Project: intelligent scissor

Team= T185

Members:

1- Omar Khaled Nasr Askr (20191700409)
2- Amr Ahmed Samy abdelfatah (20191700422)

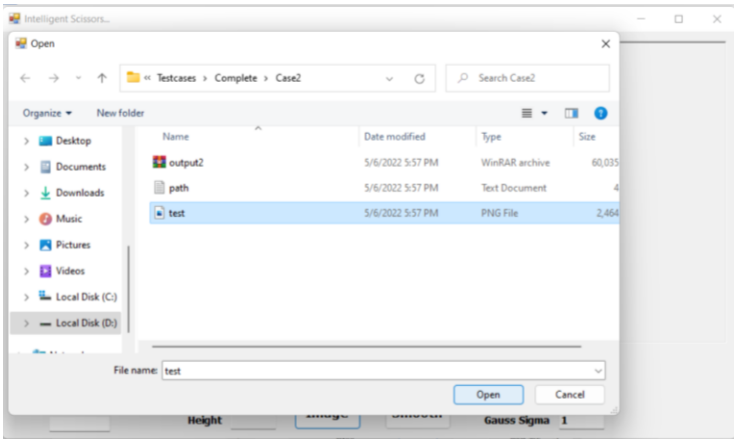
Project documentation:

Intro:

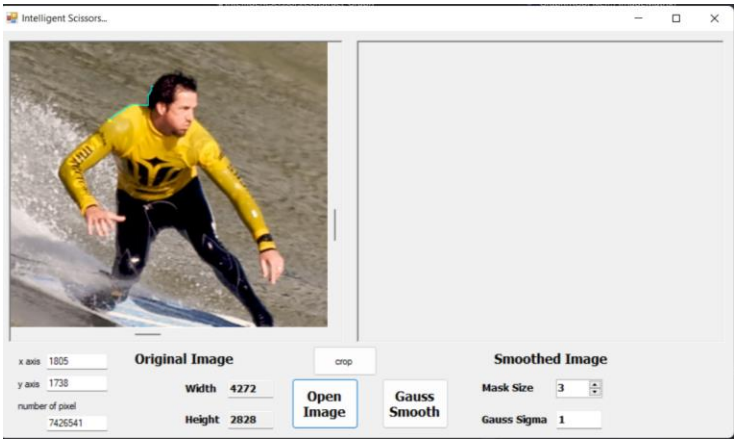
This project is intelligent scissors which use to get peace from a picture and cut it.

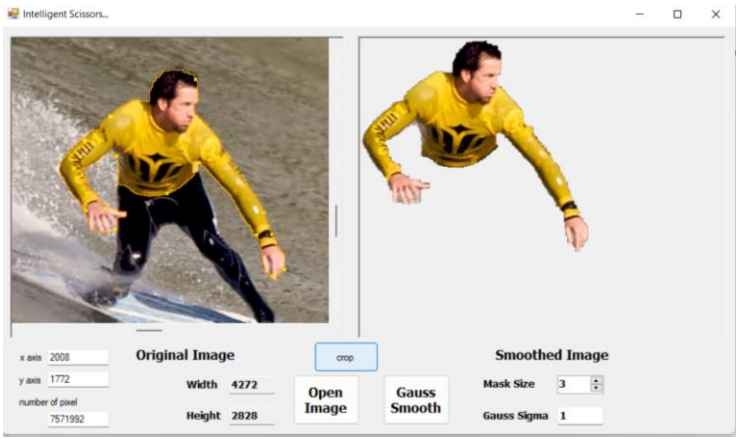
How to use:

1-click open image and select image that you want to cut from it.



2-put anchor point than move mouse on edit of thing that you want to cut and you will notice that live line will draw until you end select than click crop.





3-you will see picture you crop on another side.

Technical part:

Idea of program:

we use 2D array to represent 2D image, the image consist of small units called pixels ordered in columns, and each column contains the same number of pixels, and each pixel has a color that is combination of (red , green , blue),that is the RGB color model.

Now we want to select an object from an image ,that we will do using the array construct above.

