Image Resizing using Seam Carving

Siddharth D (CB.EN.U4AIE21064) Abinaya (CB.EN.U4AIE21064)
Sanjana (CB.EN.U4AIE21064)

Contents

1 Introduction

Traditional methods for reducing image size often involve uniform downsampling, which may result in distortions and scaling down all objects in the image proportionally. However, this approach lacks the ability to prioritize preserving interesting objects while removing less crucial areas. In order to address this limitation, we employ Seam Carving. This technique comprises three key steps: calculating the interest score for each pixel, identifying the seam with the minimum interest spanning the image, and subsequently removing that seam. We delve into the implementation details of these three stages and present the outcomes obtained by applying the algorithm to the images.

2 Steps of Seam Carving

Seam carving is accomplished by a 3-step process:

2.1 Calculation of energy

calculation of energy means to calculate the importance of each pixel in the image. The energy of a pixel is a measure of its importance. It is a function of the gradients of the pixel intensities in the image. This is done by smoothing the image and then computing the first x- and y-derivative at each point In our paper we have used 3 different filters to calculate the energy of the image. They are:

1. Sobel Filter
2. Scharr Filter
3. Canny Filter
4. Prewitt Filter

2.1.1 Sobel Filter

The Sobel filter is used to detect edges in an image. It uses two 3×3 kernels which are convolved with the original image to calculate approximations of the derivatives – one for horizontal changes, and one for vertical. If we define A as the source image, and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations respectively, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Figure 1: Sobel Filter

2.1.2 Scharr Filter

The Scharr filter is an improvement of the Sobel filter the improvement is , in that it uses a 3×3 grid with the following values:

2.1.3 Canny Filter

The Canny filter is a multi-stage edge detector. It uses a filter based on the derivative of a Gaussian in order to compute the intensity of the gradients.The Gaussian reduces the effect of noise present in the image. Then, potential edges are thinned down to 1-pixel curves by removing

$$\begin{bmatrix} +3 & 0 & -3 \\ +10 & 0 & -10 \\ +3 & 0 & -3 \end{bmatrix} \qquad \begin{bmatrix} +3 & +10 & +3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix}$$

Scharr-x **Scharr-y**

Figure 2: Scharr Filter

non-maximum pixels of the gradient magnitude. Finally, edge pixels are kept or removed using hysteresis thresholding on the gradient magnitude.

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-(k+1))^2 + (j-(k+1))^2}{2\sigma^2}\right); 1 \le i, j \le (2k+1)$$

Figure 3: Canny Filter

2.1.4 Prewitt Filter

The Prewitt filter is used for detecting horizontal and vertical edges in an image. It is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each point in the image, the result of the Prewitt operator is either the corresponding gradient vector or the norm of this vector. The Prewitt operator is based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical direction and is therefore relatively inexpensive in terms of computations like Sobel and Scharr.

$$\mathbf{G_x} = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G_y} = \begin{bmatrix} +1 & +1 & +1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} * \mathbf{A}$$

Figure 4: Prewitt Filter

2.2 Finding Seams

In general cases seams are found using dynamic programming in our paper we have used 2 different methods to find the seams. They are: 1. Dynamic Programming 2. A* Algorithm

2.2.1 Dynamic Programming for seam finding

How Pathfinding using dynamic programming works: The algorithm works by maintaining a memoization table that stores the minimum path sum from the top-left cell to the current cell.

The dynamic programming approach is used to find the seam with the minimum energy. The algorithm is as follows:

```
function findShortestPath(grid):
    n = grid.rows
    m = grid.columns

    # Create a memoization table to store calculated values
    memo = initializeMemoTable(n, m)

    # Call the recursive function to find the shortest path
    return recursiveShortestPath(grid, 0, 0, memo)

function recursiveShortestPath(grid, i, j, memo):
    n = grid.rows
    m = grid.columns

    # Base case: if we reach the bottom-right cell, return its value
    if i == n-1 and j == m-1:
        return grid[i][j]

    # If the value is already calculated, return it from the memo table
    if memo[i][j] is not null:
        return memo[i][j]

    # Move right and calculate the minimum path sum
    right = recursiveShortestPath(grid, i, j+1, memo)

    # Move down and calculate the minimum path sum
    down = recursiveShortestPath(grid, i+1, j, memo)

    # Update the memo table with the minimum path sum
    memo[i][j] = grid[i][j] + min(right, down)

    return memo[i][j]

function initializeMemoTable(n, m):
    memo = a 2D array of size n x m

    # Initialize all values to null, indicating that they are not calculated yet
    for i from 0 to n-1:
        for j from 0 to m-1:
            memo[i][j] = null

    return memo
```

2.2.2 A* Algorithm for seam finding

What is A-star Algorithm?: A* is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. One major practical drawback is its O(b^d) space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* is still the best solution in many cases.

How A-star works?: At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

The A* algorithm is used to find the seam with the minimum energy. The algorithm is as follows:

```
function AStar(grid, start, goal):
    n = grid.rows
    m = grid.columns

    # Create a priority queue for open set
    openSet = PriorityQueue()

    # Initialize costs and add the start node to the open set
    costs = initializeCosts(n, m)
    costs[start] = 0
    openSet.push(Node(start, 0, heuristic(start, goal)))

    # Closed set to track visited nodes
    closedSet = set()

    while not openSet.isEmpty():
        current = openSet.pop()

        # Check if the goal is reached
        if current.position == goal:
            return current.cost

        # Mark the current node as visited
        closedSet.add(current.position)

        # Explore neighbors
        for neighbor in getNeighbors(current.position, grid):
            if neighbor not in closedSet:
                # Calculate tentative cost to reach the neighbor
                tentativeCost = costs[current.position] + 1

                if tentativeCost < costs[neighbor] or neighbor not in openSet:
                    # Update costs and add to open set
                    costs[neighbor] = tentativeCost
                    priority = tentativeCost + heuristic(neighbor, goal)
                    openSet.push(Node(neighbor, tentativeCost, priority))

    # If no path is found
    return -1

function initializeCosts(n, m):
    costs = a 2D array of size n x m

    # Initialize all costs to infinity
    for i from 0 to n-1:
        for j from 0 to m-1:
            costs[i][j] = infinity

    return costs

function heuristic(node, goal):
    # Example heuristic: Manhattan distance
    return abs(node.row - goal.row) + abs(node.col - goal.col)

function getNeighbors(position, grid):
    n = grid.rows
    m = grid.columns
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []

    for move in moves:
        new_row, new_col = position.row + move[0], position.col + move[1]

        # Check if the new position is within bounds and not blocked
        if 0 <= new_row < n and 0 <= new_col < m and grid[new_row][new_col] != 1:
            neighbors.append((new_row, new_col))

    return neighbors
```

2.3 Reamoving Seams

After finding the seam with the minimum energy, we remove it from the image. This is done by shifting the pixels to the left of the seam to the right by one pixel. This is done by using the following algorithm:

```
function removeSeam(grid, seam):
    n = grid.rows
    m = grid.columns

    # Create a new grid with one less column
    newGrid = a 2D array of size n x m-1

    for i from 0 to n-1:
        for j from 0 to m-1:
            # Shift the pixels to the left of the seam to the right by one pixel
            if j < seam[i]:
                newGrid[i][j] = grid[i][j]

            # Copy the pixels to the right of the seam as is
            else:
                newGrid[i][j] = grid[i][j+1]

    return newGrid
```

3 Analysis of different filters used to calculate energy

In this section we will be discussing the results obtained by using different filters to calculate the energy of the image. We have used 4 different filters to calculate the energy of the image.

3.1 Using Sobel filter