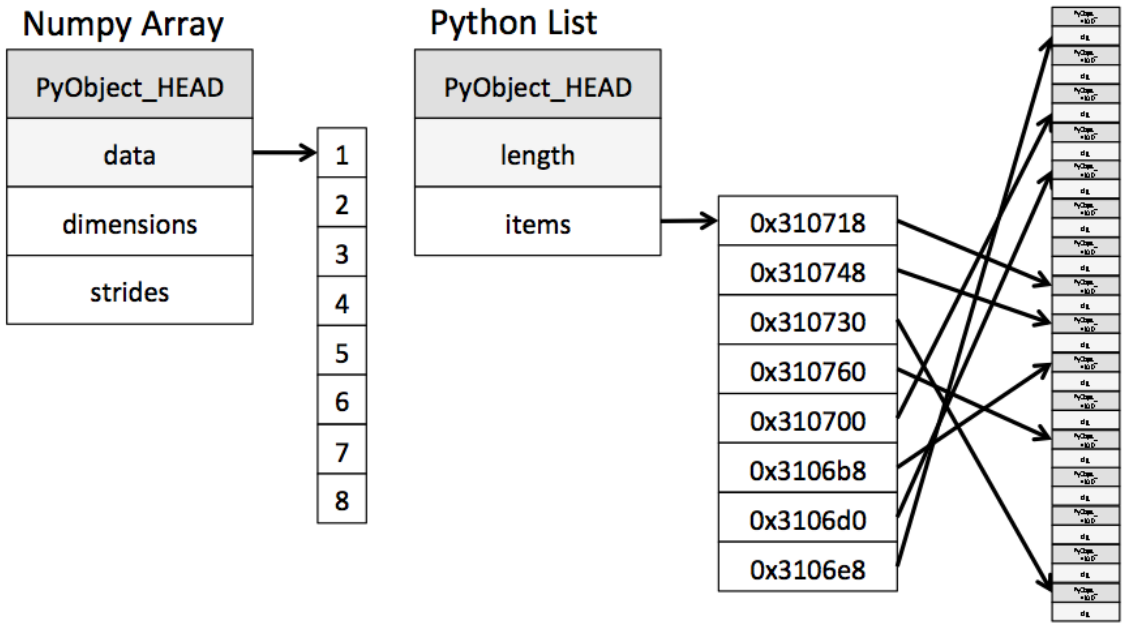

Fig. 2.1 Numpy arrays and Python lists

2.1.3 Kruskal's algorithm for finding the Minimum Spanning Tree

In chapter 5 we refer to Kruskal's algorithm for finding the Minimum Spanning Tree. We apply his algorithm to our graph to find the most optimal solution. Kruskal's algorithm works as follow.

- Consider the graph as a set of trees where each node is a subforest (at first none are connected).
- Create a set S with all the nodes in the edges of the Graph
- While S is non empty and the set of trees is non spanning:
- Find the edge E with the least weight in S and delete it
- If S connects two different trees then add it to the set of trees

2.2 Literature review

We refer the reader to appendix D for details on past implementation.

Chapter 3

Analysis and Specification

3.1 Analysis

The nature of this problem is fairly unique as we are not looking for one solution but for the perfect solution, making the complexity much bigger than a standard greedy algorithm. While a global optimum can be found quite quickly it will often results in a inaccurate solution. Therefore we need to create the most discriminate compatibility measure and a reconstructing method that utilises a few heuristic to find the perfect solution in as few steps as possible.

We divide the problem in two parts, the compatibility measure and reconstruction method. The compatibility measure aims to solve the placement problem: Given n pieces which pieces should be placed next to each other? First, it is important to note that all the pieces will not have the same number of adjacent pieces. Indeed, pieces in the corners or at the edges of the puzzle should not be place with 4 neighbours but only 2. This is why need to define a compatibility measure accurate enough to distinct pieces on the outermost of the puzzles to inner pieces. The next important part of our our solver problem is reconstructing a solution. While this may seem quite trivial at first several issue arise. To avoid creating an NP-Hard problem like the Travelling Salesman we need to define certain constraints to help the reconstruction process. We define geometric requirements which are as follows: two pieces cannot be placed at the same position and one piece can have at most four neighbours. Furthermore, should two pieces want to be placed at the same position we need to make a decision and reject certain pieces. Finally the computational complexity of the solver is considered as a crucial part of the problem. As this problem can be considered as a variation of the assignment problem, theoretically an infinite amount of iterations could yield a solution. We do not consider this way of finding a solution since we are trying to solve it in an efficient way, both in term of time and space complexity.

3.2 Requirements

3.2.1 Software requirements

We define the following software requirements:

- The solver should be able to reconstruct any type of images (Text, Monochrome, Poly-chrome etc.) of any dimension
- The solver should be able to solve puzzles in a reasonable amount of time.
- The solver should be produce a solution whether it is perfect or not for any amount of pieces.
- The solver should be able to run on any recent version of Windows, MacOS and Unix-based systems as well as multiple Harware configurations

3.2.2 User requirements

We define the following user requirements:

- This solver should be able to run without the need of third parties library installation
- Users should be able to interact with the solver via a GUI
- Users should be able to input any images and save the result via the GUI
- Users should be able to specify the number of pieces, the colour space, the compatibility measure and reconstruction method they wish to use.

Chapter 4

Design

4.1 Data flow

The data flow of this system is designed around the two major components of our solver, the compatibility measure and the reconstruction method. The figure below gives an high level overview of our system. Given any image we compute the compatibility measure, reconstruct a first solution and optimise it before displaying it for the user. While simple this data flow achieves all the necessary functions required for our system which will detail further in the next section. The output allows the user to either save the image, display it or just read the accuracy on the GUI. .

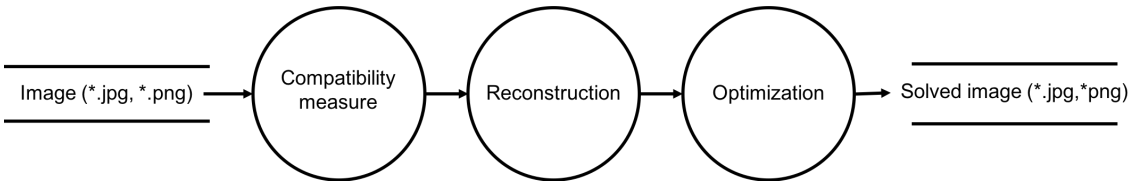

Fig. 4.1 An overview of the data flow of this system using the Gane/Sarson notation.

4.2 Compatibility Measure

We have seen in section 2 that there are currently multiple ways to design a compatibility measure to match two pieces. We decide to build ours around the Mahanalobis distance between the gradients of each edge. The Mahanalobis distance is a generalisation of the euclidean distance which takes into account the distribution of the different variables. Below is a general definition of the Mahanalobis distance (Maesschalck *et al.*, 2000) for a set of variable $\vec{x}$, another set with mean $\bar{y}$ and covariance matrix S:

$$\vec{x} = (x_1, x_2, ..., x_n)^T and \bar{y} = (y_1, y_2, ...y_n)^T$$

$$D_{Mahanalobis}(\vec{x},\vec{u}) = (\vec{x}-\vec{u})S^{-1}(\vec{x}-\vec{u})^T$$

The Mahalanobis distance accounts for the variance of each variable and the covariance between variables. This is especially useful in our case to account between color channels for each channels. If two edges are supposed to match we can reasonably expect the distribution of their color channel to be similar. In contrary if two edges should not match accounting for the co-variance of the colour channel discriminates the edges further.

