JIGSAW PUZZLE SOLVER

Team Zigvals
Prachi Agrawal 201401014
Akanksha Baranwal 201430015

AIM

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.

SCOPE

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

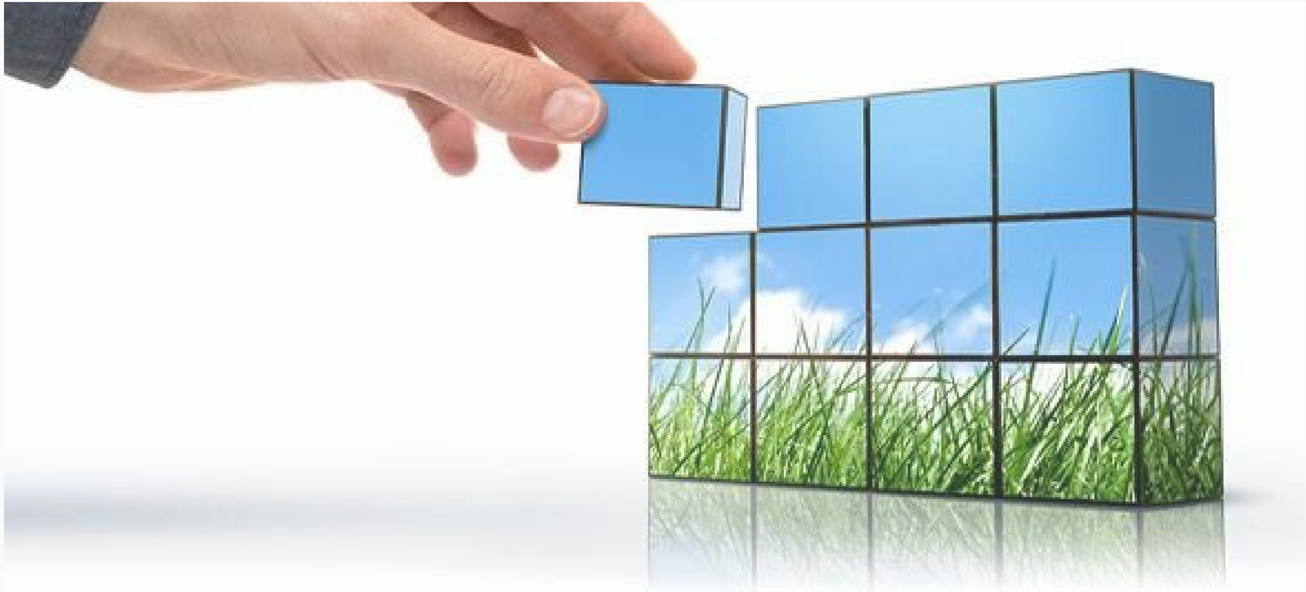
- Assembly and repair are the major robotics application tasks. This can be posed as a jigsaw puzzle assembly problem.

- 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.

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 in shape.
- The algorithm can be extended to similar square pieces puzzles.

DEVELOPMENT OF SOLUTION



Steps

1. Conversion of input image from RGB space to Lab space

   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.

Steps

2. Constructing the dissimilarity matrix using the 'dissimilarity' function

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

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

- 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
- **Asymmetric dissimilarity** (that is, $D(p_i, p_j, right) = D(p_j, p_i, left)$) has been used.
- Important when the **puzzles have missing pieces** as it helps in placing the border pieces.
- Computationally most expensive step O(4mn)

Steps

3. Constructing the compatibility matrix using the 'dissimilarity' function

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

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 -> {up, down, left, right}

- A small dissimilarity between two pieces may not always be a reliable metric to conclude adjacency (smooth regions).
- 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.**

Steps

4. Calculating the mutual compatibility

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

relation r_2 is the opposite of relation r_1

➔ An average of the compatibilities in corresponding complementary directions proves to be a useful metric to find a piece with the strongest neighbors in all spatial directions

Best buddies metric

Two pieces are best buddies if **both agree that the other piece is their best neighbor** in the corresponding spatial direction.