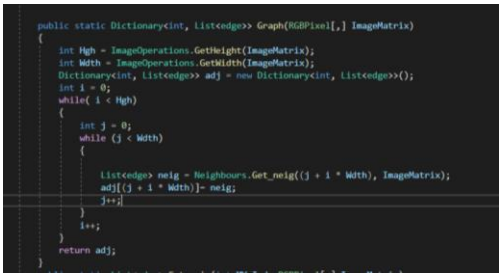
So we will select object by go on his boundary ,by the change of color of each pixel we will know the boundary of object.

If I have two pixel have white color so I cant exit an edge between them because no change in color between them ,no cost between them so we make graph to save value of first pixel and secand pixel and cost between them.

Lets return to our idea we want to calssify edges of the picture in the graph ,if we gor from white to white we will have low cost and if go from black and white we will have hight cost so we make shortest path that go from node to another on same color because it has low cost

There for we want to keep track with biggest change in color to get around the image not inside or outside,

And that code of graph:



```
public static Dictionary<int, List<edge>> Graph(RGBPixel[,] ImageMatrix)
{
    int Hgh = ImageOperations.GetHeight(ImageMatrix);
    int Wdth = ImageOperations.GetWidth(ImageMatrix);
    Dictionary<int, List<edge>> adj = new Dictionary<int, List<edge>>();
    int i = 0;
    while( i < Hgh)
    {
        int j = 0;
        while (j < Wdth)
        {

            List<edge> neig = Neighbours.Get_neig((j + i * Wdth), ImageMatrix);
            adj[(j + i * Wdth)]= neig;
            j++;
        }
        i++;
    }
    return adj;
}
```

Make dictionary to save each node and his neighbors and what I mean by neighbors it is four nodes around it (up , right , down , left)

Edge is the node that has high density difference in RGB.

```
public static List<edge> Get_neig(int MY_Ind, RGBPixel[,] imgmatrix)
{

    int Wdth = ImageOperations.GetWidth(imgmatrix);
    var free = Free_X_Y(MY_Ind, Wdth);
    int X_axis = (int)free.X, Y_axis = (int)free.Y;
    var Energy_Weight = ImageOperations.CalculatePixelEnergies(X_axis, Y_axis, imgmatrix);
    int Hgt = ImageOperations.GetHeight(imgmatrix);
    List<edge> neghlist = new List<edge>();


    if (X_axis < Wdth - 1)
    {
        if (Energy_Weight.X == 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis + 1) + Y_axis * Wdth), 1000000000000000));
        else if (Energy_Weight.X != 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis + 1) + Y_axis * Wdth), 1 / (Energy_Weight.X)));
    }
    if (Y_axis < Hgt - 1)
    {
        if (Energy_Weight.Y == 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis) + (Y_axis + 1) * Wdth), 1000000000000000));
        else if (Energy_Weight.Y != 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis) + (Y_axis + 1) * Wdth), 1 / (Energy_Weight.Y)));
    }
    if (X_axis > 0)
    {
        Energy_Weight = ImageOperations.CalculatePixelEnergies(X_axis - 1, Y_axis, imgmatrix);

        if (Energy_Weight.X == 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis - 1) + Y_axis * Wdth), 1000000000000000));
        else if (Energy_Weight.X != 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis - 1) + Y_axis * Wdth), 1 / (Energy_Weight.X)));
    }

    if (Y_axis > 0)
    {
        Energy_Weight = ImageOperations.CalculatePixelEnergies(X_axis, Y_axis - 1, imgmatrix);
        if (Energy_Weight.Y == 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis) + (Y_axis - 1) * Wdth), 1000000000000000));
        else if (Energy_Weight.Y != 0)
            neghlist.Add(new edge(MY_Ind, ((X_axis) + (Y_axis - 1) * Wdth), 1 / (Energy_Weight.Y)));
    }


    return neghlist;
}
}
```

We use function get_neig to get four node around the node that we want (right , bottom , down , left) and check in every one if it has this node we make edge between them and pot cost = 1/cost and id there is no node beside it in any direction we make edge but has max number cost.