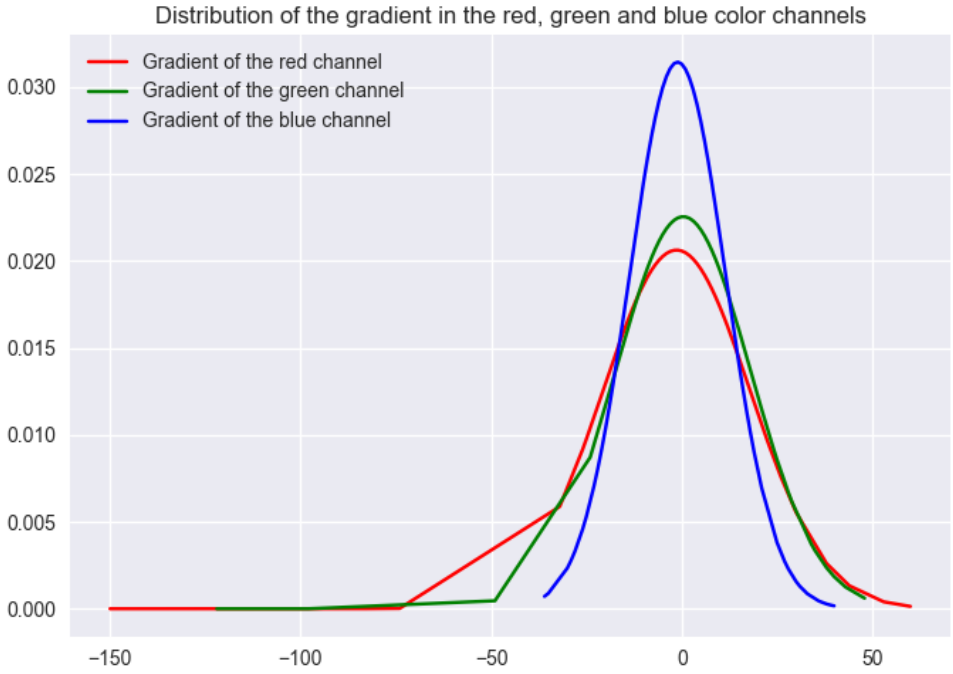

Fig. 4.2 Distribution of the gradients in the RGB colour space for one edge

We observe an average of 20% improvement when introducing the covariance in the distance calculation. Below is a comparison of using the covariance and the identity matrix when attempting to solve a 40 pieces puzzle. Effectively using the identity matrix reduces the distance measure to the Euclidean distance between the gradients.

Furthermore, similar to the SIFT algorithm (Lowe, 1999), after calculating all the scores between each pieces and edges we divide the best score by the second best. This small operation is essential to our solver as it allows our compatibility measure to discriminate pieces even more by rejecting edges whose best score is closer to 1

Fig. 4.3 Using the covariance on the left and the identity matrix on the right

4.3 Reconstruction

The reconstruction for our system is based on a graph. The graph is built from the compatibility measure with n nodes for a puzzles of n pieces. Each node can only have 4 edges and each edges are weighted with the compatibility between the connected nodes at the relevant edge. We store the edge number in the node to enforce geometric constraints and make sure that two nodes cannot be connected to a third node via the same edge. This design allows our system to determine the piece that should be on the edge of the solution very accurately.

From this graph we perform Kurskal's algorithm (Kruskal, 1956) for finding a minimum spanning Tree to find the best solution to our problem. After performing a Breadth-First Search we can begin rebuilding the solution by following the output.

4.4 User Interface

Following our dataflow design and user requirements, we build our user interface to be minimalist and efficient. The GUI gives the ability to input any files, display the solution if needed and save it. We also build a logger which has several levels of severity built in to inform the user on the progress of the solver and display error messages if need be. As we will see in section 5 we implement our GUI in Python using PyQt an open source library originally built for C. Figure 4.4 shows the GUI, which allows the user to perform any function of the requirements.

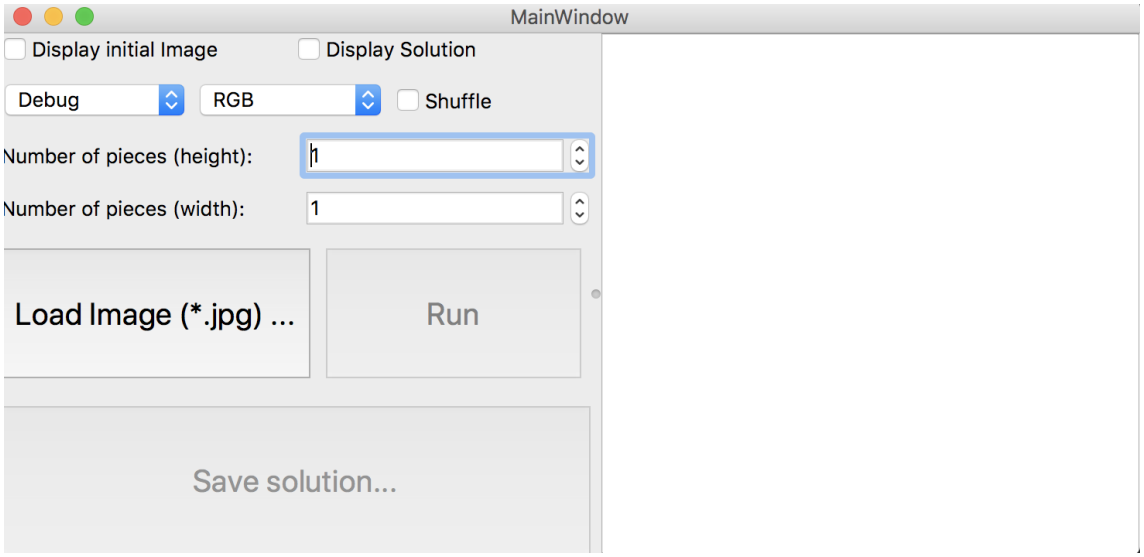

Fig. 4.4 Screen-shot of the GUI

Chapter 5

Implementation and Testing

We described in the previous section an high level view of our solver with the data flow, in this section we go into more details about the implementation of the various components in Python and the testing strategy used. Throughout this section and for clarity we consider a 3*3 puzzle which can be seen at Figure 7.2.

Fig. 5.1 The puzzle we consider for this section

5.1 Data Structure

5.1.1 NumPy arrays

The first step to our solver is to take the original image, transform it to a NumPy array and build the required number of pieces. The shape of the array is $height * width * 3$ as NumPy uses row indexing (the first dimension represent rows). For this purpose we define a function called `build_pieces` which takes an array representation of the original image, number of horizontal pieces and vertical pieces as its input and outputs a list of the pieces. Here is the pseudo code for this function:

Algorithm 1 build_pieces

```
split_height = height//vertical_piece_number
split_width = width//horizontal_piece_number
for i in range(piece_number_height) do
    x = 0
    for j in range(piece_number_width) do
        new_piece = image[y : y + split_height, x : x + split_width, :]
        append new piece to list of pieces
        x+ = split_width
    end for
    y+ = split_height
end for
shuffle list of pieces
return list of pieces
```

In our example this function outputs a shuffled list of 9 NumPy arrays which are the pieces of the puzzle. From this point forward the solver does not know how many pieces there is horizontally or vertically.

The next step is to compute the compatibility score between all the pieces and their edges, Figure 5.2 shows the positions of the edges for each pieces. As we have shown in chapter 4 the compatibility score consists of the Mahalanobis distance between each edges and their gradients. The function `get_scores` of our solver calculates the score between two edges. Figure 5.2 shows two pieces of our example, G and A which should match on edge 0 of piece A and edge 2 of piece G. Below is the pseudo code of `get_scores`, we only detail the case edge == 0 as this is relevant to our example. The code is slightly changed for edges 1,2 and 3 to match the geometric configuration.