Steps

5. Placing

- **Naive Approach**

NAIVE APPROACH

- ➢ Calculation of display_mat (store of best buddies for every piece) using compatibility function
- ➢ Mutual compatibility not yet calculated.

display_mat

64x8 double

	1	2	3	4	5	6	7	8
	right		left		top		bottom	
Current piece --->1	0.4251	61	0.4648	54	0.4633	4	0.0357	42
2	0.9388	17	1	6	0.9645	47	-0.0052	40
3	0.9359	15	0.9152	30	0.8385	28	0.9160	4
4	0.9530	18	0.9141	9	0.9160	3	0.4633	1

NAIVE APPROACH

Depending upon whether we get best buddies continuously, we kept on adding pieces to form rows from left to right.

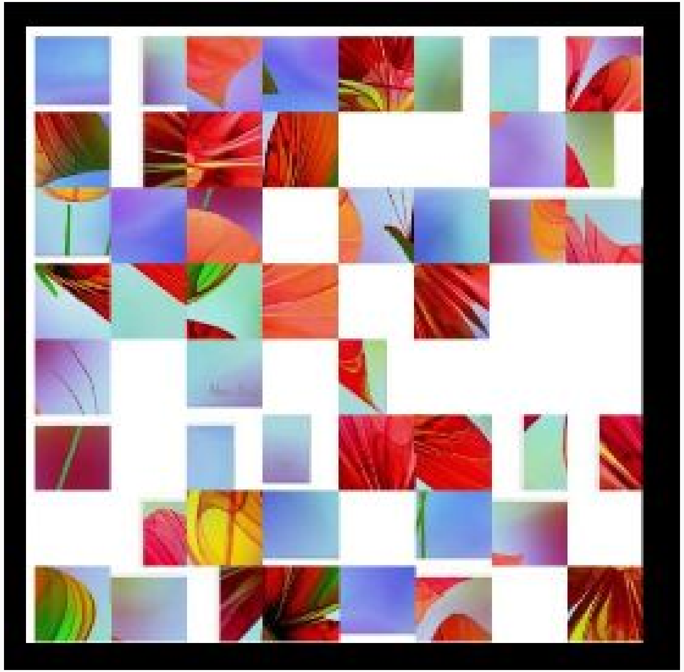
NAIVE APPROACH

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]
```

NAIVE APPROACH

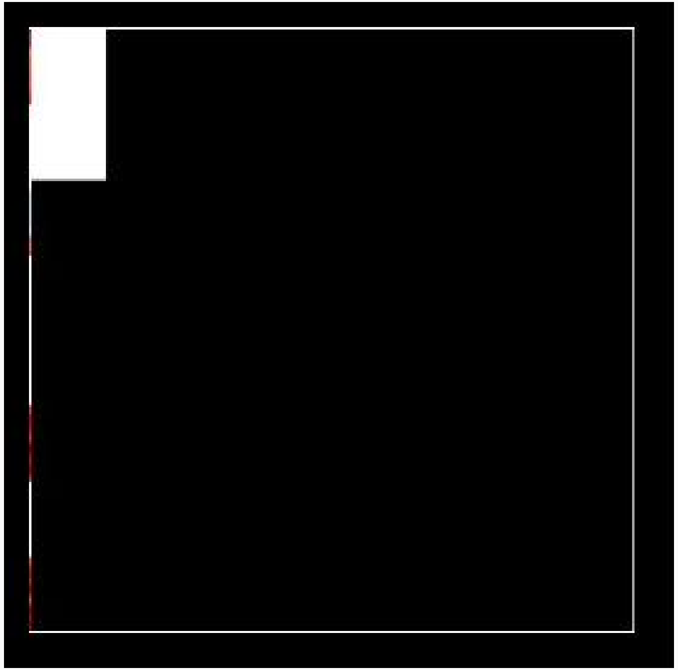
Input image



NAIVE APPROACH

Expected Output

Actual Output



- The above solver doesn't handle the pieces lying in the constant intensity patch region.
- It is not able to handle puzzles with missing pieces.
- The complexity of this approach is O(mn).

- Involvement of mutual compatibility

Involvement of mutual compatibility

- ➔ Calculation of display_mat using mutual compatibility function.
- ➔ Relative position of all pieces is being stored in both horizontal and vertical directions.
- ➔ Recursive placement of pieces in their relative positions for all the pieces at once.

