Content-Aware Image Resizing with Seam Carving

> **Author: [李思晴 12212964；张子涵 12210464；欧炜娟 12212252]**

Team Role

Li Siqing is responsible for algorithm development, including energy function calculation, shortest path determination, and image shrinking and enlarging algorithm implementation.

Zhang Zihan is responsible for building the repository and primarily focuses on the development of the GUI basics, including file operations, image display, and other human-computer button designing,as well as collaboratively complete the basic parts of the algorithm.

Ou Weijuan is in charge of implementing the image enlargement functionality and the human-computer interaction part, as well as contributing to report writing.

Introduction

In the field of image processing, resizing images without compromising their visual content is a challenging task. Traditional resizing methods often result in distorted or cropped images, losing important details. To address this issue, we implemented a seam carving algorithm that allows content-aware resizing by intelligently removing or adding seams (paths of pixels) with the least energy. This report outlines the methodology behind our energy function calculation, shortest path determination, image resizing, and enlargement algorithms. By leveraging dynamic programming and user-defined energy adjustments, we ensure the preservation of essential image features during resizing operations.

Part 1: Energy Function Calculation in Image Processing

The energy function calculation is a crucial step in image processing tasks, particularly in applications such as seam carving for content-aware image resizing. The energy function measures the importance of each pixel in an image based on its surrounding pixels, highlighting areas with significant intensity changes which often correspond to edges or boundaries within the image. This section elaborates on the methodology used for calculating the energy values of pixels in an image and converting these values into a normalized grayscale image for visualization purposes.

1. Energy Calculation

The energy of a pixel is determined by evaluating the gradient magnitude at that pixel. The gradient measures the rate of change of intensity in the image, with larger changes indicating more significant features. For an RGB image, the energy at each pixel is calculated using the following steps:

1. **Gradient Computation**:
    - The gradients in the x-direction (`gradientX`) and y-direction (`gradientY`) are computed separately.

- For each pixel at coordinates `(x, y)`, the gradients are calculated using the intensity differences with its neighboring pixels:

```
Color left = picture.get(x - 1, y);
Color right = picture.get(x + 1, y);
Color up = picture.get(x, y - 1);
Color down = picture.get(x, y + 1);

double gradientX = Math.pow(right.getRed() - left.getRed(), 2)
                 + Math.pow(right.getGreen() - left.getGreen(), 2)
                 + Math.pow(right.getBlue() - left.getBlue(), 2);

double gradientY = Math.pow(down.getRed() - up.getRed(), 2)
                 + Math.pow(down.getGreen() - up.getGreen(), 2)
                 + Math.pow(down.getBlue() - up.getBlue(), 2);
```

2. **Energy Calculation**:

- The energy value for the pixel is the sum of the gradients in both directions:

```
double energy = gradientX + gradientY;
```

3. **Border Handling**:
    - Pixels at the borders of the image are assigned a maximum energy value to ensure they are not part of the seams during the seam carving process. This is because border pixels do not have enough neighboring pixels to compute accurate gradients.

```
if (isBorder(x, y, picture)) {
    return BORDER_ENERGY;
}
```

2. Energy Matrix Conversion to Grayscale Image

To visualize the energy distribution across the image, the calculated energy values are converted into a normalized grayscale image. This process involves the following steps:

1. **Normalization**:
    - The energy values are normalized by dividing each value by the maximum energy value in the matrix. This ensures that the energy values range from 0 to 1, where 1 corresponds to the maximum energy.

```
double maxVal = calculateMaximumValue(energy);
float normalizedGrayValue = (float) energy[i][j] / (float) maxVal;
Color color = new Color(normalizedGrayValue, normalizedGrayValue,
normalizedGrayValue);
```

2. **Image Creation**:
    - A new `BufferedImage` is created, and each pixel is assigned a grayscale color based on its normalized energy value. This visualization aids in understanding the importance of different regions in the image.

```
BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_RGB);
image.setRGB(i, j, color.getRGB());
```