Before calculating the distance we append the following dummy gradients [0, 0, 0], [1, 1, 1], [-1, -1, -1], [0, 0, 1], [0, 1, 0], [1, 0, 0], [-1, 0, 0], [0, -1, 0],[0, 0, -1] to the gradients of piece1 and piece2 to prevent errors when calculating the inverse of the covariance matrix. We call this function n * 4 * n times to calculate all the necesseray scores.

Algorithm 2 get_scores (piece1, piece2, edge)

if $edge == 0$ **then**
$grad\_p_1 = abs(piece_1[0,:,:] - piece_1[1,:,:])$
$grad\_p_2 = abs(piece_2[-1,:,:] - piece_2[-2,:,:])$
$grad\_p_1p_2 = abs(piece_2[-1,:,:] - piece_1[0,:,:])$
end if
$gr\_diff\_p_1\_mean = abs(grad\_p_1p_2 - mean(grad\_p_1))$
$gr\_diff\_p2\_mean = abs(grad\_p_1p_2 - mean(grad\_p_2))$
$mahalanobis\_dist\,p_1p_2 = \sqrt{\sum(gr\_diff\_p_1\_mean * cov(grad\_p_1)^{-1} * gr\_diff\_p_1\_mean^T}$

$mahalanobis\_dist\,p_2p_1 = \sqrt{\sum(gr\_diff\_p_2\_mean * cov(grad\_p_2)^{-1} * gr\_diff\_p_2\_mean^T}$