```
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
```

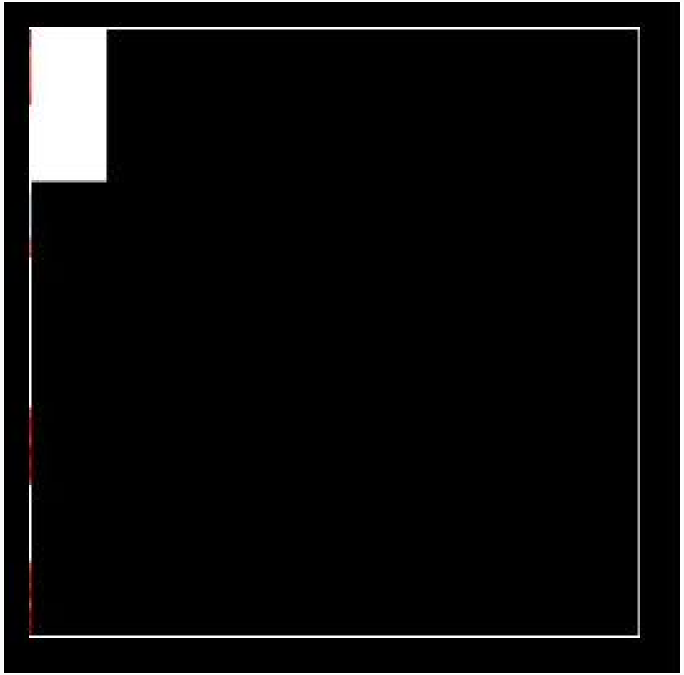
Involvement of mutual compatibility

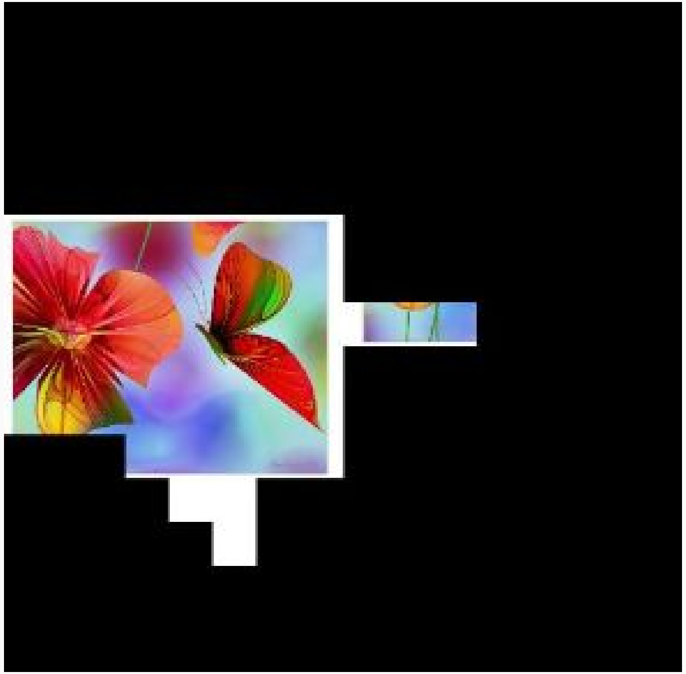
➔ Intermediate step in recursive placement of pieces

0	16	3	0
0	22	11	34
0	49	0	45
0	$\boxed{31}$	28	58
0	0	0	0
0	0	0	0

Involvement of mutual compatibility

Improvement from the naive approach



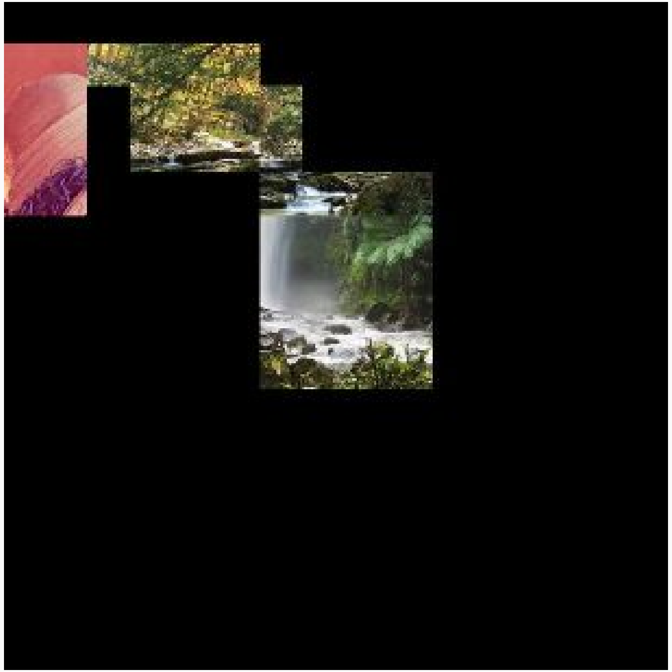


Involvement of mutual compatibility

????????

Expected Output

Actual Output



- 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.
- Most prominent in puzzles with mixed image puzzles as it gets confused at the border of these subimages.
