Image Processing Miniproject

Peter Plass Jensen, ROB3 AAU

November 25, 2022

1 Introduction

This report will present a program that automatically calculates the score value of any given setup from the board game King Domino. In King Domino each player is the lord of their own kingdom. Each player takes turns to select domino-like tiles with two terrain sections to add to their kingdom. Some terrain sections have crowns, which contribute to the final score of the board. The goal is to get as many matching terrains to be arranged in series, adjacent to one another, as possible. The final score is then calculated by taking the total amount of same-terrain adjacent tiles and multiplying that by the number of crowns present in that terrain. Repeat for each area in your kingdom and add up the numbers. The player with the highest scoring board wins.

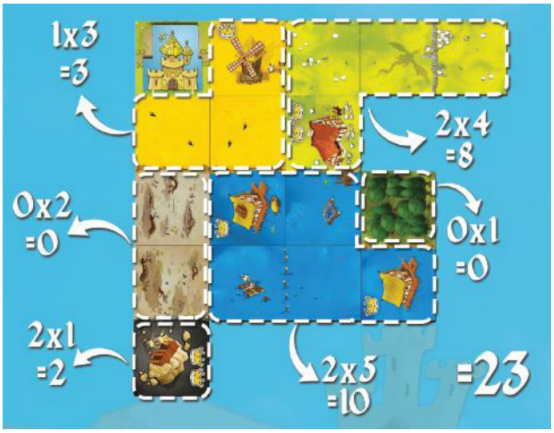


Figure 1: An example of how the scores of a King Domino board is calculated

2 Limitations and Framework

The data-set supplied for this project consists of 74 cropped and perspective corrected images of 5x5 King Domino boards. This program is only able to handle these, and not any full game areas. The program is written in Python and uses OpenCV, an open source computer vision library, Numpy, a mathematics library for python, Imutils, a library with a number of image utility functions as well as deques from the Collections library.

3 The General Approach

The data-set has been split into a training set and a test set with a 80-20 distribution. This means that 80% of the data set will be used to train and develop the program and the remaining 20% will be used to test its performance. The program first imports the board and images of different crowns for template matching. The board is sliced into 25 squares, that each represent one section of terrain and the slices are saved in a nested list. The "returnCoL" function is then called each time a slice is generated, which evaluates the image color components and returns an expected color character to

an array. At the same time, the expected terrain type is drawn on the image on the corresponding terrain area. When all the terrain types have been evaluated, the "drawCrown" function is called, which in turn calls the "detectCrown" function. The detectCrown function uses template matching to find possible matches in the image, and saves all the matches as coordinates for bounding boxes in a list called boxes. The drawCrowns function then runs non max suppression on these bounding box coordinates, selects the best fit and draws that bounding box on the image. The function also saves the amount of crowns detected in an array for the total point calculation. The program has now found all tile types on the board and detected all the crowns. It then runs the retCmap function on the array that stores the representative characters of tile types, and isolates each type of terrain for grassfire. When grassfire is done for one terrain type, the score for that terrain is calculated and added to the score of the board. This is repeated for all terrain types and the score is written on the board.

3.1 Detecting Terrain Types

The program initially slices the board into 25 squares, so that each slice represents one terrain tile. The slices are then converted into the HSV colorspace, and the mean values for hue, saturation and value are extracted from the slice. The RGB values are also extracted and these values are then used to determine the terrain type. The benefit of using the HSV colorspace is that it can be easier to numerically isolate certain colors and is a very common practice in computer vision and image processing. HSV and RGB are just two of many different colorspaces.