return $mahalanobis_dist\,p_1p_2 + mahalanobis_dist\,p_2p_1$

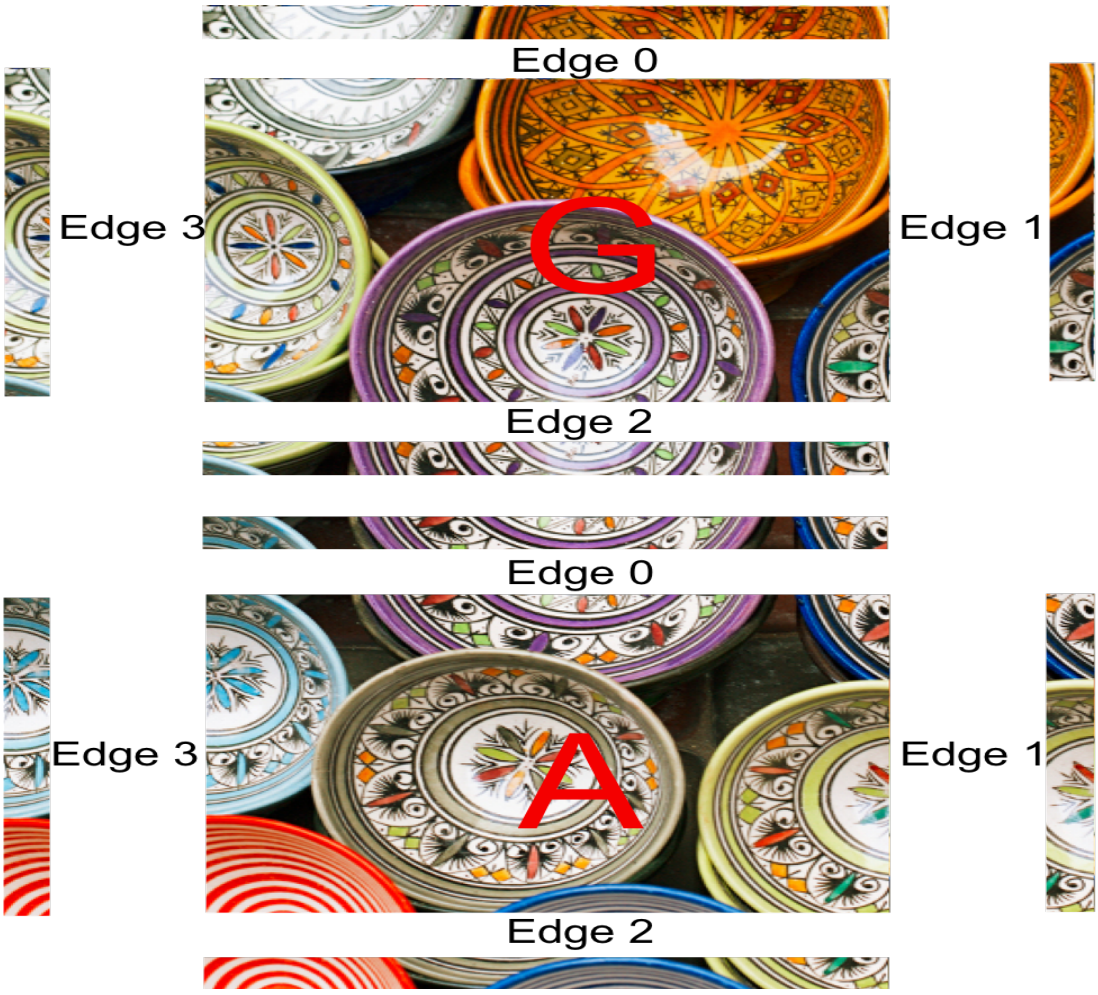

Fig. 5.2 Piece G and A with edge numbers

5.1.2 Graph

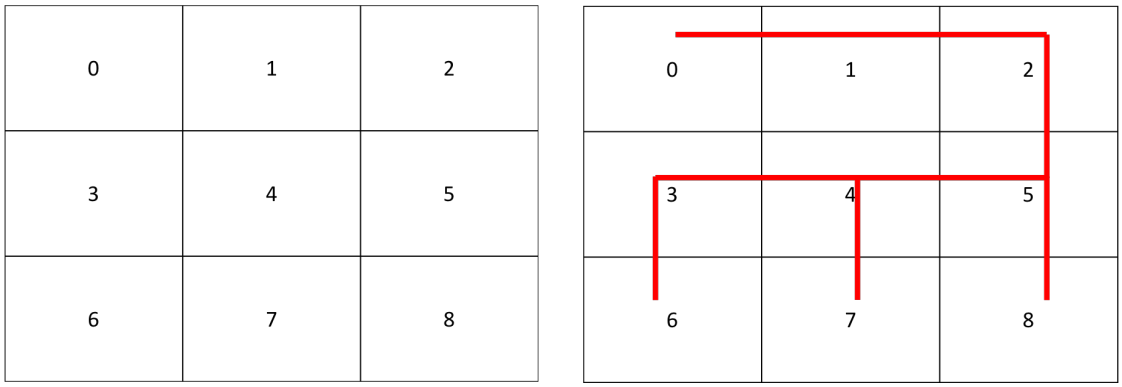

Fig. 5.3 Initial piece placement and solution of a 3 by 3 puzzle

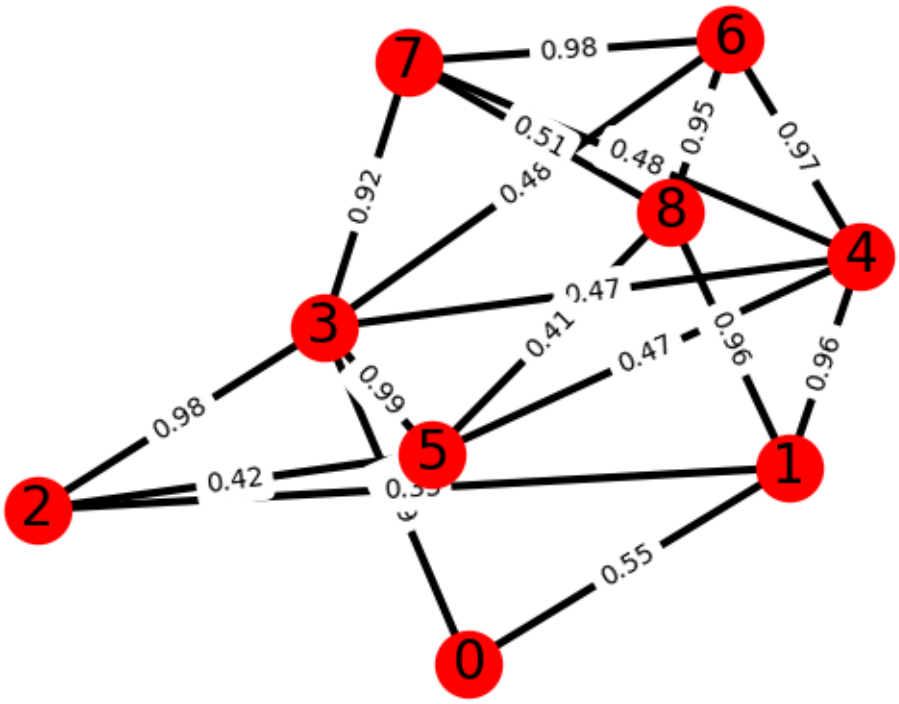

Fig. 5.4 Initial graph of the 3 by 3 puzzle

After calculating all the scores we build an undirected weighted graph. We add constraints to the graph: vertices can only have up to 4 edges and cannot be connected to the same vertex twice. In each edge we store the weight and the corresponding geometric

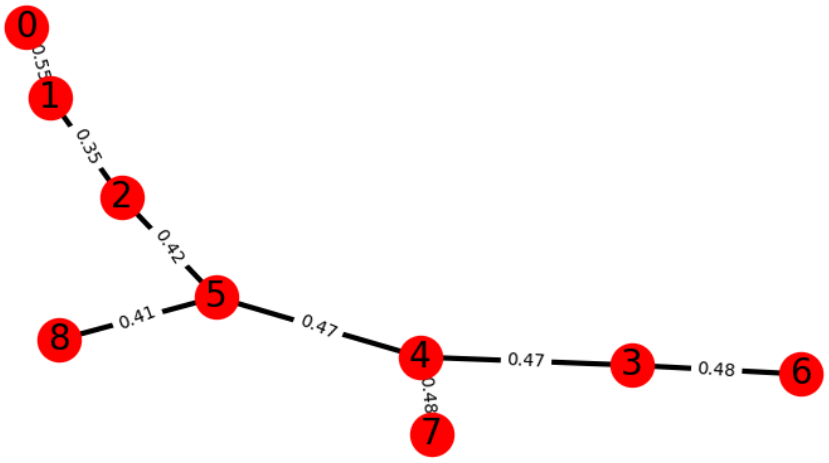

Fig. 5.5 Minimum spanning Tree of the graph

configuration. We can clearly see on Figure 5.4 the piece that should be placed on one of the edges of the puzzle. The node 1 for instance shares a good score with the node 0 (its left neighbour) and another low score with the node 2 (its right neighbour). On the contrary node 1 has a score very close to 1 with node 8 (0.96) and node 4 (0.96) which shouldn't be placed next to each others. We get rid of such edges with scores close to 1 when finding the minimum Spanning Tree.

The function `graph_create` is responsible for the creation of the graph. It takes the scores as input and outputs the graph as well as its minimum Spanning Tree.

Algorithm 3 create_graph (results)

```
for piece in range(len(results)) do
  for edge in range(4) do
    match_piece, match_score = get_best_match(piece,results,edge)
    if edge exists then
      if match_score < weight of the edge then
        add_edge(x,math_piece,match_score,edge)
      end if
    else
      add_edge(x,math_piece,match_score,edge)
    end if
  end for
end for
```

We then use NetworkX to output a minimum spanning Tree. Using this new graph that can be seen on Figure 5.5 we perform Breadth First Search to find a path in order to

output the final solution. We create a grid with width and height both equal to the number of pieces. In our example we create a 9 by 9 empty grid. The function `reconstruct` is responsible for the transition from a graph to a grid representative of the pieces positions. We use the information stored in the edges of the graph to find the positions of each pieces neighbours and iterate through the path found by Breadth First Search. We also use a helper function `insert_piece` to determine if the new piece we are inserting is outside the boundaries of the grid. In that case we shift the array in the required direction (ie if we wish to insert a piece on top of another piece that is at row 0 of the grid we shift the column down by 1). `insert_piece` also checks that the position is empty before inserting a new piece.

5.1.3 Placing the remaining pieces

Unfortunately sometimes our approach results in conflicts. This is a common problem described frequently in Puzzle assembly work ((Gallagher, 2012), (Cho *et al.*, 2010) or (Chung *et al.*, 1998)). Indeed for ambiguous images two pieces will sometimes want to be matched with the same third one. Due to the nature of our solver we cannot backtrack when encountering such a problem. A consequence is that we can be left with pieces that aren't inserted (rejected pieces). The function `insert_failure` aims to solve this problem. It takes the temporary solution as an input and for each pieces not inserted looks for the best position. Sometimes pieces need to be inserted in a black space, which is the best scenario. However sometimes we need to insert it between two pieces already placed thus creating a conflict. We only replace a piece with a failure when the neighbours at this position have a better score with the new Candidate. This ensures that we aren't creating more mistakes at this stage. Sometimes our function cannot find the best place, in that case we reject the piece and move on to the next one, hoping that the remaining pieces will be inserted and the ones left out will find a new position after that.

5.2 Testing

In order to benchmark our solver against past work we adopt two methods: Direct comparison and Neighbour comparison, both defined by (Cho *et al.*, 2010) and used in many papers to compare accuracy. Direct comparison simply looks at number of pieces that are placed correctly compared to the original image. It gives a good idea of the general accuracy of the puzzle but can be sometimes misleading as when working with puzzles of unknown entire row or columns can be shifted by a few pieces. When using direct comparison the entire row will be labelled as false while in reality most of the times the pieces are placed correctly locally. Neighbour comparison aims to solve this issue and

looks at the correct number of neighbour per pieces. An average is then calculated for all the pieces.

Algorithm 4 direct_comparison (shuffled, original_pieces, coordinates, piece_number)

$count = 0$
for $idx, valueinenumerate(coordinates)$ **do**
  if $shuffled[val] == original\_pieces[x]$ **then**
    $count += 1$
  end if
end for
return $\frac{count}{piece\_number}$

Algorithm 5 neighbour_comparison (shuffled, original_pieces, coordinates, piece_number)

$original\_coordinates = 0$
for $for(y,x), current\_pieceinnp.ndenumerate(coordinates)$ **do**
  Reset count and candidate
  Candidate is equal to the number of neighbours at (x,y)
  Find the coordinates of current_piece in the original image
  Compare each neighbour of current_piece to the ones in the solution
  If the neighbour is equal to its counterpart in the solution increment count
  Divide count by candidate and append it to the list of counters
end for
return $mean(counters)$

In our solver the functions `direct_comparison` and `neighbour_comparison` handle the testing and output their results directly in the GUI (in the console window). In `direct_comparison` and `neighbour_comparison` `shuffled` is the list of shuffled pieces, `original_pieces` is the list of unshuffled pieces and `coordinates` is the relative position of the pieces created by `reconstruct`.