- The complexity of this approach is O(mn).

- **On Fly Placement**

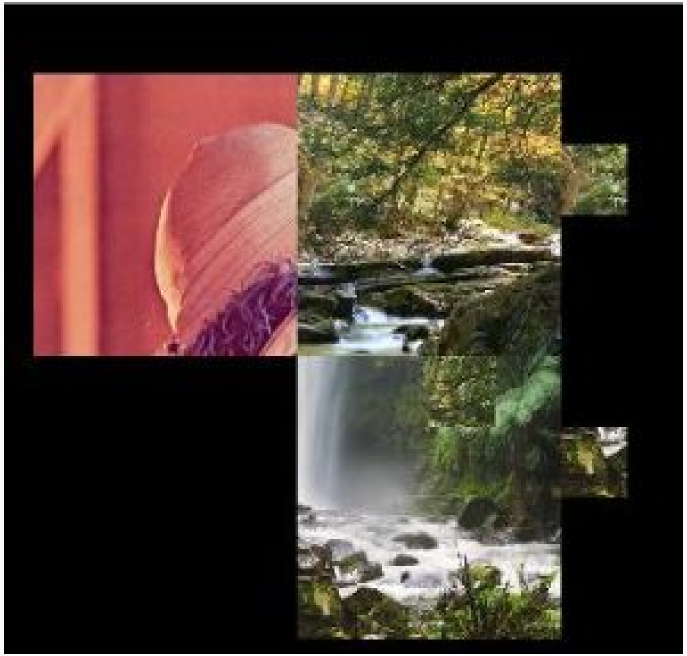
On Fly Placement

- 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.

On Fly Placement

Expected Output

Actual Output



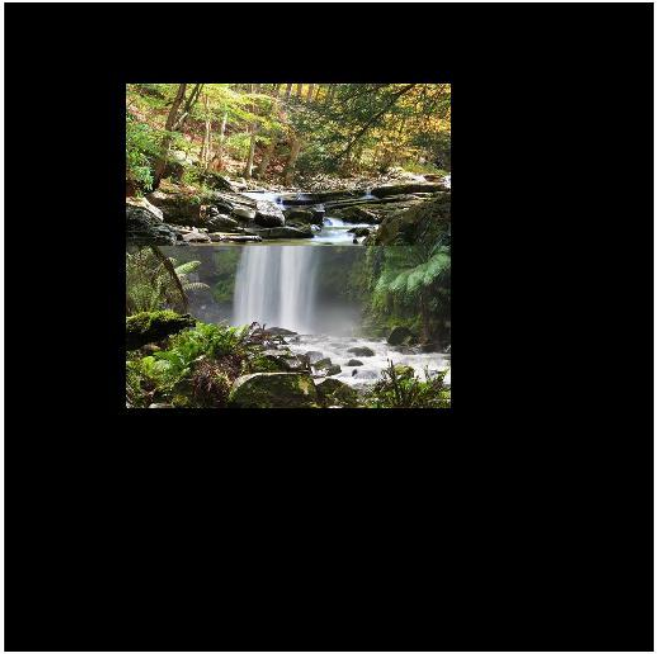
- → while calculating the relative positions, the addresses returned by different best buddies sometimes came out to be different.

On Fly Placement

Result obtained after handling the “multiple” positions issue by allowing a piece to be placed at-most once

On Fly Placement

Expected Output

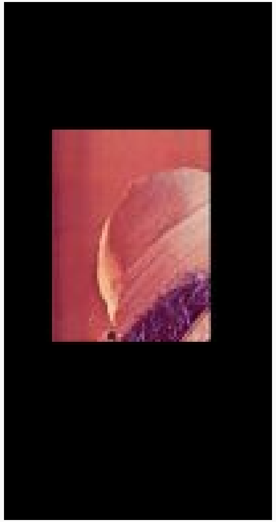


Actual Output