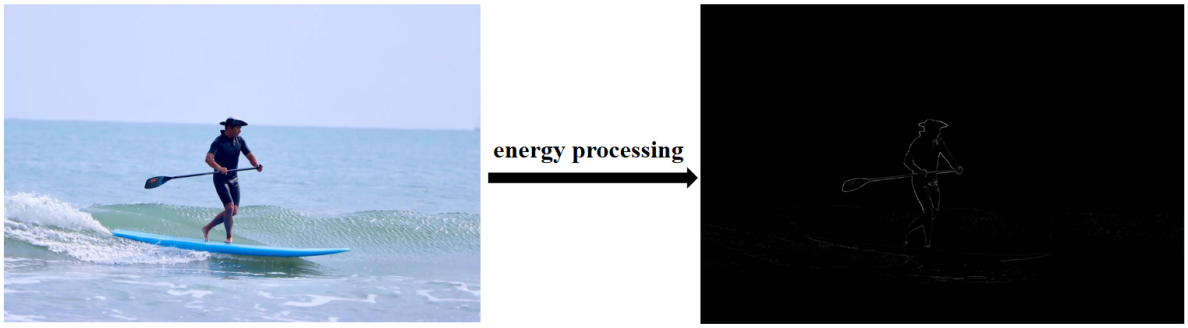
3. User-Selected Energy Adjustment

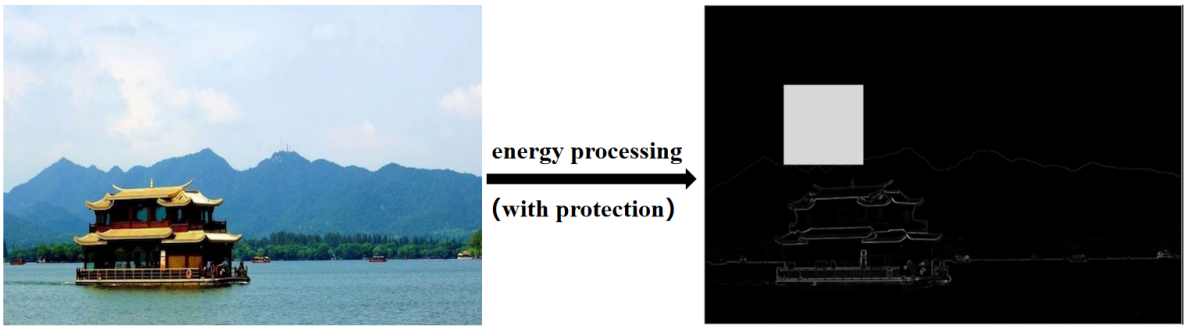
In interactive applications, users may want to protect or remove specific regions of an image. This requires recalculating the energy values based on user selection:

1. **User Selection Matrix**:
    - A matrix `userSelection` is provided, where each element indicates whether a pixel should be protected (value `1`) or removed (value `-1`). Pixels with no special designation have a value of `0`.
2. **Energy Adjustment**:
    - For protected pixels, the energy value is set to the maximum energy to prevent them from being part of any seam.
    - For pixels marked for removal, the energy value is set to zero, indicating they should be included in the seam.

```
if (isVertical) {
    if (userSelection[x][y] == 1) {
        energy[x][y] = BORDER_ENERGY;
    }
    if (userSelection[x][y] == -1) {
        energy[x][y] = 0;
    }
} else {
    if (userSelection[y][x] == 1) {
        energy[x][y] = BORDER_ENERGY;
    }
    if (userSelection[y][x] == -1) {
        energy[x][y] = 0;
    }
}
```

4. Example





This approach to energy calculation and adjustment ensures that the resulting image processing tasks, such as seam carving, are performed efficiently while respecting user preferences and maintaining the integrity of important image features.

Part 2: Shortest Path Calculation Using Dynamic Programming

The shortest path calculation in the context of seam carving is a critical step that determines the optimal seam (a connected path of pixels from one edge to the opposite edge) with the least total energy. This step employs a dynamic programming approach to efficiently find the minimum energy seam. The following sections describe the methodology in detail.

1. Initialization of the Energy Matrix

The process begins with the initialization of an energy matrix, where each element represents the energy of the corresponding pixel in the image. This matrix is derived from the gradient magnitudes calculated for each pixel, as described in the energy function calculation section.