Chapter 6

Project Management

6.1 Initial planning and Timeline

When I first started to plan this project I decided to adopt the waterfall development model strategy which is detailed at figure 6.1 This linear model is well suited to a project like this one for several reasons which are listed below:

- Requirements were documented and fixed.
- The goal of this project was clear and unlikely to change.
- The various technology that could be used to solve this problem were understood (ie unlikely to evolve during this project).
- There was no unclear requirements.
- The project had a set start and end date.

The purpose of this project which is to solve a jigsaw puzzle given any image was set from the beginning, the requirements which are detailed in section 3 were also straightforward as the core problem would not change and this is why I chose this model. The project proposal I wrote in September defined the problem I was solving and the various requirements. I spent the first two months researching and designing the solver in order to plan for a smooth implementation of all the features. Along with a clear plan for all the steps of this project I also wrote a time-line which spanned from September to March. I defined different tasks for each month which corresponded to various phases of the Waterfall model. The entire project proposal can be seen in appendix C along with the time-line. As we will see in the next section this approach fitted this project really well but fell short on a few points.

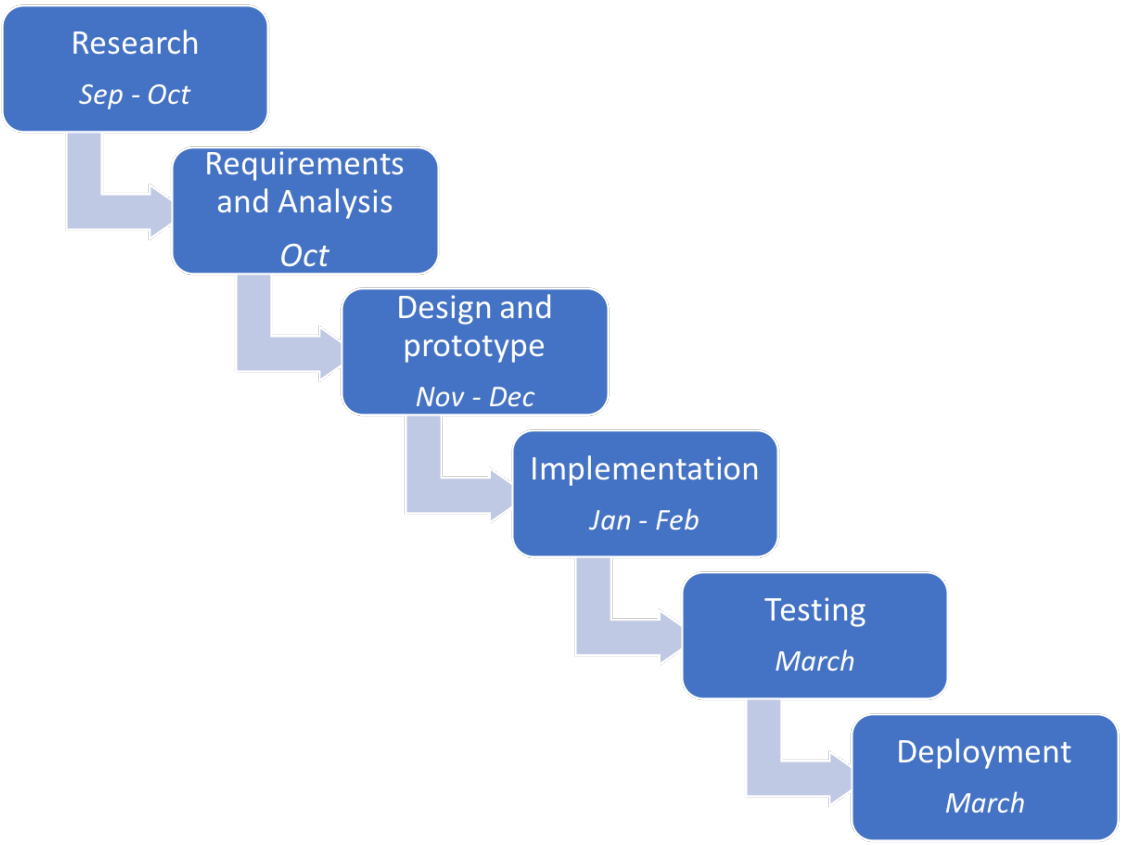

Fig. 6.1 The waterfall model for software development.

6.2 Execution

Having an high level overview of the entire project early on as well as detailed tasks for each week was essential to the success of this project. This project had a strict time constraint and spending too much time on any steps of the waterfall model can be detrimental as other phases would suffer. For instance I was tempted to dive straight into making a prototype and implementing basic features as early as September and rushing the research phase. In hindsight, researching past work relevant to this project was not only a good decision but it saved me a lot of time in the next phases of this project. I wasn't familiar with Image processing before this project and this substantial research phase allowed me to familiarise myself with common techniques in this field as well as past work. Additionally, building a prototype after the research phase allowed me to realise what technical aspect of this project was feasible before implementing all the features and get a proof of concept done early on.

However there are drawbacks to the strategy I used during this project. This model doesn't allow much modification or reflection since it is linear. The inability to deliver a working solver until late in the development process was the main disadvantage. Adding

feature becomes risky as modifying the scope late in the cycle can endanger it. In the original time-line I reserved two months for the coding of all the features. The last feature that I added was the GUI which I completed mid-march in time for the demonstration. As as result testing suffered a little and might not be as extensive as one could expect it to be due to the longer time it took to implement all the features. Furthermore it was difficult to acquire tangible data other than the result image to measure success up until the testing phase. At that point it was too late to make significant changes to the compatibility measure. This is why I built a prototype early on and wrote a thorough literature review to start the next phases in the best manner possible.

Overall I believe that choosing such a model helped the development of this project as at any time during these two semesters I knew what I was supposed to be doing next in the short and long term. This model is easy to understand and each phase is processed and executed one at a time making the development process easier. As an aside some of my personal projects end up not getting deployed due to poor planning and the rigidity of this model helped me a lot in regards to delivering this project in time. The waterfall model is designed for projects with clear requirements and this is why it fits this project so well. As early as September I was able to have a clear view of the next steps and deadline I had to met which benefited the quality of this project a lot. Aside from the assistance I received throughout this project on many subjects thanks to my supervisor, explaining and presenting my progress every week really helped me to stay on track and make sure I knew where I was. The role of this model and its time-line cannot be understated as having a rigorous high level plan for such a project is really beneficial.

Chapter 7

Results and Evaluation

In order to test the accuracy of our solution we use two different methods, the direct comparison and neighbour comparison that we defined in chapter 5. As a reminder, the direct comparison computes the fraction of correctly placed local neighbours for each pieces and computes the average. It does not take into account whether or not a given piece is at the correct position in the solution, rather if it is next to the correct pieces. The direct comparison in the contrary calculates the number of pieces that are correctly placed in comparison to the correct solution. Due to the nature of our solver it is hard to produce consistent results for a specific number of pieces and image. While it is certain that a lower number of pieces results in a higher chance of perfect recons

Figure 7.3 and Figure 7.5 show the solution of a 300 pieces puzzle consisting of two images. Our solver is capable of handling any number of images with different dimension. As we can see in Figure 7.5 the accuracy for the neighbour comparison measure is quite good. The direct comparison is a metric harder to get past 90+% as our solver sometimes fail in the final phase which is getting rid of any black space and adjusting the puzzle accordingly.