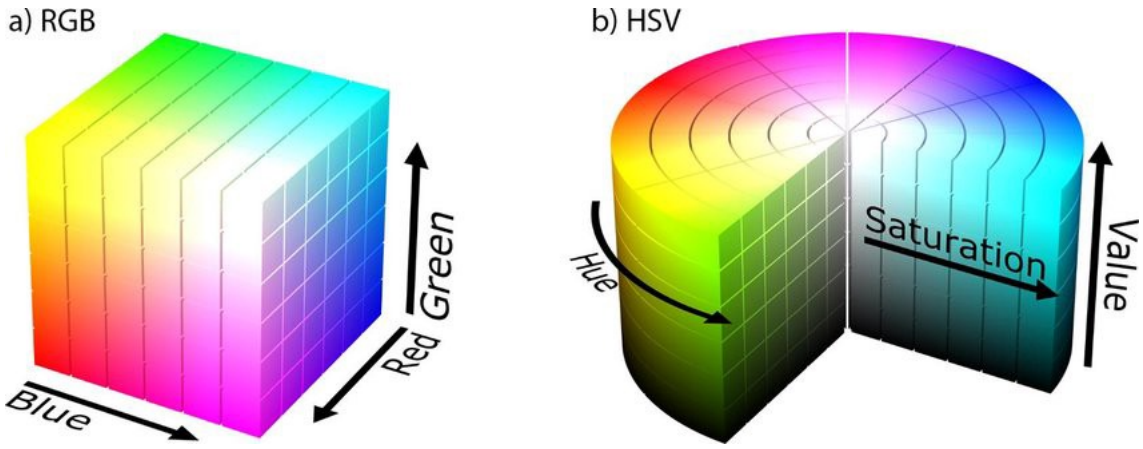


Figure 2: A visualization of the RGB and HSV colorspace

The approach for this has been brute force, by printing the values for each slice in the console and looking for patterns for each type and setting if-statements accordingly. The terrain types are then saved to a 5x5 array as letters representing the terrain types. The function that is called to generate this array is called returnCol. This function is being called for each slice until a complete matrix of representative letters has been generated (See fig. 3). This, however, is not the ideal way to implement a terrain identifying function. A different, more viable, approach would be to record a data-set for each type of terrain and using k-Nearest-Neighbor(kNN) to determine the most probable terrain type. The data-set used for kNN would be comprised of distinguishing features of each type of terrain, for example the hue and saturation values. The idea behind kNN is to plot the features as points in Cartesian space and comparing the distance between points from known terrain types to the points of unknown tiles to determine their most probable type.

3.2 Connected Component Analysis

When the terrain types have been determined, the next step is to figure out which and how many sections of the same type are adjacent and make up "islands". This is an area in graph-theory more commonly known as connected component analysis. In figure 3 the first matrix is the output of the terrain-type detection. To compute the connected component map, so that it accounts for multiple distinct islands, it is first converted, by type, to a "binary image" containing all zeros in places that

does not match the current type, and ones in every place that contains the current type. For this example, the current type is the pastures, denoted 'P'. The binary image for pastures can be seen in the center of figure 3. When the matrix of ones and zeros has been generated, a programming algorithm called "Grassfire" is used to distinguish the islands. Grassfire is an algorithm for connected component analysis that works on binary images, and the goal is to give all groups/islands of connected pixels/components a unique ID. Grassfire is implemented by going through each pixel/component and burning all the visited components. When a island/object is encountered, all associated "pixels" of that type are burned and named before continuing. This can be done both recursively or sequentially, but the result is the same.

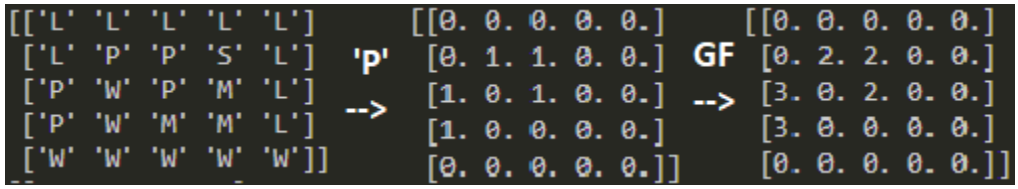
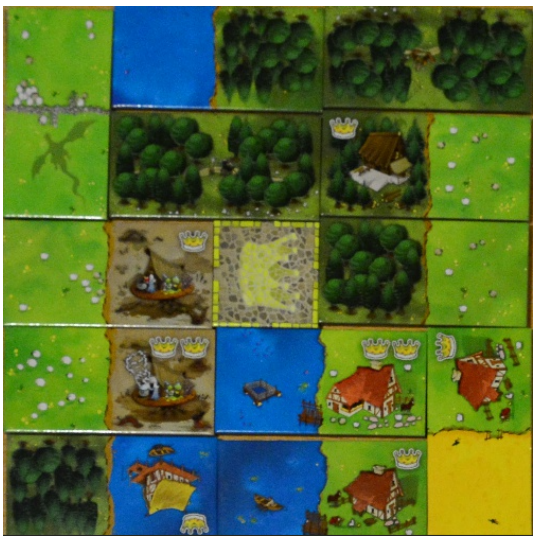


Figure 3: An example of the component map conversion

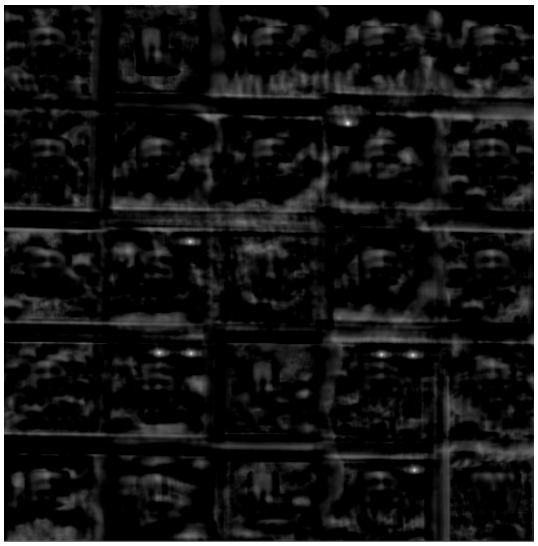
In this case there is two "islands" of pastures, so the first island that is encountered is assigned the unique ID "2" and the second one "3". This could go on for as many islands as necessary. The third matrix in figure 3 is the resulting matrix from the connected component analysis, and this is repeated for each terrain type.