2. Construction of the Lookup Table

The lookup table, denoted as `paths`, is a 2D array where `paths[x][y]` stores the minimum cumulative energy required to reach the pixel at `(x, y)` from the top of the image. The construction of this table proceeds as follows:

1. **Bottom Row Initialization**:
    - The last row of the lookup table is initialized with the energy values of the corresponding pixels:

$$\text{paths}[x][\text{height} - 1] = \text{energy}[x][\text{height} - 1]$$

    - This step sets the base case for the dynamic programming recursion.
2. **Filling the Lookup Table**:
    - Starting from the second-to-last row and moving upwards, each entry in the lookup table is filled using the following recurrence relation:

$$\text{paths}[x][y] = \text{energy}[x][y] + \min(\text{paths}[x-1][y+1], \text{paths}[x][y+1], \text{paths}[x+1][y+1])$$

    - This formula ensures that each pixel's entry in the lookup table reflects the minimum energy path to reach that pixel from the previous row, considering the three possible preceding pixels (top-left, top, top-right).

3. Seam Identification

Once the lookup table is fully constructed, the optimal seam is identified by tracing back from the top row to the bottom row:

1. **Finding the Starting Point**:
    - The starting point of the seam is the pixel in the top row with the smallest cumulative energy value:

$$x_{\text{start}} = \arg\min_{x} \text{paths}[x][0]$$

    - This identifies the beginning of the seam.
2. **Tracing the Seam**:
    - The seam is traced back by selecting the pixel with the minimum cumulative energy from the three possible preceding pixels in the next row down:

$$\text{seam}[y] = \arg\min_{i=x-1}^{x+1} \text{paths}[i][y+1]$$

    - This process is repeated for each row from top to bottom, forming the seam.

4. Improved Implementation

The improved implementation introduces several optimizations to enhance performance:

1. **Parallel Computing**:
    - Java's parallel streams (`IntStream.range().parallel()`) are utilized to initialize the bottom row and compute the cumulative energy for each pixel in parallel. This significantly reduces the computation time, especially for large images.
2. **Heuristic Pruning**:
    - During the cumulative energy calculation, the algorithm minimizes unnecessary computations by only considering valid neighboring pixels. This reduces the number of comparisons and updates required, leading to faster execution.
3. **Memory Optimization**:
    - The algorithm updates the user selection region in-place and avoids creating unnecessary intermediate arrays. This reduces the memory footprint and enhances the efficiency of the algorithm.

Detailed Steps

1. **Bottom Row Initialization**:
    - The bottom row of the lookup table is populated with the energy values from the energy matrix using a parallel stream:

```
IntStream.range(0, width).parallel().forEach(x -> {
    paths[x][height - 1] = energy[x][height - 1];
});
```

2. **Cumulative Energy Calculation**:
    - For each row from the second-to-last to the top, the cumulative energy is computed in parallel:

```
for (int y = height - 2; y >= 0; y--) {
    IntStream.range(0, width).parallel().forEach(x -> {
        double min = Double.MAX_VALUE;
        for (int i = x - 1; i <= x + 1; i++) {
            if (i >= 0 && i < width && paths[i][y + 1] < min) {
                min = paths[i][y + 1];
            }
        }
        paths[x][y] = min + energy[x][y];
    });
}
```

3. **Finding the Shortest Path**:
    - After constructing the lookup table, the algorithm identifies the seam with the lowest cumulative energy:

```
private int[] lookupShortestPath(double[][] lookup) {
    int width = lookup.length;
    double shortest = lookup[0][0];
    int minX = 0;
    for (int x = 1; x < width; x++) {
        if (lookup[x][0] < shortest) {
            minX = x;
            shortest = lookup[x][0];
        }
    }
    return findPathStartingAt(lookup, minX);
}

private int[] findPathStartingAt(double[][] lookup, int x) {
    int height = lookup[0].length;
    int[] path = new int[height];
    path[0] = x;
    for (int y = 1; y < height; y++) {
        path[y] = min(lookup, y, path[y - 1]);
    }
    return path;
}
```

The optimized shortest path calculation using dynamic programming, parallel computing, heuristic pruning, and memory optimization significantly improves the efficiency of the seam carving algorithm. These enhancements ensure that the algorithm can handle large images more effectively, providing faster and higher quality resizing.

