DIGITAL IMAGE PROCESSING

PROJECT REPORT

JIGSAW PUZZLE SOLVER

Team ***ZigVals***

Team Number **16**

Project Guide **Dr Vineet Gandhi**

Prachi Agrawal **201401014**

Akanksha Baranwal **201430015**

AIM & SCOPE

The aim is to reconstruct the original image from a set of non-overlapping, unordered, square puzzle parts. Multiple puzzles mixed into one and puzzles with up to 30% missing pieces can be handled.

Following cases have been handled:

1) Single image puzzles with very less variations in patch
2) Single image puzzles with some constant intensity patches
3) Puzzles with missing pieces
4) Puzzle may be a mix of puzzles from different puzzles

APPLICATIONS

- ➔ Assembly and repair are the major robotics application tasks. This can be posed as a jigsaw puzzle assembly problem.
- ➔ The problem of multiple puzzles mixed into one is similar to restoring archaeological findings. For example when torn documents or broken artifacts are found mixed and lack parts.
- ➔ This solution has an interesting application in biology too. In mitochondria of Diplonema, genes are systematically fragmented into small pieces that are encoded on separate chromosomes, transcribed individually, and then concatenated into contiguous RNA molecules.

ASSUMPTIONS

- ➔ Testing has been done on images multiple 512x512 images.
- ➔ The size of the puzzle pieces has been taken to be 64x64.
- ➔ All puzzle pieces are square.
- ➔ The algorithm can be extended to similar square pieces puzzles.

FINAL WORKFLOW

STEP1: Conversion of input image from RGB space to Lab space.

Since in a jigsaw puzzle it is important to keep similar pieces together, similarity would be better measured in Lab space rather than in RGB space.

STEP2: Constructing the dissimilarity matrix using the 'dissimilarity' function as mentioned below.

$$D(p_i, p_j, right) = \sum_{k=1}^{K} \sum_{d=1}^{3} \|([2p_i(k, K, d) - p_i(k, K-1, d)] - p_j(k, 1, d))\|$$

K - piece size,

d - dimension in the LAB color space (all 3 spaces are combined together and added)

p_i, p_j - two pieces between which dissimilarity is being calculated

In this function, the dissimilarity between every pair of pieces in right direction is being calculated by considering the last 2 columns of the first piece and the first column of the second piece (similarly for all four directions). Here **asymmetric dissimilarity** (that is, D(p_i, p_j, right) = D(p_j, p_i, left)) has been used. This gains importance when the **puzzles have missing pieces** as it helps in placing the border pieces.

This function is the most computationally expensive step O(4mn), where m denotes the number of rows and n denotes the number of columns in the input image respectively in all 4 directions.

STEP3: Constructing the compatibility matrix using the 'dissimilarity' function as mentioned below.

$$C(p_i, p_j, right) = 1 - \frac{D(p_i, p_j, r)}{secondD(p_i, r)}$$

secondD(p_i, r) is the value of the second best dissimilarity of piece p i to all other pieces with relation r.
D(p_i, r) is the value of the best dissimilarity of piece p i to all other pieces with relation r.

$$r \in \{up, down, left, right\}$$

A small dissimilarity between two pieces may not always be a reliable metric to conclude adjacency. This is especially true in **smooth regions** where the dissimilarity between any

two pieces would be a very small value. So we need to consider closest as well as second closest neighbor. If relative dissimilarity between the closest and second closest neighbor differs, then that piece is more likely to be the neighbor. This has an effect of **normalization** so that in the algorithm thresholds become image invariant.