3.3 Crown Detection

Crown detection is done by use of template matching, which is an area of neighborhood processing. Six different templates of crowns, one for each terrain type, is rotated to 4 different orientations and matched with the board image. This creates a heat map of the original image, where hot spots are located where the templates match the best. Match threshold values are passed to the function along with the templates, to determine how "hot" an area should be to be considered a match. The coordinates of all the matches, along with an offset set of coordinates is appended to a list called "Boxes". This "boxes" list contains the coordinates for the bounding boxes of the crowns.



(a) The original image



(b) Template matching heat map

Figure 4: This figure shows the result of template matching the upright crown with the board.

The hot spots in question can be seen in figure 4, where white-hot spots are present in the same locations that the crowns are in, in the original image. The detectCrown function carries out template

matching for all four orientations of every template and saves these coordinates, and the drawCrown function uses non max suppression on these coordinates. This is to prevent the inevitable multiple bounding box coordinates from being drawn on the final image. Non max suppression finds appropriate bounding boxes and removes all others. When the appropriate boxes have been isolated, they are drawn on the board image and the location of the crowns is saved in a 5x5 array called crownCount.

3.4 Score Calculation

Once all the crowns have been found and saved in appropriate indexes of the crownCount array and while the islands of each type of terrain are being distinguished, the scores of each island is calculated and added to the final score. This is done by looping through each island, by their unique ID number, and determining the number of adjacent tiles and whether these tiles contain any crowns. Simple multiplication carries out the calculation, and the sum of scores for each terrain type is returned by the calcPoints function. Finally, the final score is drawn on the board, making a resulting image seen in figure 5.

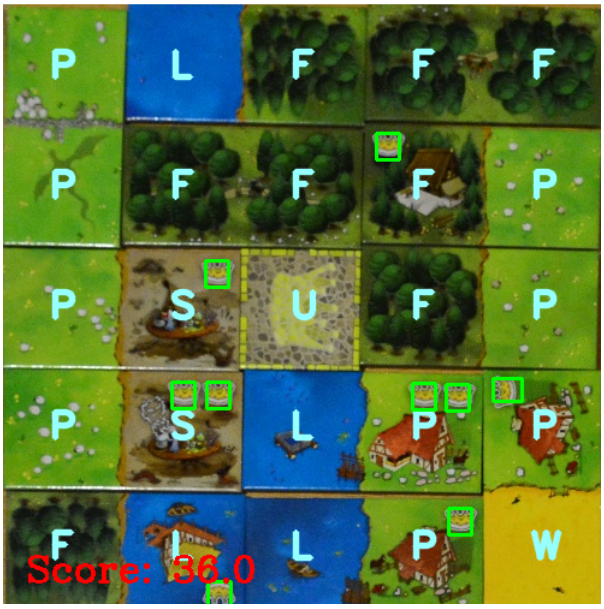


Figure 5: How the final image looks, when the program has finished running

4 Performance

The program has been tested on all 74 images in the cropped and perspective corrected data-set, which has shown that the overall ability to detect the correct amount of crowns is at 99.05%. The ability to correctly detect the terrain type has been shown to be 91.24%. These results are quite satisfactory, but they affect the overall score calculation quite significantly. The program was able to get the correct score on 43.24% of the data-set, that is 32 correct scores, with an average deviation of 12.5% from the actual score on incorrect boards. The program results from the "test-set" part of the data-set, allocated for testing were: Tile detection percentage at 86.67%, crown detection at 100% and correct scores on 26.66% with an average deviation from the correct score of 31.2%.

5 Conclusion

While the program has perfectly good working parts, the score calculation is not consistent or accurate. This is of course because of the propagation of error through small errors in crown and tile detection. If the program should be improved, a more robust implementation of tile detection would have to be implemented, and perhaps a crown detection algorithm based on the buildings associated with the

crowns instead of one based on crowns alone. The reasoning for this is, that some crowns are obscured in the images, and cannot be detected with this program as it is now. The buildings are always at least partially visible, which could be enough to reasonably detect the number of crowns present in that tile. The tile terrain type detection would need a huge improvement for this program to become totally viable, as there are many quality and color differences in the images provided, making it extremely difficult to characterize each tile type completely. Some options come to mind, like the previously mentioned kNN approach or maybe even an AI for determining tile types.