- The corner and the border pieces involving multiple images puzzle was mismatching the best buddies if it was not able to find any neighbour.

On Fly Placement

➔ We observed that in the cases with the previous problem, 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.

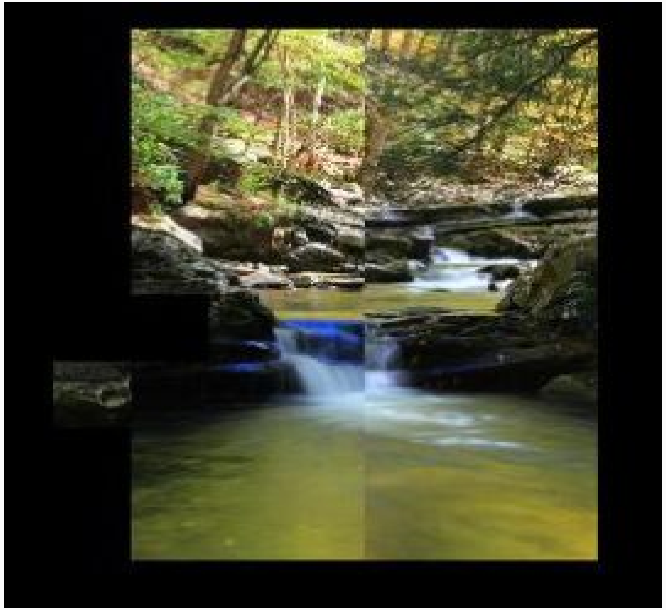


On Fly Placement

Reconstructed Puzzle with missing pieces

1/5$_{th}$ pieces missing

1/4$_{th}$ pieces missing



Steps

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

Steps

6. Placing the other consecutive pieces

- 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.
- 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
```