Part 3: Image Resizing using Seam Carving Algorithm

Image resizing is used to adapt images to different display sizes or aspect ratios. Seam carving is an advanced technique for resizing images while preserving important content and minimizing distortion. This section explains the implementation of seam carving for image resizing and its key components, including the removal and addition of seams, as well as handling user selections.

1. Seam Carving Algorithm Overview

Seam carving involves iteratively removing or adding seams, which are connected paths of pixels, from the image. These seams typically span the entire width or height of the image and pass through low-energy regions. By intelligently selecting seams to remove, the image can be resized while minimizing the distortion of important features.

2. Image Resizing Procedure

The image resizing process involves the following steps:

1. **Initialization**:
    - Initialize the energy calculator and stacks to store vertical and horizontal seams.
    - Accept user input specifying the number of columns and rows to remove from the image.

```
public Picture resize(Picture initialPicture, int removeColumns, int
removeRows, int[][] selection) {
    userSelection = selection;
    verticalSeams = new Stack<>();
    horizontalSeams = new Stack<>();
    Picture picture = removeColumns(initialPicture, removeColumns, true);
    picture = removeRows(picture, removeRows);
    return picture;
}
```

2. **Seam Removal**:
    - Calculate the energy of the image pixels using the energy calculator.
    - Identify the seams with the lowest energy using dynamic programming-based shortest path algorithms.
    - Remove the identified seams from the image by eliminating the corresponding pixels.
    - Update the user selection matrix to reflect changes in the image dimensions.

```
private Picture removeColumns(Picture picture, int removeColumns, boolean
isVertical) {
    for (int i = 0; i < removeColumns; i++) {
        double[][] energy = energyCalculator.computeSelectedEnergy(picture,
userSelection, isVertical);
        int[] seam = findSeam(energy);
        picture = removeSeam(picture, seam, isVertical);
        if (isVertical) {
            verticalSeams.push(seam);
        } else {
            horizontalSeams.push(seam);
        }
    }
    return picture;
}

private Picture removeSeam(Picture picture, int[] seam, boolean isVertical)
{
    Picture newPicture = new ArrayPicture(picture.getWidth() - 1,
picture.getHeight());
    for (int y = 0; y < picture.getHeight(); y++) {
        int i = 0;
```

```
            for (int x = 0; x < picture.getWidth(); x++) {
                if (x != seam[y]) {
                    newPicture.set(i++, y, picture.get(x, y));
                }
            }
        }
        // Update user selection area
        userSelection = updateUserSelection(userSelection, seam, isVertical,
newPicture);
        return newPicture;
    }

    private int[][] updateUserSelection(int[][] userSelection, int[] seam,
boolean isVertical, Picture newPicture) {
        int[][] newUserSelection;
        if (isVertical) {
            newUserSelection = new int[newPicture.getWidth()]
[newPicture.getHeight()];
            for (int y = 0; y < newPicture.getHeight(); y++) {
                int i = 0;
                for (int x = 0; x < newPicture.getWidth(); x++) {
                    if (x != seam[y]) {
                        newUserSelection[i++][y] = userSelection[x][y];
                    }
                }
            }
        } else {
            newUserSelection = new int[newPicture.getHeight()]
[newPicture.getWidth()];
            for (int y = 0; y < newPicture.getWidth(); y++) {
                int i = 0;
                for (int x = 0; x < newPicture.getHeight(); x++) {
                    if (y != seam[x]) {
                        newUserSelection[i++][y] = userSelection[x][y];
                    }
                }
            }
        }
        return newUserSelection;
    }
```