STEP4: Calculating the mutual compatibility

A piece's **position is determined relative** to other pieces instead of absolute position. So, for a piece's position to be fixed, it needs to be a strong neighbor of other pieces. This can be measured by taking an average of the compatibilities in corresponding complementary directions.

$$\widetilde{C}(p_i, p_j, r_1) = \widetilde{C}(p_j, p_i, r_2) = \frac{C(p_i, p_j, r_1) + C(p_i, p_j, r_2)}{2}$$

relation r_2 is the opposite of relation r_1

Best buddies metric

Two pieces are best buddies if **both agree that the other piece is their best neighbor** in the corresponding spatial direction. The best neighbors in the four directions have been calculated and stored in a matrix (display_mat). Best neighbour for a piece in a particular direction is the one having maximum mutual compatibility with it in that direction. The respective best neighbors in complementary directions can be accessed in O(1) from the display_mat and compared to know whether it is a best buddy or not.

STEP5: Finding the first piece

Using the display_mat, we search for a piece that **has best buddies in all four directions**. The more distinctive such a piece is, the better will the coming pieces be placed relative to this piece I.e. their position would be more definite. In case of missing pieces in puzzle, such a piece may not exist as it need not have neighbors in all four directions. In these cases, we consider fewer neighbors.

The metric for deciding this **has been calculated by adding up the mutual compatibility values of best buddies** in corresponding directions.

STEP6: Placing the other pieces according to following algorithm

➔ A **pool** is maintained that consists of those pieces whose neighbors were strong pieces and have already been placed.

➔ The next piece(best piece) to be placed is taken from this pool, and the best buddies of this best piece are inserted into the pool.

➔ A threshold value for the compatibility with the best buddies is also taken. (This becomes especially important in the case of multiple images puzzle.)

➔ The best piece selected is such that the sum of the mutual compatibilities (with its closest neighbors) is maximum.

Base algorithm:

```
While there are unplaced pieces
    if pool is empty
        set compatibility of remaining pieces with placed pieces = -inf
        recalculate compatibility
        recalculate mutual compatibility
        recalculate display_mat
        find the first piece of the new unsolved image part
    else
        remove the best piece from the pool
        add the best buddies of the best piece into the pool
    place the best piece
    if (bestpiece not placed) and (pool is empty) and (old_bestpiece=best piece)
        break
    else
        old_bestpiece=bestpiece
```

To place the above obtained best piece from the pool, the best buddies of this piece are taken into account. For each best buddy, following sequence of operations are performed.

```
If best_buddy not already placed
        add best_buddy to the pool
otherwise
        the best_buddy provides a location for the best piece to be placed
        according to its own position and its spatial relation with the best
        piece thereby updating the 'isthere' map.
```

The **'isthere' map** stores the position of a piece as its key and its frequency as its value. If a position hasn't been discovered earlier, it is added to the map otherwise its count is updated. After iterating over all the best buddies, the position corresponding to maximum frequency is selected for the best piece to be placed in.

```
For all keys in isthere map
    if isthere(key) < max_frequency
        max_frequency = isthere(key)
        final_position = key
```

DEVELOPMENT OF THE ABOVE WORKFLOW

FOLLOWING IS AN OUTLINE OF THE STEPS AND ERRORS WHICH LEAD US TO THE ABOVE ALGORITHM

Naive approach

Using only the compatibility function, the display_mat was calculated. Depending upon whether we get best buddies continuously, we kept on adding pieces to form rows from left to right. Then according to the display_mat{down} value, we arranged the first pieces of each row to form the complete image. This worked well for few images with single puzzle.

For the placement of the pieces, horizontal strips are constructed first using the below algorithm followed by their vertical placement.

