INTELLIGENT SCISSORS

TEAM NO. 1
TEAM MEMBERS:

Hassan Adham Hassan

Dina Yehia Hussein Riad

Rodaina Hesham Sokkar

Hazem Youssef El Sebaay

Abdullah Mohammed Abd El-Ghany

ABSTRACT
A program that can select and crop any part of a picture with speed efficient and very high accuracy for selecting edges.

Function: BuildGraph O(N^2)

```
/* Receives and array of pixels.
* Building Graph using Adjacency List.
* An ImageGraph array is defined of size N^2, each index has the node ID and each node has a
list of edges.
* The list is built for each node to hold its neighboring nodes (Left,Right,Up,Down) and the
weights between them and the current node.
* The code is classified according the node position in the image. (Node with 2 neighbors, node
with 3 neighbors and node with 4 neighbors).
* (For current node)
* Add to its list the edge weight between current node and right node (If exists) and between
bottom node (If exists).
* For the right node, add to its list the current node as its left node. (If exists).
* For the bottom node, add to its list the current node as its above node. (If exists).
* At the end, the ImageGraph array is full all connections between all nodes.
*/
```

```
    public struct Edge
    {
        public int p;
        public double w;
    }

    public static List<Edge>[] ImageGraph;
    public static void BuildGraph(RGBPixel[,] image)//O(N^2)
    {
        int width = GetWidth(image);
        int height = GetHeight(image);
        Vector2D temp = new Vector2D();
        ImageGraph = new List<Edge>[width * height];
        for (int k = 0; k < width * height; k++) //O(N^2)
        {
            //Create a list of edges for each node in the image.
            ImageGraph[k] = new List<Edge>();
        }
        for (int i = 0; i < height; i++)//O(N)
        {
            for (int j = 0; j < width; j++)//O(N)
            {
                //If node is not in last column nor last row,
                //i.e Must have both right and bottom nodes.
                if (i != height - 1 && j != width - 1)
                {
                    temp = CalculatePixelEnergies(j, i, image);
                    Edge e = new Edge();
                    e.p = i * width + j + 1;
                    if (1 / temp.X == double.PositiveInfinity)
                        e.w = 1E+16;
                    else
                        e.w = 1 / temp.X;
                    ImageGraph[i * width + j].Add(e);
                    e.p = (i + 1) * width + j;
                    if (1 / temp.Y == double.PositiveInfinity)
                        e.w = 1E+16;
```

```
        else
            e.w = 1 / temp.Y;
        ImageGraph[i * width + j].Add(e);
        e.p = i * width + j;
        if (1 / temp.X == double.PositiveInfinity)
            e.w = 1E+16;
        else
            e.w = 1 / temp.X;
        ImageGraph[i * width + j + 1].Add(e);
        e.p = i * width + j;
        if (1 / temp.Y == double.PositiveInfinity)
            e.w = 1E+16;
        else
            e.w = 1 / temp.Y;
        ImageGraph[(i + 1) * width + j].Add(e);
    }
    //Current node is in last row but not in last column.
    //i.e Doesn't have bottom node.
    else if (i == height - 1 && j != width - 1)
    {
        temp = CalculatePixelEnergies(j, i, image);
        Edge e = new Edge();
        e.p = i * width + j + 1;
        if (1 / temp.X == double.PositiveInfinity)
            e.w = 1E+16;
        else
            e.w = 1 / temp.X;
        ImageGraph[i * width + j].Add(e);
        e.p = i * width + j;
        if (1 / temp.X == double.PositiveInfinity)
            e.w = 1E+16;
        else
            e.w = 1 / temp.X;
        ImageGraph[i * width + j + 1].Add(e);
    }
    //Current node is in last column but not in last row.
    //i.e Doesn't have right node.
    else if (i != height - 1 && j == width - 1)
    {
        temp = CalculatePixelEnergies(j, i, image);
        Edge e = new Edge();
        e.p = (i + 1) * width + j;
        if (1 / temp.Y == double.PositiveInfinity)
            e.w = 1E+16;
        else
            e.w = 1 / temp.Y;
        ImageGraph[i * width + j].Add(e);
        e.p = i * width + j;
        if (1 / temp.Y == double.PositiveInfinity)
            e.w = 1E+16;
        else
            e.w = 1 / temp.Y;
        ImageGraph[(i + 1) * width + j].Add(e);
    }
```

```
        }
    }
}
```