=

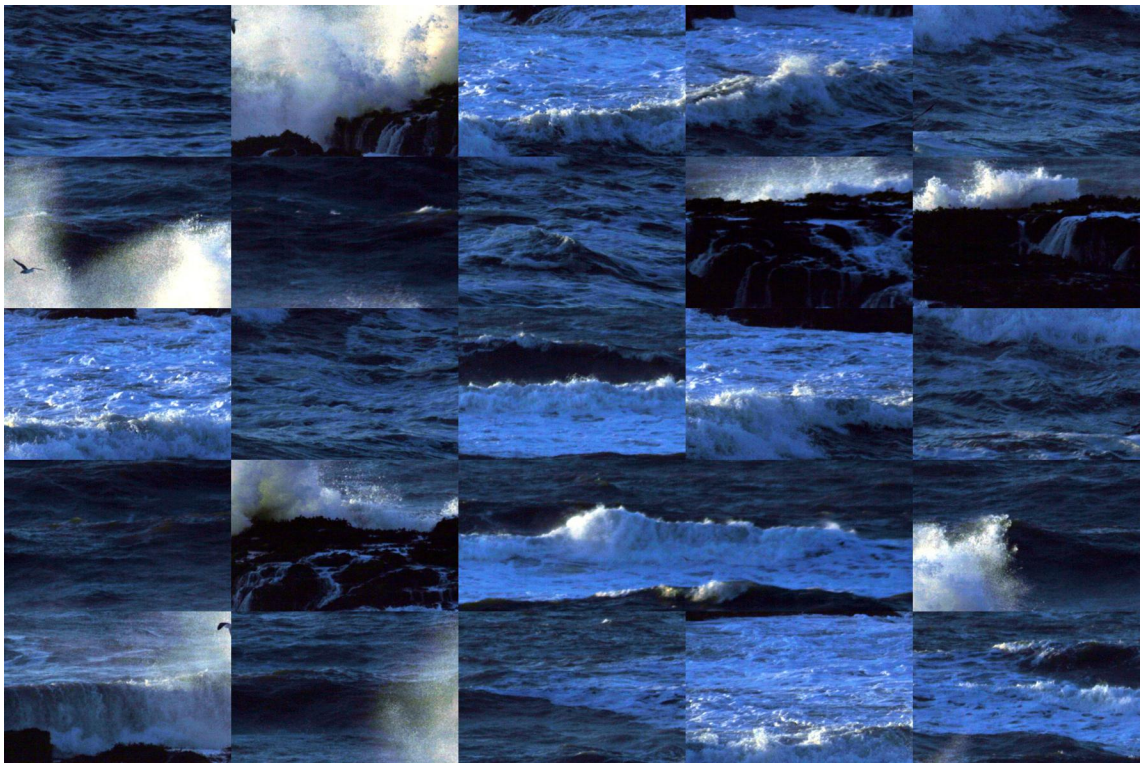

Fig. 7.1 Initial image, 25 pieces puzzle

Fig. 7.2 Solution, Neighbour comparison: 0.88 Direct comparison: 0.92

Fig. 7.3 Initial image, 300 pieces puzzle

Fig. 7.4 Solution, Neighbour comparison: 0.94 Direct comparison: 0.84

Piece Number	Direct Comparison	Neighbor Comparison
10	1	1
25	0.96	0.9
50	1	1
100	0.94	0.96
130	0.92	0.95
200	0.9	0.91
300	0.88	0.86

Fig. 7.5 Direct and Neighbour comparison for various number of pieces

Chapter 8

Discussion

8.1 Successes

The aim of this project was to provide a fully automatic Jigsaw puzzles solver for puzzles of unknown dimensions. As we have presented in chapter 7 this solver is capable of consistent results for puzzles with up to 100 pieces. The success of this project is due to the accuracy of the compatibility measure and the simple reconstructing technique.

8.2 Limitations

The main limitation of this solver is the number of pieces it can handle. Another limitation is naturally handling pieces of less than a few pixels. It is not so much the number of pieces that matters but the size of each pieces. Since the formula we use for the compatibility measures calculates the distance from each gradients of one edge to the average of all the gradients of the other edge, the fewer data points available the less accurate this average will be. This is the reason why large images can handle more pieces than smaller images. These limitations are common in puzzle solving and a problem that to this day still needs solving. Another limitation due to the design of our solver is the inability to backtrack if too many errors are made. We mentioned this briefly above but if we start with a bad node the entire puzzle can suffer. Using genetic algorithms such as the work published in (Sholomon *et al.*, 2013) solves this problem brilliantly by constantly re-evaluating the population and changing it if better solutions are found.

8.3 Improvements

As we have discussed above one of the biggest issue with this solver is the optimisation after the reconstruction. In many papers such as (Sholomon *et al.*, 2013) or (Gallagher,

2012) the author chooses an iterative approach to improve the initial solution. In (Sholomon *et al.*, 2013) the number of iterations can be really large (up to several thousands) due to the nature of genetic algorithms which is an obvious trade off of such approaches. This solver as we described it in the beginning of this report is designed to be as general as possible. It would be possible to improve it by fitting our compatibility measure to a particular subset of puzzles such as text, monochrome images or specific problems such as reconstructing banknotes or shredded documents. In many cases the problem becomes simpler and thus we can make assumptions to improve the solver. In regards to shredded documents we could keep the general structure of the solver but make a few modifications to improve the accuracy. For traditional types of shredded documents the problem would be reduced to determining the horizontal placement of pieces thus reducing the complexity and getting rid of the edge problem that we encounter for a general solver. In this configuration it becomes beneficial to regard this problem as an Assignment problem and we can solve it with algorithms such as the Hungarian method with the cost matrix being the compatibility score.

References

1. T. S. Cho, S. Avidan, W. T. Freeman, presented at the Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on, pp. 183–190.
2. M. G. Chung, M. M. Fleck, D. A. Forsyth, presented at the Signal Processing Proceedings, 1998. ICSP'98. 1998 Fourth International Conference on, vol. 2, pp. 877–880.
3. R. Diestel, *Graph Theory* (Springer-Verlag Berlin Heidelberg, ed. 3, 2016), chap. 1, ISBN: 9783662536216.
4. A. C. Gallagher, presented at the Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, pp. 382–389.
5. R. Gonzalez, R. Woods, *Digital Image Processing* (Pearson/Prentice Hall, 2008), chap. 4,5, ISBN: 9780131687288.
6. J. B. Kruskal Jr., *Proc. Amer. Math. Soc.* **7**, 48–50, ISSN: 0002-9939 (1956).
7. D. G. Lowe, presented at the Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2, pp. 1150–, ISBN: 0-7695-0164-8, (http://dl.acm.org/citation.cfm?id=850924.851523).
8. R. D. Maesschalck, D. Jouan-Rimbaud, D. Massart, *Chemometrics and Intelligent Laboratory Systems* **50**, 1–18, ISSN: 0169-7439 (2000).
9. D. Mondal, Y. Wang, S. Durocher, presented at the Computer and Robot Vision (CRV), 2013 International Conference on, pp. 249–256.
10. T. R. Nielsen, P. Drewsen, K. Hansen, *Pattern Recognition Letters* **29**, 1924–1933 (2008).
11. D. Sholomon, O. David, N. S. Netanyahu, presented at the Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on, pp. 1767–1774.

Appendix A

How to run this project

Windows OS / Mac OS

Install Python 3

1. Install python from /urlhttps://www.python.org/
2. Navigate to the zip folder and run gui.py with the virtual environment included in the fyp-jigsaw/bin/.

Appendix B

Detailed log

B.1 Semester 1

Week 1

I spent this week mostly on reading the Handbook and getting more information on the general outlines of how this project should be run. I also discussed the scope and goals of this project with my supervisor in more details to make sure we are on the same page.

Week 2

I gathered information about the different possible ways to solve this problem by reading a few papers, notably (Chung *et al.*, 1998) and (Cho *et al.*, 2010). I also wrote the first draft of my project proposal which contains the details about the scope of this project and success criteria. Finally, I wrote a rough time-line of the project which essentially divides it in two major phases: research and implementation.