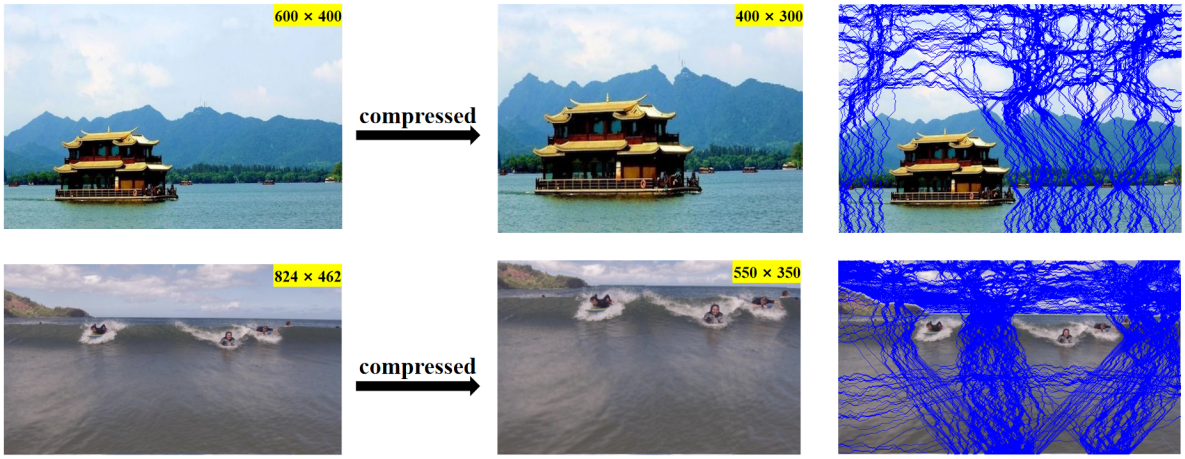
3. **Finalization**:
    - Convert the modified image to a buffered image for display or further processing.

```
for (int x = 0; x < picture.getWidth(); x++) {
for (int y = 0; y < picture.getHeight(); y++) {
bufferedPicture.set(x, y, picture.get(x, y));
    }
}
return bufferedPicture;
```

3. Handling User Selections

Users may specify regions of the image to protect or remove during resizing. This is achieved through the `userSelection` matrix, where each element indicates whether a pixel should be protected (value `1`) or removed (value `-1`). This matrix is updated dynamically during the seam removal process to ensure that user selections are respected.

4.Example



Part 4: Image Enlargement Algorithm

The `SeamCarverEnlarge` class is designed to enlarge images by adding seams with the lowest energy. This process ensures minimal disruption to the visual content of the image. The enlargement involves a series of steps that focus on energy calculation, seam identification, and image reconstruction.

1. Initialization

- The algorithm begins by initializing structures to track the vertical and horizontal seams that will be added to the image.
- A set is also initialized to track unique energy levels, which helps in avoiding the addition of seams with similar energy values, thereby maintaining the visual quality of the image.

2. Energy Calculation

- The initial energy matrix of the image is computed. This matrix represents the "energy" or importance of each pixel in the image, with lower energy values indicating less important pixels that are more suitable for seam insertion.

3. Seam Addition

- Seams are added iteratively to the image. The number of seams to be added is specified by the user in terms of columns and rows.
- For each seam addition, the algorithm identifies the path of lowest energy and updates the image by inserting a new seam along this path. This ensures that the added seams blend seamlessly with the existing content.

```
private Picture addSeam(Picture picture, int[] seam) {
    Picture newPicture = new ArrayPicture(picture.getWidth() + 1,
picture.getHeight());
    for (int y = 0; y < picture.getHeight(); y++) {
        int i = 0;
        for (int x = 0; x < picture.getWidth(); x++) {
            newPicture.set(i++, y, picture.get(x, y));
            if (x == seam[y]) {
                newPicture.set(i++, y, picture.get(x, y));
            }
        }
    }
    return newPicture;
}
```

4. Energy Matrix Update

- After each seam is added, the energy matrix is updated to reflect the changes in the image structure. This involves recalculating the energy values around the newly added seams to ensure that subsequent seams are added in the optimal locations.
- A boost in energy is applied to the newly created seams to prevent them from being selected again, which helps maintain the integrity of the added seams.
- By adding the energy boost, the seam that is previously selected have a lower probability of being selected again, in order to reduce the possibility of duplicate selections adding the same seam.