Class: Heap

```
class heap
{
    public int size, last;
    private pair[] arr;
    public heap(int n)
    {
        arr = new pair[n];
        size = n;
        last = 1;
    }
```

Function: add O(logN)

```
/*
* Check the size of the array of nodes, if the size is not enough then double it.
* Add new node at the end of the array.
* Compare each node with its parent (where parent_index = node_index/2).
* If(value of the node < its parent), then swap them.
*/
```

```
public void add(double a, int b,int c)//O(logN).
{
    if (last == 0)
        last++;
    //Check if the array size can have anymore elements or not.
    if (last == size)
    {
        //Double the size of the array.
        Array.Resize(ref arr, arr.Length * 2);
        //Set the size varialbe to the new size of the array.
        size = arr.Length;
    }
    //Put the new element after the last element in the array.
    arr[last] = new pair(a, b, c);
    int i = last;
    //Compare the new element with it's parent and swap them to keep it minimum
     tree.
    while (i != 1 && arr[i].first < arr[i / 2].first)//O(log(N)).
    {
        pair x = arr[i / 2];
        arr[i / 2] = arr[i];
        arr[i] = x;
        i /= 2;
    }
    last++;
}
```

Function: getMin O(logN)

```
/*
* First node in the array is the minimum node.
* After virtually deleting the minimum node, we check if the array has other nodes.
* If there is still other nodes, put the last node as the first node in the array.
* Update the position of the first node, by looking at both {left(index*2) and
right(index*2+1)} nodes. Then swap this node with minimum of (left and right).
* When no children are left, or when the current node is already smaller than it's right and
left, the loop breaks.
* Finally return minimum node in the array.
*/
```

```
    public pair getmin()//O(LogN).
    {
        pair x = arr[1],y;
        last--;
        //Update the tree if it still have any elements.
        if (last != 0)
        {
            //Put the last element in the first place.
            arr[1] = arr[last];
            int i = 1;
            //Update the i-th element with it's children.
            while (i < last)//O(log(N)).
            {
                //Finding minimum between i and it's children to update the tree.

                //Check if valid right and left children.
                if ((i * 2) + 1 < last)
                {
                    if (arr[i * 2].first < arr[(i * 2) + 1].first &&
                          arr[i * 2].first < arr[i].first)
                    {
                        y = arr[i * 2];
                        arr[i * 2] = arr[i];
                        arr[i] = y;
                        i *= 2;
                    }
                    else if (arr[i * 2].first >= arr[(i * 2) + 1].first &&
                    arr[(i * 2) + 1].first < arr[i].first)
                    {
                        y = arr[(i * 2) + 1];
                        arr[(i * 2) + 1] = arr[i];
                        arr[i] = y;
                        i *= 2;
                        i++;
                    }
                    else
                        break;
                }
                //Check if valid left child.
                else if (i*2<last)
```

```
                {
                    if (arr[i * 2].first < arr[i].first)
                    {
                        y = arr[i * 2];
                        arr[i * 2] = arr[i];
                        arr[i] = y;
                        i *= 2;
                    }
                    else
                        break;
                }
                //This node has no children.
                else
                {
                    break;
                }
            }
        }
        return x;
    }

        public bool empty()
        {
            if (last == 0)
                return true;
            return false;
        }
    }
```

Class: Pair

```
class pair
{
    public double first;
    public int second;
    public pair(double a, int b)
    {
        first = a;
        second = b;
    }
}
```