Operation on best buddies & isthere map

```
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.
```

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

FINAL RESULTS

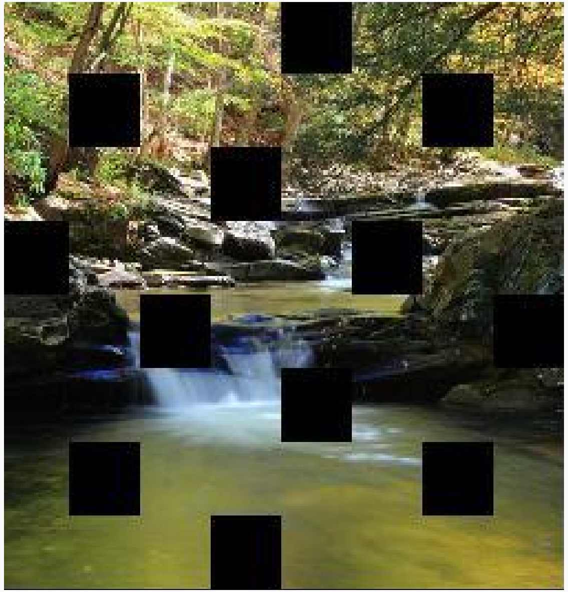
Single images

Input puzzle

Original image

Solved image

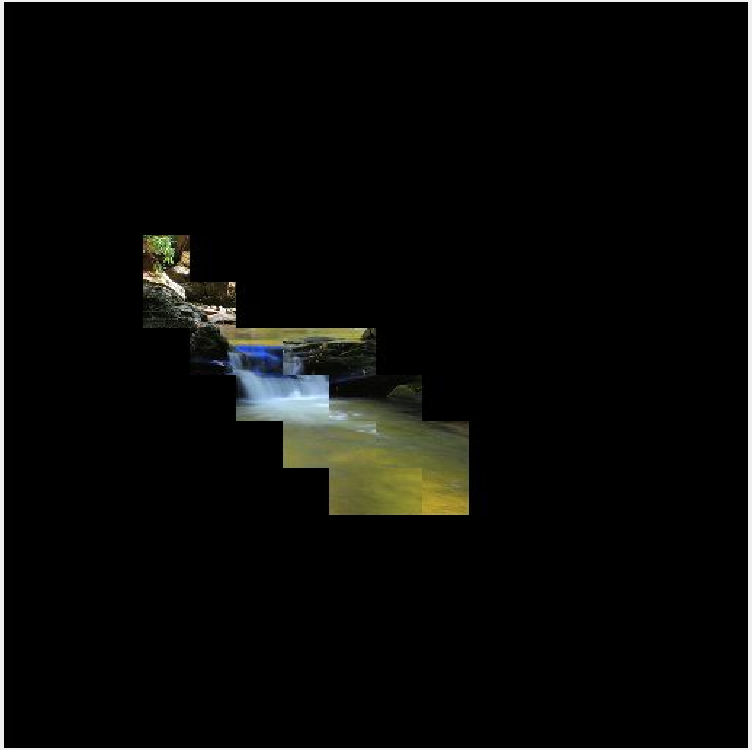
Missing pieces

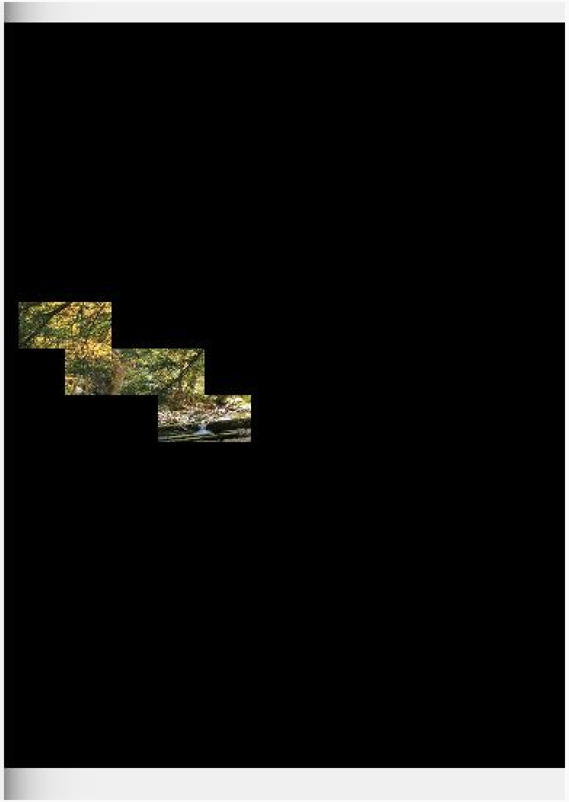


Puzzle with 1/5th pieces missing

Puzzle with 1/4th pieces missing

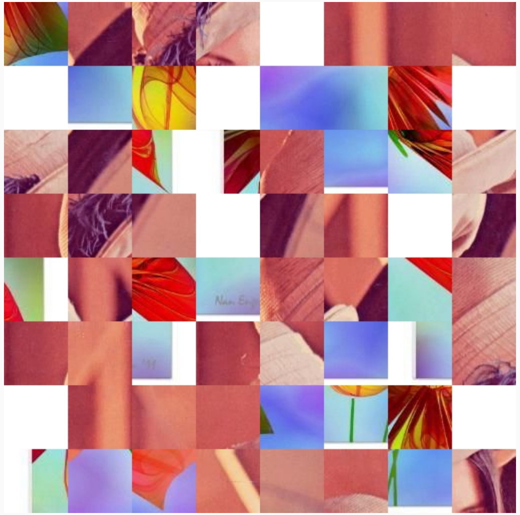