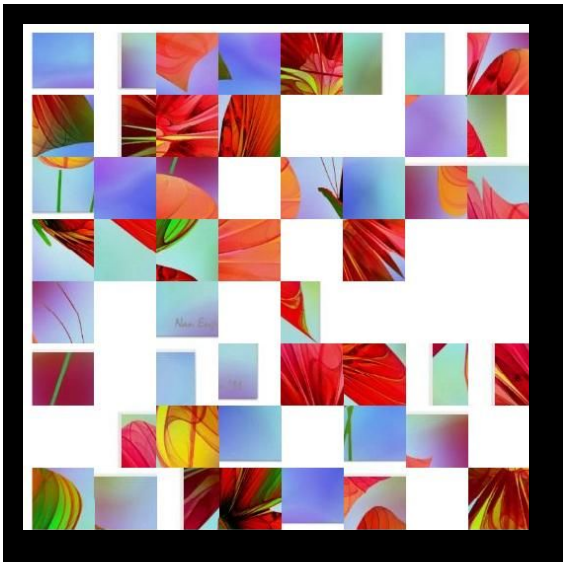
```
curr_piece = Find a leftmost piece.
// right[] array will represent the horizontal strip formed
while there exists a right neighbor of curr_piece satisfying mutual correspondence
    right[curr_piece] = right_neighbor_of_currpiece  (using display_mat)
    curr_piece = right[curr_piece]
```

Similarly, the above obtained horizontal strips will be aligned vertically.

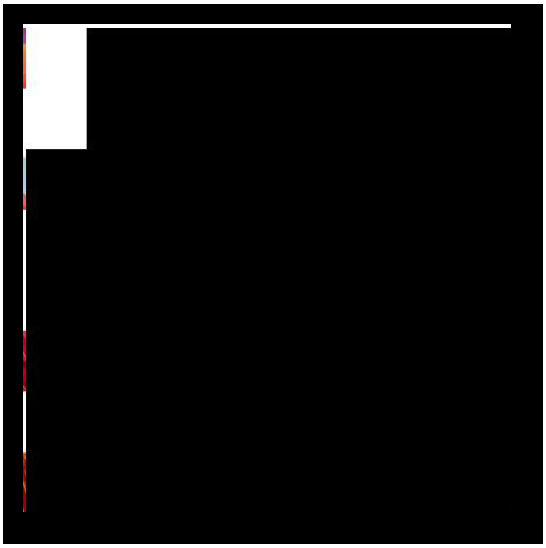
Problems: The above solver doesn't handle the pieces lying in the constant intensity patch region. If the loop breaks anytime in between, complete rows may not get formed. It is not able to handle puzzles with missing pieces. The complexity of this approach is O(mn).

Expected result:

Given puzzle:



Wrong incomplete result:



Involving the mutual compatibility

Construction of the display_mat just using the mutual compatibility matrix. Here placement is not being done on fly. First the **relative position of pieces is being stored** in both horizontal and vertical direction and then later after we get all the values construction is being done. For every piece, the neighbors in all the spatial directions are stored in four lists (right, left, top, bottom).

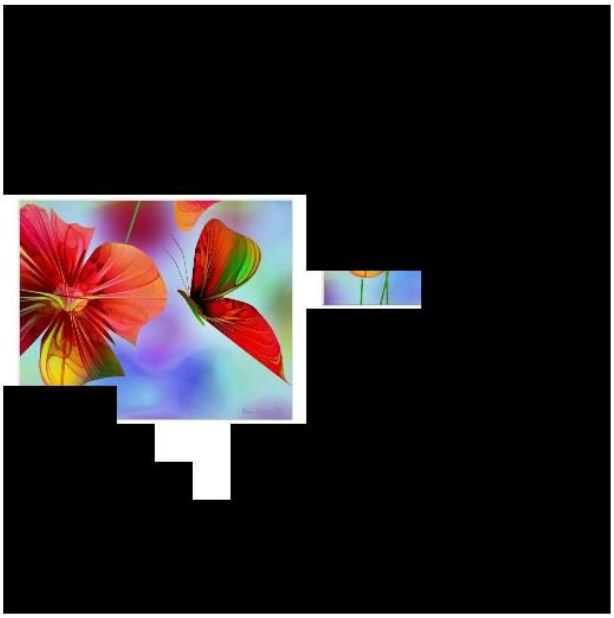
To place the pieces in their relative positions, a recursive approach has been followed. Initially an output matrix double the size of the input image with all zeros is constructed. The first piece obtained by the base algorithm is placed at the center of the output matrix. Then the traversal in all four spatial directions for this first piece is performed with best buddies and their positions in respective directions.

```
function build(curr, i j)
        if curr has not been placed
                store curr in output matrix
                build(curr->left, i, j-1)
                build(curr->right, I, j+1)
                build(curr->top, i-1, j)
                build(curr->bottom, i+1, j)
        else
```

```
                    return
```