Function: ShortestReach O(E log(N))

```
/*
* It builds an array to save the shortest paths from source to each node.
* Another array is defined to hold all parents of each node.
* ---------------------------------------------------------------------
* Setting an initial value for each node with infinity, then add in the priority queue the
source node with a path to itself equal zero.
* Loop keeps iterating until no nodes are yet found.
* Check the paths value with an already saved value in the array, then it updates the parents.
* Then it starts to add the connected nodes to it (value of path = current node + edge weight).
* The function returns the array of parents, to track the paths of each node.
*/
```

```
    public static int[] shortestReach(int n, List<Edge>[] edges, int s)
    {
        /*
            *  E is the number of edges.
            *  N is the number of nodes.
        */
        //Array holds the path value from source to each node.
        double[] arr = new double[n + 1];
        int[] pa = new int[n + 1];
        for (int i = 0; i <= n; i++)//O(N)
        {
            //Set path value from source to each node as high value.
            arr[i] = 1.7E308;
            pa[i] = -1;
        }
        heap h = new heap(n);
        //Add the source node path value equals 0
        h.add(0,s,s);//O(log(N)).
        while (!h.empty())//O(E log(N)).
        {
            //Get the minimum value and remove it from the heap.
            pair x = h.getmin();
            //Check if the new path value is better than the one we already have.
            if (arr[x.second] > x.first)
            {
                //Update the path value.
                arr[x.second] = x.first;
                pa[x.second] = x.p;
                //Loop over edges connected to the node we have now .
                for (int i = 0; i < edges[x.second].Count; i++)//O(E).
                {
                    //Check if the new path value is better than the one we already
                     have.
                    if (arr[edges[x.second][i].p] > x.first + edges[x.second][i].w)
                    {
                        h.add(x.first + edges[x.second][i].w,
                          edges[x.second][i].p,x.second);//O(log(N)).
                    }
                }
```

```
        }
    }
    return pa;
}
```

Function: line   O(N)

```
/*
* Looping on the array of parents, until the node is equal to its parent.
* Inside the loop, each node is assigned with the value of its parent.
*/
```

```
public static int[] line(int d,int []par)//O(N).
{
    //Create list to hold the nodes in the shortest path from source to
     destination.
    List<int> l = new List<int>();
    // Start first time from destination and loop till it equals the source
     node.
    while (d != par[d])//O(N).
    {
        l.Add(d);
        d = par[d];
    }
    l.Add(d);
    int[] a = new int[l.Count];
    for (int i = 0; i < a.Length; i++)//O(N)
    {
        //Copy the nodes from the list to an array.
        a[i] = l[i];
    }
    return a;
}
```

Function: output   O(N^2)

Creates "output.txt" text file that represents the graph.

```
public static void output()//O(N^2).
{
    using (StreamWriter writetext = new StreamWriter("output.txt"))
    {
        string g = "The constructed graph" + Environment.NewLine;
        writetext.WriteLine(g);
        for (int i = 0; i < ImageGraph.Length; i++)//O(N^2).
        {
            string s = " The  index node" + i + Environment.NewLine +
                            "Edges" + Environment.NewLine;
            for (int j = 0; j < ImageGraph[i].Count; j++)//O(N^2).
            {
                if (ImageGraph[i][j].w == double.PositiveInfinity)
                    s += "edge from    " + i + "  To   " +
                     ImageGraph[i][j].p + "  With Weights  " + 1E+16 +
```

```
                        Environment.NewLine;
                    else
                        s += "edge from   " + i + "  To  " +
                          ImageGraph[i][j].p + "  With Weights  " +
                          ImageGraph[i][j].w + Environment.NewLine;
                }
                s += Environment.NewLine + Environment.NewLine;
                writetext.WriteLine(s);
            }
        }
    }
```

Function: outputShortestPath   O(N)

Creates "outputShortestPath.txt" text file that represents the shortest path.

```
    public static void outputShortestPath(Point[] arr, int source, Point
    sourcePoint, int destination, Point destintaionPoint)//O(N).
    {
        using (StreamWriter sw = new StreamWriter("shortestPath.txt"))
        {
            sw.WriteLine(" The Shortest path from Node  " + source + "at
                          position   " + sourcePoint.X + "  " + sourcePoint.Y);
            sw.WriteLine(" The Shortest path to Node  " + destination + "at
                          position   " + destintaionPoint.X + "  " +
                          destintaionPoint.Y);
            for (int i = arr.Length - 1; i >= 0; i--)//O(N).
            {
                sw.WriteLine("Node  " + arr[i] + " at position x " + arr[i].X
               + " at position y   " + arr[i].Y);
            }
        }
    }
```

Function: drawLine   O(N)

Draws a line for the shortest path from anchor point to destination.

```
    private void drawLine(Point[] arr, Color C, int S)//O(N)
    {
        //"arr" is an array of points holding the path points.
        Graphics g = pictureBox1.CreateGraphics();
        Rectangle r = new Rectangle();
        Pen pen = new Pen(C, S);
        pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;
        PaintEventArgs p = new PaintEventArgs(g, r);
        p.Graphics.DrawCurve(pen, arr); //O(N)
    }
```