But there will be problem that every node will have list of 4 and the number of pixel is very big so we need to dijkstra algorithm to find shortest path between nodes in non negative weight

So we use data structure called priority-queue we called it in code data structure ,this it job to maintain set of elements ,each element has value called key and put element with minimum value on top of the array similar to binary tree and each element represent node in tree

```
public void push(edge Edge)
 {
     tree.Add(Edge);
     repositionup(tree.Count - 1);
 }
```

This first function in queue and it response to add element in node and rearrange elements that top element has minimum value by this function :

```
private void repositionup(int edge)
      {
          if (tree[edge].weight >= tree[parent_edge(edge)].weight)
              return;
          if (edge == 0)
              return;
          else
          {
              edge temp = tree[parent_edge(edge)];
              tree[parent_edge(edge)] = tree[edge];
              tree[edge] = temp;
              repositionup(parent_edge(edge));
          }
      }
```

The second function is:

```
public edge Pop()
      {
          edge temp = tree[0];
          tree[0] = tree[tree.Count - 1];
          tree.RemoveAt(tree.Count - 1);
          repositiondown(0);
          return temp;
      }
```

And it response to delete the root of the tree and remove last element in array this by function:

```
private void repositiondown(int edge)
      {

          if (right_edge(edge) >= tree.Count)
          {
              if (left_edge(edge) < tree.Count) {
                  if (tree[left_edge(edge)].weight >= tree[edge].weight)
                            return;
              }
          }


          if (left_edge(edge) < tree.Count && tree[left_edge(edge)].weight >=
tree[edge].weight )
          {
              if (right_edge(edge) < tree.Count && tree[right_edge(edge)].weight >=
tree[edge].weight)
```

```
                {
                    return;
                }
            }
            if (left_edge(edge) >= tree.Count)
                    return;
            if (right_edge(edge) < tree.Count && tree[right_edge(edge)].weight <=
tree[left_edge(edge)].weight)
            {
                edge temp = tree[right_edge(edge)];
                tree[right_edge(edge)] = tree[edge];
                tree[edge] = temp;
                repositiondown(right_edge(edge));
            }

            else
            {
                edge temp = tree[left_edge(edge)];
                tree[left_edge(edge)] = tree[edge];
                tree[edge] = temp;
                repositiondown(left_edge(edge));
            }
        }
```

Third function in datastructure is empty to return Boolean if tree empty or not :

```
public bool empty()
        {
            if (tree.Count == 0)
                return true;
            return false;
        }
```

Lets go back to dijkstra algorithm:it make shortest path from source node and all other destination ,we make overloaded Dijkstra algorithm

One take source and destination and imagematrix and the second take source and imagematrix only.

This is one the way we go from source node to all other destination in the the edge of node with minimum cost, now we will relax the destination that mean that act that node is not exist,we got number of shortest paths and we get the minimum path from remaining ones and do the last two steps and return the array to list called path_list

```
public static List<int> Dijkstra(int Source, RGBPixel[,] ImageMatrix)
        {
            List<double> min_distance = new List<double>();
            List<int> sourceofcurrentnode = new List<int>();

            int Width = ImageOperations.GetWidth(ImageMatrix);
            int Height = ImageOperations.GetHeight(ImageMatrix);
            int nodes_number = Width * Height;
            min_distance = Enumerable.Repeat(Max_Value, nodes_number).ToList();
            sourceofcurrentnode = Enumerable.Repeat(-1, nodes_number).ToList();

            Datastructure ds = new Datastructure();
```

```
            ds.Push(new edge(-1, Source, 0));

            while (true)
            {
                if (ds.empty())
                {
                    break;
                }

                edge topedge = ds.Top();
                ds.Pop();
                List<edge> neibours = neighbours.Get_neig(topedge.To, ImageMatrix);
                if (topedge.weight >= min_distance[topedge.To])
                    continue;

                min_distance[topedge.To] = topedge.weight;
                sourceofcurrentnode[topedge.To] = topedge.From;

                int i = 0;
                while ( i < neibours.Count)
                {
                    edge edg_h = neibours[i];

                    if (min_distance[edg_h.To] > min_distance[edg_h.From] + edg_h.weight)
                    {
                        edg_h.weight = min_distance[edg_h.From] + edg_h.weight;
                        ds.Push(edg_h);
                    }
                    i++;
                }
            }
```