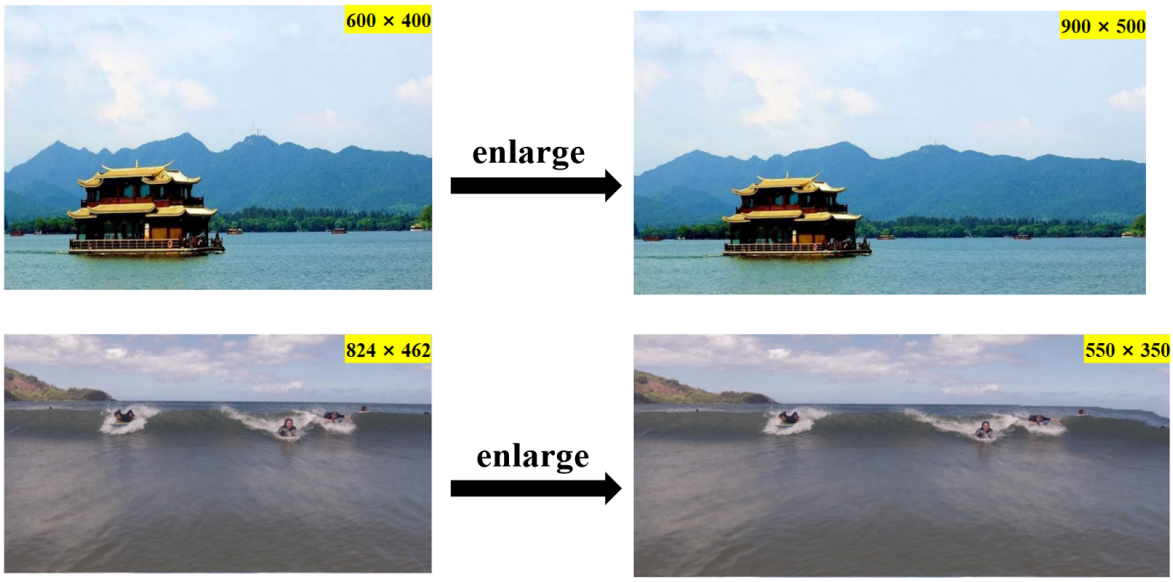
```
private double[][] updateEnergy(double[][] energy, int[] seam) {
    int ENERGY_BOOST = 20020;
    int width = energy.length;
    int height = energy[0].length;
    double[][] newEnergy = new double[width + 1][height];
    for (int y = 0; y < height; y++) {
        int i = 0;
        for (int x = 0; x < width; x++) {
            if (x != seam[y]) {
                newEnergy[i++][y] = energy[x][y];
            } else {
                newEnergy[i++][y] = energy[x][y] + ENERGY_BOOST;
                newEnergy[i++][y] = energy[x][y] + ENERGY_BOOST;
            }
        }
    }
    return newEnergy;
}
```

5. Image Reconstruction

- The final step involves reconstructing the image by adding the calculated seams. This process ensures that the image's dimensions are enlarged while preserving the overall visual coherence.
- The algorithm handles the duplication of pixels along the seam paths to create a smooth transition and avoid noticeable artifacts.

This version should be correctly formatted for Markdown. Let me know if there are any additional changes or sections you'd like help with!

6. Example



The image enlargement algorithm implemented in the `SeamCarverEnlarge` class efficiently increases the dimensions of an image by intelligently adding seams in regions of low energy. This method preserves the important visual features of the image while expanding its size, making it suitable for applications that require image resizing without significant loss of detail.

Part 5: Image Processing GUI

Introduction

This report describes the development process of a Java-based Image Processing GUI application. The application provides functionalities for loading, displaying, energy computation, resizing, enlarging images, and protecting or deleting specific areas. The application uses the Seam Carving algorithm to achieve intelligent image resizing and offers a user-friendly graphical interface for easy interaction.

Functionality Overview

File Operations

- **File Button:** A dropdown menu with "Input" and "Output" options allows users to select an image file for processing or choose where to save the processed image.
- **Input Option:** Enables users to select an image file from the file system for processing.
- **Output Option:** Allows users to choose a directory to save the processed image.

Image Display

- **Image Panel:** Displays the selected image and provides information about the image size.
- **Scroll Pane:** Contains the image panel and allows users to scroll to view large images.

Energy Computation

- **Energy Button:** Computes and displays the energy map of the image.

Image Resizing

- **Shrink Button:** Allows users to input target width and height to shrink the image to the specified dimensions.
- **Enlarge Button:** Allows users to input target width and height to enlarge the image to the specified dimensions.