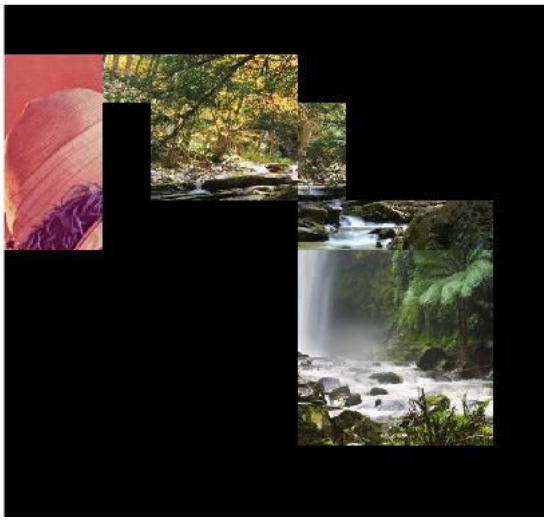
Improvement: This approach helps in handling the border failures (as in case of the previous approach). It can also to a certain extent help in the missing pieces as the algorithm doesn't require the pieces to be connected together. Can also handle multiple pieces.

Now the above input puzzle gets formed in the following way

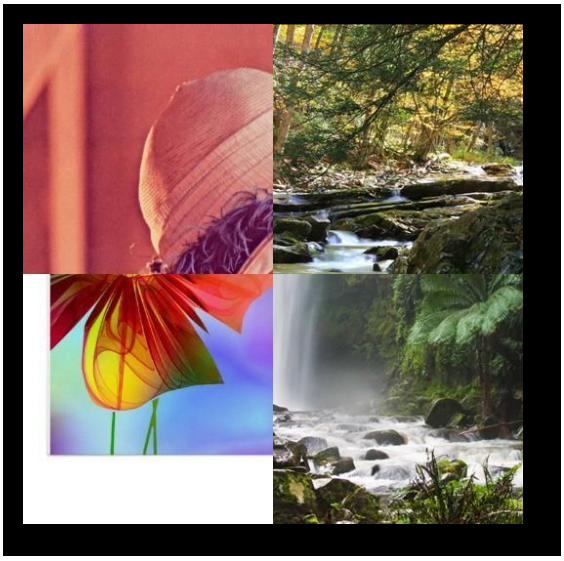


Problems: The more uncertain pieces(pieces whose best buddies are not yet placed) might get placed in the very starting phase of reconstruction, thus leading to erroneous results. The complete reconstruction will happen relative to the not so definite piece and the error would keep on increasing. This is most prominent in puzzles with mixed image puzzles because it gets confused at the border of these subimages.

Actual output (mixture of 4 puzzles together in one):



Expected output (the given 4 images should have come out)



<u>On fly placement</u>

This is the procedure we have discussed in the work flow.