Then we used backtracking technique to help us to draw the path

(shortestpath function to get nodes that we must connect together)

And shortpath list include the points that we should connect together .

```
public static List<Point> shortestpath(List<int> prvlist, int dst, int width_matrix)
        {
            List<Point> shortpath = new List<Point>();

            Stack<int> revesrsepath = new Stack<int>();

            revesrsepath.Push(dst);

            int prev;

            prev = prvlist[dst];
            while (true)
            {
                if (prev == -1)
                {
                    break;
                }
                revesrsepath.Push(prev);
                prev = prvlist[prev];
            }
```

```
            while (true)
            {
                if (revesrsepath.Count == 0)
                {
                    break;
                }
                var n = functions.make_2d(revesrsepath.Pop(), width_matrix);
                Point p = new Point((int)n.X, (int)n.Y);
                shortpath.Add(p);
            }
            return shortpath;
        }
```

DFS algorithm run about points on borders of object that we select and stop when face with non valid pixel.

```
private static void dfs(Vector2D s_point)
        {
            Stack<Vector2D> dfs_heab = new Stack<Vector2D>();
            dfs_heab.Push(s_point);
            while (maax( dfs_heab.Count , 0))
            {
                Vector2D current = dfs_heab.Pop();
                if (functions.Vaild_Pixel((int)current.X, (int)current.Y, simg))
                {
                    if (!simg[(int)current.Y, (int)current.X].block)
                    {
                        simg[(int)current.Y, (int)current.X].green = 240;
                        simg[(int)current.Y, (int)current.X].red = 240;
                        simg[(int)current.Y, (int)current.X].blue = 240;
                        simg[(int)current.Y, (int)current.X].block = true;

                        Vector2D vec1 = new Vector2D();
                        Vector2D vec2 = new Vector2D();
                        Vector2D vec3 = new Vector2D();
                        Vector2D vec4 = new Vector2D();

                        vec1.X = current.X;
                        vec2.X = current.X;
                        vec3.X = current.X + 1;
                        vec4.X = current.X - 1;
                        vec1.Y = current.Y + 1;
                        vec2.Y = current.Y - 1;
                        vec3.Y = current.Y;
                        vec4.Y = current.Y;

                        dfs_heab.Push(vec1);
                        dfs_heab.Push(vec2);
                        dfs_heab.Push(vec3);
                        dfs_heab.Push(vec4);
                    }
```

```
            }

        }
```

Then path of anchor

```
        public static List<Point> pathofanchor()
        {
            int freeze = 0;
            List<Point> free_path = new List<Point>();
            for (int i = 0; i < Cut_wire.Count; i++)
            {
                if (cool_Time[i] > 1)
                {
                    if(new_rope_color[i] >= 8)
                        freeze = i;
                }
            }
            int l = 0;
            while (l < freeze)
            {
                free_path.Add(Cut_wire[l]);
                l++;
            }
            return free_path;
        }
```

```
public static void Update(List<Point> Path, double Ctime)
        {
            int wire_size = Cut_wire.Count;
            int path_length = Path.Count;


            int i = 0;
            int j = 0;
            while (j < wire_size && i < path_length)
            {
                if (Path[i] != Cut_wire[j])
                {

                    Cut_wire[j] = Path[i];
                    cool_Time[j] = 0;
                    new_rope_color[j] = 0;
                }
                else if(Path[i] == Cut_wire[j])
                {
                    cool_Time[j] += Ctime;
                    new_rope_color[j] += 1;

                }
                i++; j++;
            }
```

```
        for (int z = 0; z < wire_size; z++)
        {
            cool_Time[z] = 0;
            new_rope_color[z] = 0;
            Cut_wire[z] = new Point(-1, -1);

        }

        for (int k=0;k< path_length;k++)
        {
            cool_Time.Add(0);
            new_rope_color.Add(0);
            Cut_wire.Add(Path[k]);
        }


    }
```