Week 3

This week was spent on research and trying to summarise different papers. I took a look at (Gallagher, 2012) and (Sholomon *et al.*, 2013) which are two of the most recent papers on this subject and have very different approaches. I also finished writing my project proposal which clearly states defines the problem, the timeline and requirements of this project.

Week 4

I read and took notes on more papers, especially (Nielsen *et al.*, 2008) and (Mondal *et al.*, 2013). Most papers seem to converge towards a compatibility method which is based on

measuring the dissimilarity between pieces. I also did the peer review for a project on encryption.

Week 5

I spent this week gathering my findings and identifying papers with similar methods. This process allowed me to create a table grouping papers by the different methods used to solve this problem. I wrote an early draft of the literature review.

Week 6

I did more work on the literature review, added detailed information on compatibility measures (probabilistic, dissimilarity and genetic). I also started writing the reconstruction part of the literature review, missing a few details however.

Week 7

I finalised my literature review, it mostly needs references and a little more details on the genetic reconstruction method. I started to work on the prototype which will be based on a simple euclidean distance between gradients.

Week 8

I handed in the literature review. I worked some more on the prototype mainly defined the formula for the distance measure (euclidean) which will calculate the difference between gradients in the horizontal or vertical direction depending on the edge. I also started to look for relevant functions available in Matlab for the implementation of the prototype and final product.

Week 9

Started to implement the prototype in Matlab. It is able to shuffle any image and calculate a score for the left and right edges. The reconstruction is essntially done by using the best buddy measure defined by (Cho *et al.*, 2010) which looks for pieces with have the same best match (ie lowest compatibility score).

Week 10

Finished implementing the prototype. It now works with all edges (top, bottom, right and left). The solution cannot be displayed graphically but it works for most images for puzzles with low amount of pieces (9 to 25).

Week 11

I spent this week assessing the strength and weaknesses of the prototype and based on my findings in the literature review made some decisions on various improvements. A tree based reconstruction seems like the most effective way to solve the edge problem. Introducing the covariance between color channels would improve the accuracy of the compatibility measure by a lot as shown by (Gallagher, 2012), it needs to be added to the final solution.

B.2 Semester 2

Week 1

My Christmas break was spent defining a clear design strategy including data structure and details on compatibility and reconstruction which will use Kuskal's algorithm to find a minimum spanning Tree. I spent this week most of this week in Matlab by improving the prototype and implementing the new compatibility algorithm.

Week 2

I have made the decision to switch from Matlab to Python for this project. This choice is motivated by two reasons, first is that future opportunities outside this project have asked me to improve my python skills, second is that I feel more comfortable building this project in Python. My python skills are more developed and it will allow me to save time as I dive more into the implementation part of this project.

Week 3

This week was spent on implementing the Mahalanobis distance in the compatibility algorithm in Python using Numpy. The solver can now shuffle an image, compute scores and display a solution for basic puzzles. There is still a lot of work to be done in the reconstruction part.

Week 4

Similarly to last week, I spent this one working on the implementation and solving a bug in the reconstruction function of my solver. Using NetworkX the solver can now create a Graph and compute its minimum spanning Tree. While the Tree looks promising I still need to interpret the geometric configuration of the output and display it.

Week 5

The solver can now output a solution for puzzles of any number of pieces. While the solution isn't entirely accurate it does work most of the time for smaller puzzles.

Week 6

I improved the overall complexity of the program by removing unnecessary nested loops and taking advantage of vectorization in Numpy.

Week 7

Week 8

I spent this week testing various to optimize the solution. Unfortunately most methods fail due to the random nature of the solution (sometimes all pieces are places but some in the wrong position, sometimes the dimensions are wrong, sometimes pieces need to be inserted and the solution shifted etc.)

Week 9

Created a GUI for the solver using PyQT, a C wrapper for GUIs in Python. It is quite basic but performs all the functions of our solver. No more features will be implemented, as we get closer to the demo my work will focus on testing and writing the report.

Week 10

Created a Test class to measure the accuracy of solutions. It supports the Mean square error, structural similarity and neighbor comparison defined by (Chung *et al.*, 1998). The test class is also able to display a graph of the accuracy of the solution with the various testing methods for any number of tests. I also started to work on the report, writing the Further readings and Design sections.

Week 11

I finished running tests on various images and number of pieces to include in the final report. The rest of the time before the 9th April deadline will be focused exclusively to writing the report.

Appendix C

Project proposal

Project proposal

Jean-Malo Delignon - 1581938

Overview

This project aims to design a program capable of solving jigsaw puzzles using image recognition. The pieces will be blocks of the same size and form allowing for the compatibility algorithm to focus on the features of each pieces rather than their shape. The research will focus on improving the success rate of the algorithm as well as its complexity. The orientation of each pieces will be fixed, hence this project will use "Type 1" (Gallagher, 2012) puzzles.

Methodology

Timeline

October

- Read and summarize relevant scientific papers to determine advantages and draw backs of existing methods
- Write literature review

November

- Implement prototype in Java using a basic compatibility and reconstruction algorithm
- Use findings of literature review to establish the approach that will be chosen for this project detailing the maths and pseudo code

December – January

- Implement improved compatibility algorithm and reconstruction method in Java
- If implementation is done early, work on an approach for "Type 2" puzzles. "Type 2" puzzles unlike "Type 1" have unknown pieces orientation, multiplying the number of possible solutions by a factor of 4^k for a k-pieces puzzle (Gallagher, 2012).

February

- Polish program: UI, automated testing
- Evaluation: calculate running time and success rate using various datasets
- Prepare for the presentation and start writing report

March

- Finish writing report

Success criteria

The success of this project will be measured in two ways. First by comparing the result to the original image and calculating a success rate from 0 (no resemblance to the original image) to 1 (perfect match), second with the running time and complexity of the algorithm. Results will be compared to prior implementation of the program and existing designs detailed in the literature review.

Resources

Software and hardware

This project will be implemented in Java 1.8 with IntelliJ IDEA on windows 64bits. Basic Java libraries may be used but no third party libraries. The hardware used will be a GTX 1080, Intel core i5 8400 and 16GB of DDR4 ram.

Other

The work of past papers may be used in order to design parts of the algorithm and the datasets will come from various sources, credit will be given in the final report.

References

Gallagher, A. C., 2012. *Jigsaw puzzles with pieces of unknown orientation,* s.l.: CVPR.

Appendix D

Literature review

Literature review

Jean-Malo Delignon – 1581938

Contents

Overview

The first attempt at a computational approach to solving jigsaw puzzle was published by Freeman and Gardner [1] in 1964. Their work paved the way for researches to explore different methods as image recognition improved over the years. While a lot of the work in this area attempts to solve this problem using pattern matching on the geometric shapes of each pieces, this project tackles the problem using shapeless pieces and focuses on the features of each piece. We will look at different papers and try to define advantages and drawback of each methods.

Compatibility

The first step to solving jigsaw puzzles is to define a compatibility method to match each piece with its right neighbour.

Sample choice and colour space

Most papers agree that a strip of pixels at the edge of each piece is the best choice. Indeed, choosing the entire piece is highly inefficient as it introduces noise in the data, the edge being the most relevant part to find a correct match. Chung [2] proposes to use circles at regular interval along the edges. This method proved to be less efficient, but it can be superior when working with real jigsaw puzzles who often have inaccurate edges due to wear or build quality. With this method, circles can be larger than a strip while introducing a reasonable amount of data if the interval is appropriate.

Another point of convergence between most researchers is the colour space. Gallagher [3], Pomeranz [4] and Chung [2] chose the LAB colour space, arguably the most discriminative one. The HSI colour space outperforms the LAB colour space when combined with a Sobel filter and weighted with geometric matching [5]. Overall, the LAB colour space provides the best results as proven by Chung [2].