Missing pieces



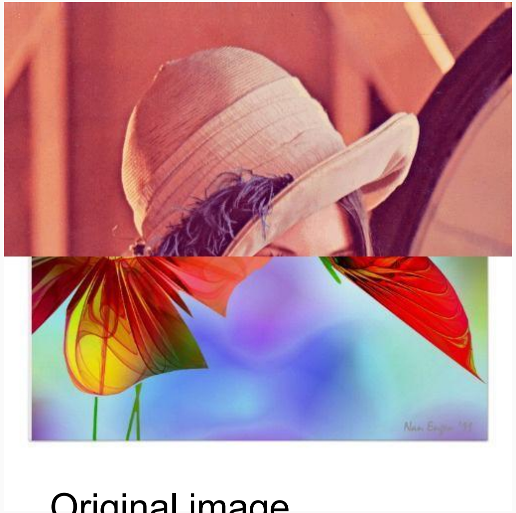


Puzzle with 1/3rd pieces missing

Multiple puzzles in one



Input puzzle

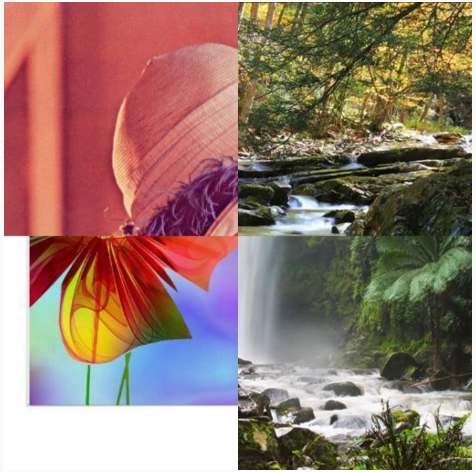


Original image

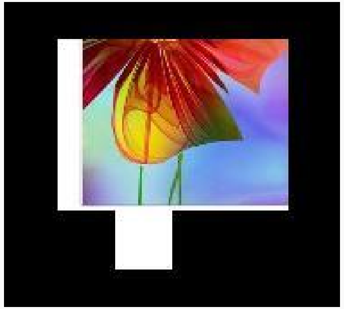
Solved images

Multiple puzzles in one

Input puzzle



Original image



Output images

CHALLENGES & LIMITATIONS

- The algorithm has a high complexity, 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/