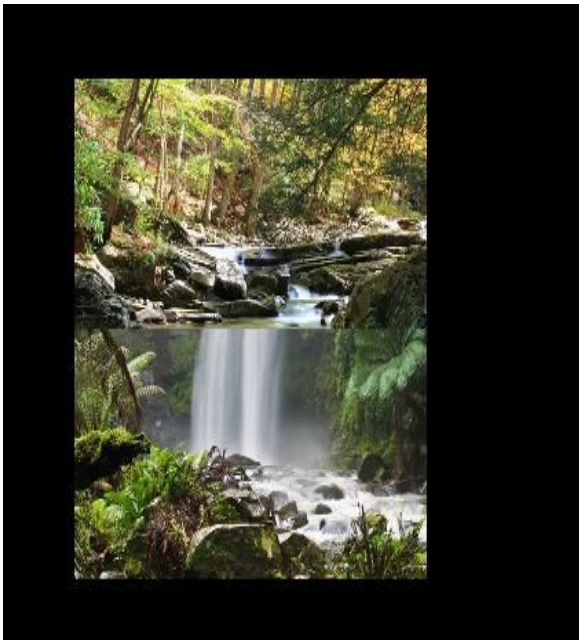
➔ The difference with the previous method is that now we are fixing the positions in the order of best pieces as and when they come. So the likelihood of a piece being placed at the wrong place is very less.

➔ One major problem we faced was that while calculating the relative positions, the addresses returned by different best buddies sometimes came out to be different. This was handled using the isthere map as discussed above.

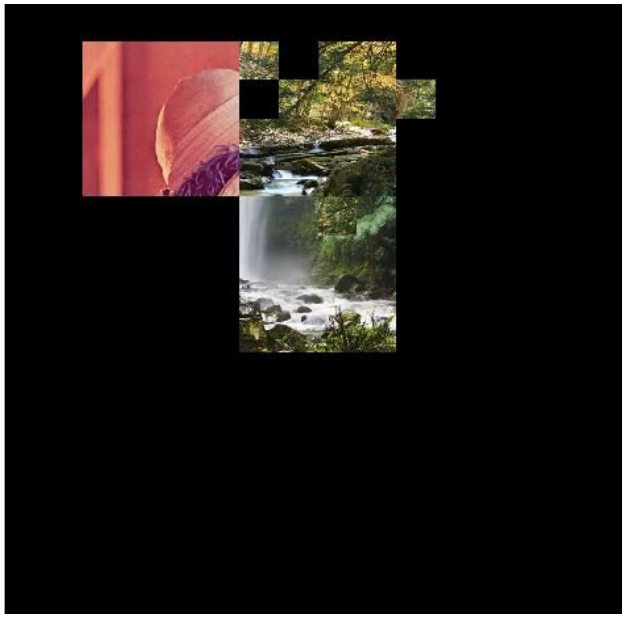
One of the problems due to this was that pieces would get overwritten or haphazardly placed

➔ Also the corner and the border pieces involving multiple images puzzle was mismatching the best buddies if it was not able to find any neighbour. We observed that in these cases, the value of the mutual compatibility was particularly very low. So this was handled by setting proper threshold while returning best buddies of the best piece.

Haphazard placing due to this problem



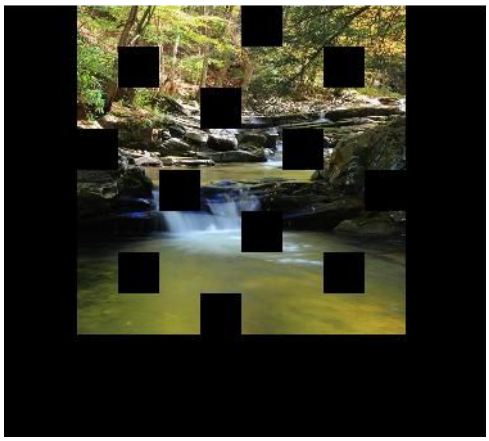
(expected)



Improvement: Now puzzles involving a mix of puzzles into one single puzzles could neatly get distinguished and get separated out. Puzzles with up to 30% missing pieces could also be solved using this.