While colour space and choice of sample can have an impact, a significant difference in success rate appears with the choice of a compatibility method.

Distance-based compatibility

This approach, perhaps the most intuitive, computes the dissimilarity between two edges by using a formal distance measure. Chung [2] introduces it first in 1998 with a function called HistoDiff. This function computes the absolute difference between each colour channel and averages the results to create an average for each edge. Chung managed to solve puzzles of up to 54 pieces [2] by weighting this approach with the geometric features of each pieces.

Expanding on Chung's work Cho [6] develops his own formula, the squared difference of a $K\ x\ K\ x\ 3$ matrices. Let x_j be the left piece and x_i the right piece, then we have [2] [4]

$$D_{LR}(x_j\ , x_i) = (\sum_{k=1}^{K}\sum_{l=1}^{3}(\left|x_j(k,K,l) - \ x_i(k,1,l)\right|)\text{^}p)\text{^}(q/p)$$

Chung used $p\ =\ 2$ and $q\ =\ 2$ but after some tests Pomeranz [4] found that $q\ = \frac{1}{16}$ and $p\ = \frac{3}{10}$ allowed for an increase in the success rate by up to 7% [4]. This method is however inaccurate with complex patterns as it assumes pieces should be matched when their edges share the most similar colour. This is false when a change in colour occurs near the edge of a piece and therefore can create false negatives.

To solve the drawbacks of this method Gallagher [3] computes the covariance between colour channels and uses the Mahalanobis distance. Therefore, gradient intensity is penalized rather than intensity.

$$D_{LR}(x_i, x_j) = \sum_{p=1}^{P}(G_{ijLR}(p) - \ \mu_{\mathrm{iL}})S_{iL}^{-1}(G_{ijLR}(p) - \ \mu_{iL})^T$$

Where $G_{ijLR}(p)$ is the gradient from the right side of piece x_i to the left side of piece x_j , at row position p, $S_{iL}^{-1}$is the 3×3 covariance matrix and $\mu_{iL}$is the gradient's mean distribution on the left side. The final compatibility score is computed by adding $D_{LR}(x_i, x_j) + D_{RL}(x_i, x_j)$ [3].

Prediction-based compatibility

Following on the observation that pixels next to each others are likely to share similarities we can estimate what the pixel after an edge should be. Pomeranz [4] considers the difference between the last two pixels and use first order Taylor's expansions to find an estimate for the next pixel. The method is used twice, in a symmetric fashion [4] to make sure the match found the first time is a good one.

$$Pred(x_j, x_i, r) = \sum_{k=1}^{K}\sum_{d=1}^{3}[([2x_i(k,K,d) - x_i(k,K-1,d)] - x_j(k,1,d))^{\frac{3}{10}} + ([2x_j(k,1,d) - x_j(k,2,d)] - x_i(k,K,d))^{\frac{3}{10}}]^{\frac{5}{24}}$$

Paikin [7] proposes an improvement of this method which argues that using the L_1 norm performs better when working with a puzzle with missing pieces.

Reconstruction

When working with real puzzle pieces the border can be found easily using a heuristic for the assignment problem. Chung [2], Goldberg [8] and Nielsen [5] divide their reconstruction in two steps, the border and the interior. The border is checked using geometrical features and corner symmetry to make sure the solution is perfect. This method reduces the complexity as the solver for the interior knows the dimension of the puzzle and can rely on the perfect placement of the border pieces.

Greedy approach

Defined by Pomeranz [4] the best buddy metric [4] states that two pieces two pieces are said to be best buddy if their compatibility score is higher than that of any other pieces available.

Pomeranz [4] divide the problem in 3 subproblems. First the best buddy metric is used to match pieces together. While this metric is good to match pieces locally it often leads to errors when reconstructing a large puzzle as it does not consider global arrangements or true locations [4]. This issue is solved with an algorithm to match segments defined in [9]. The largest segment is then selected as the new seed and iterates once again until the best buddy metric has find a local maximum [4].

Since Nielsen [5] can rely on a perfect border an obvious approach is to find empty pieces with two adjacent pieces and use the result of the compatibility score to find the best match. Starting from corners the puzzle can filled one piece at a time.

Search tree

A minimal spanning tree is used by Gallagher [3] and revealed quite efficient. Trimming is used to ensure that if the tree does not result in a rectangular shape the excess pieces are returned to the forest, finally if the tree has empty pieces the available piece with the highest compatibility score is chosen.

Genetic algorithm

This implementation of the reconstruction is proposed by Sholomon [10]. Instead of using a compatibility function first and then reconstructing the puzzle this method generates 1000 random reconstructions, evaluates each of them with a fitness function based off Cho's work [6] and a new

population is created using the best parents. When creating a child, each piece is added one at time using a crossover function. First all pieces that parents agree on are place at the boundary of the current reconstruction. Then the best buddy system defined by Pomeranz et al. [4] is used to place remaining pieces. Finally, a random boundary is chosen, and the best available piece is placed. While this method performs similarly to most approaches, it adds unnecessary complexity. Indeed, we are looking for the best solution not "a solution" and this algorithm will take longer than most greedy approaches to reach a perfect solution. It can be beneficial for large puzzles which is the case here, tested on 22,834 pieces this method reached a success rate of 91.74 % [10].

Conclusion

We have looked at the different approaches to solving jigsaw puzzles studied over the years and established the drawbacks and advantages of each methods. In the context of our work it appears that choosing a dissimilarity based compatibility method with a greedy reconstruction is the best choice. However, it remains to see if a weighted approach combining various compatibility methods could outperform previous work.

Bibliography [10]

[1] H. . Freeman and L. . Garder, "Apictorial Jigsaw Puzzles: The Computer Solution of a Problem in Pattern Recognition," *IEEE Transactions on Electronic Computers,* vol. 13, no. 2, pp. 118-127, 1964.

[2] M. G. Chung, M. M. Fleck and D. A. Forsyth, "Jigsaw puzzle solver using shape and color," , 1998. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?isnumber=16697&arnumber=770751&count=213&index=6. [Accessed 8 11 2017].

[3] A. C. Gallagher, "Jigsaw puzzles with pieces of unknown orientation," CVPR, 2012.

[4] D. . Pomeranz, M. . Shemesh and O. . Ben-Shahar, "A fully automated greedy square jigsaw puzzle solver," , 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/cvpr/cvpr2011.html. [Accessed 8 11 2017].

[5] T. R. Nielsen, P. . Drewsen and K. . Hansen, "Solving jigsaw puzzles using image features," *Pattern Recognition Letters,* vol. 29, no. 14, pp. 1924-1933, 2008.

[6] T. A. S. a. F. Cho, "A probabilistic image jigsaw puzzle solver," in *Computer Vision and Pattern Recognition (CVPR) IEEE Conference on (pp. 183-190)*, June 2010 .

[7] G. a. T. A. Paikin, "Solving multiple square jigsaw puzzles with missing pieces.," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[8] D. . Goldberg, C. . Malon and M. W. Bern, "A global approach to automatic solution of jigsaw puzzles," *Computational Geometry: Theory and Applications,* vol. 28, no. , pp. 165-174, 2002.

[9] D. . Roman, M. . Fisher and J. . Cubillo, "Digital image processing-an object-oriented approach," *IEEE Transactions on Education,* vol. 41, no. 4, pp. 331-333, 1998.

[10] D. D. O. a. N. N. Sholomon, “A genetic algorithm-based solver for very large jigsaw puzzles,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 1767-1774).*, 2013.