Area Protection and Deletion

- **Protection Mode:** Enables users to select areas in the image to protect, which will not be altered during resizing.
- **Delete Mode:** Allows users to select areas in the image to delete, which will be prioritized for removal during resizing.

Technical Details

Image Processing

Image processing is implemented using the Seam Carving algorithm, with methods from the `SeamCarvingRun` class handling energy computation, shrinking, and enlarging operations. This algorithm intelligently adjusts image size while preserving important content.

User Interaction

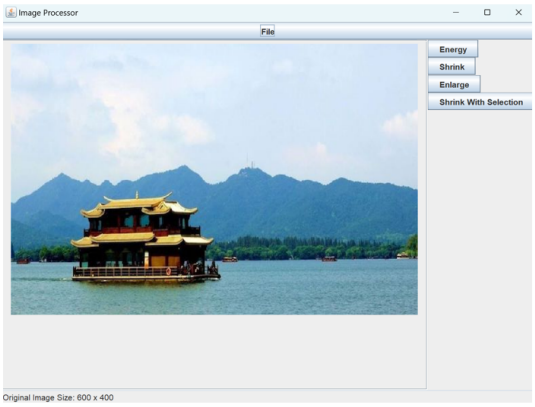
File selection is implemented using the `JFileChooser` component, user input for target dimensions using `JOptionPane`, and area selection and protection using `MouseAdapter`.

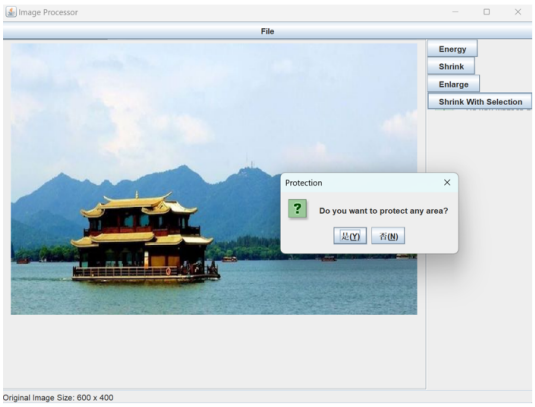
Interface Layout

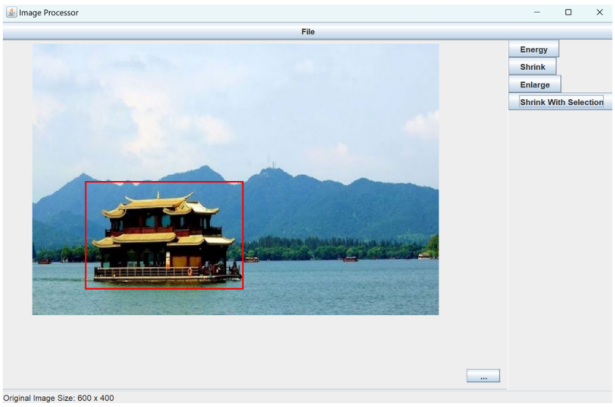
The application uses `JFrame` as the main frame, `JPanel` for image display, `JScrollPane` for scrollable view, and `BoxLayout` for button layout. The interface is designed to be simple and user-friendly, facilitating easy operation.

GUI'S Conclusions

This Image Processing GUI application provides convenient image resizing and editing functionalities through an intuitive user interface and intelligent image processing algorithms. The project demonstrates the potential of the Seam Carving algorithm in practical applications and offers valuable learning and usage opportunities.







Conclusion

The dynamic programming approach efficiently calculates the shortest path through the energy matrix, ensuring that the seam with the least energy is identified. This seam is then removed to resize the image with minimal impact on its visual quality.

Advantages:

- **Preservation of Important Features**: The algorithm maintains the visual integrity of key image regions.
- **User Control**: Users can protect or prioritize specific areas of the image, enhancing customization.
- **Versatility**: The method works for both resizing and enlargement, making it adaptable to various applications.

Limitations:

- **Computational Complexity**: The process can be computationally intensive, especially for large images.
- **Artifacts in Complex Images**: In highly textured or detailed images, the algorithm might introduce artifacts or distortions.

Overall, our seam carving implementation provides a powerful tool for content-aware image resizing, balancing the need for size adjustment with the preservation of visual quality.