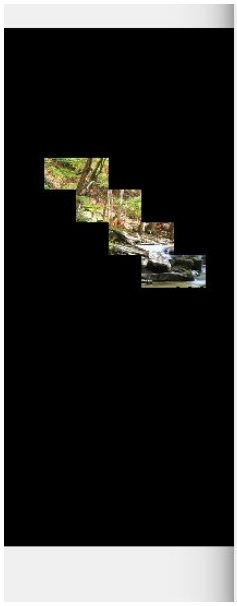
RESULTS

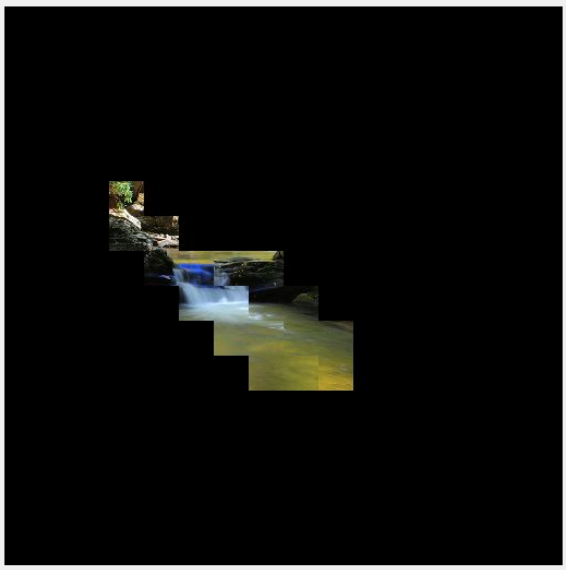
Reconstructed Puzzle with missing pieces (1/5th pieces missing) :

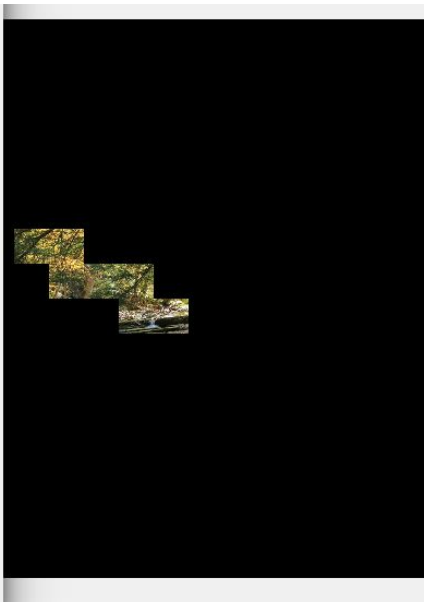


Reconstructed Puzzle with missing pieces (1/4th pieces missing) :

Reconstructed Puzzle with missing pieces (1/3rd pieces missing) :



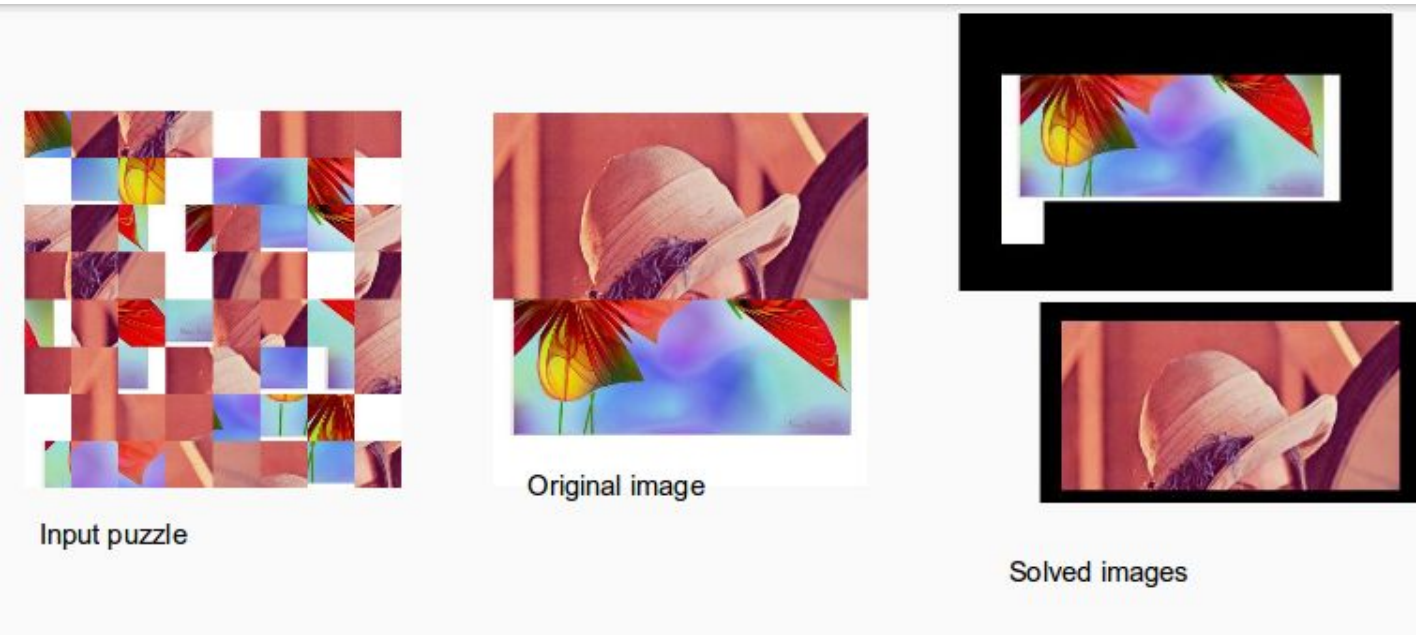


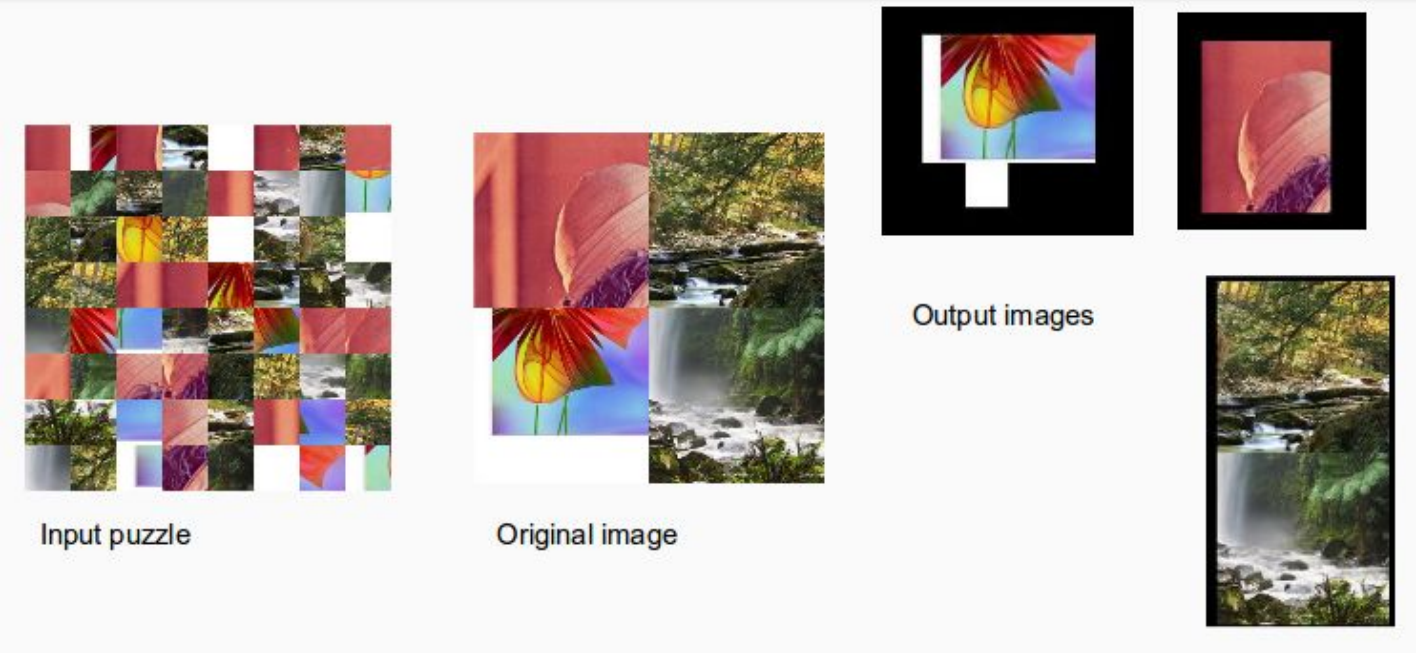


Input puzzle

Original image

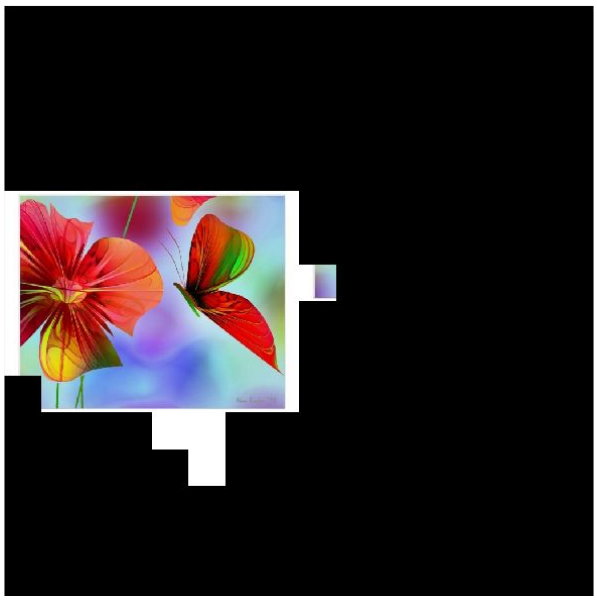
Solved image

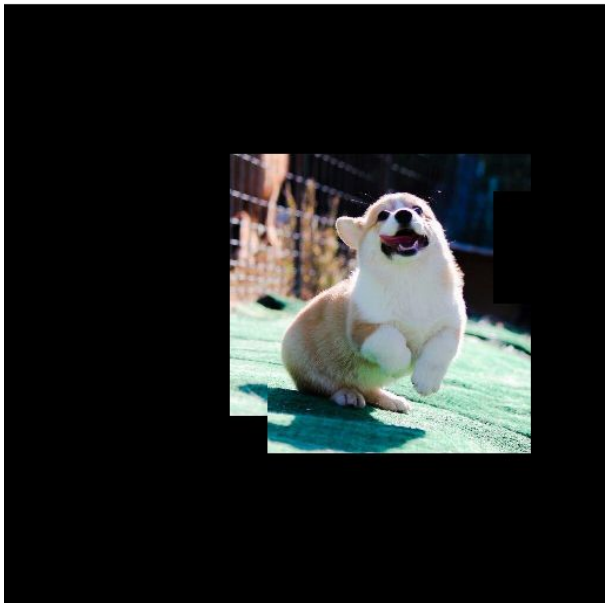




Problems: It is difficult to handle puzzles with white borders or those having completely constant patches as the algorithm gets confused with the best buddies and starts placing things erroneously. Also in cases where the missing pieces are more, the algorithm still searches for the closest neighbors and tries to fill in the gaps.

For example





CHALLENGES AND LIMITATIONS

The algorithm has a high complexity and the system used by the researchers has high computational power. We do not have access to such a system, so working with large number of puzzle pieces is difficult.

We were not able to efficiently handle puzzles having constant intensity patches.

REFERENCES

Genady Paikin and Ayellet Tal. Solving multiple square jigsaw puzzles with missing pieces. In Computer Vision and Pattern Recognition (CVPR), 2015 IEEE Conference.

Code for generating the puzzle pieces taken from:
https://www.cs.bgu.ac.il/~icvl/icvl_projects/automatic-jigsaw-puzzle-solving/