# Intelligent Scissors Project

**TA: محمد رأفت**

**Team ID : 162**

**Team Members :**

1. ضحي احمد فتح الله
2. روان سامح حلمي
3. ايمان ايمن عبد المنعم

- **Graph Construction**

Constructing the graph in `Dictionary<int ,Dictionary<int,double>>` .The key of the first dictionary is the index for each vertex in the graph , the value is another dictionary that holds the index of neighbors in a list as a key and holds the wights of neighbors in a list of double as a vale for the second dictionary.

```
1 reference
public static Dictionary<int ,Dictionary<int,double>> Get_Graph(RGBPixel[,] ImageMatrix){

    Dictionary<int, Dictionary<int, double>> My_graph = new Dictionary<int, Dictionary<int, double>>();
    int height = ImageOperations.GetHeight(ImageMatrix);
    int width = ImageOperations.GetWidth(ImageMatrix);
    int parent;
    const double infinity = 10000000000000;
    Vertex adj;
    for (int i = 0; i < height; i++) // Row
    {

        for (int j = 0; j < width; j++) //Column
        {
            double weights;
            int indices;
            parent = (i * width)+j; //position
            Dictionary<int, double> pair = new Dictionary<int, double>();
            if (i == 0) // first row
            {
                if (j == 0)
                {

                    //bottom
                    adj.weight = ImageOperations.CalculatePixelEnergies(j, i, ImageMatrix).Y;
                    weights = 1 / adj.weight;
                    if(adj.weight == 0)
                    {
                        weights = infinity;

                    }
                    indices = ((i + 1) * width) + j;
                    pair.Add(indices, weights);
                    //right
```

Complexity : $O(N^2)$ .

- **Shortest Path**

  To get Shortest Path , we used Dijkstra Algorithm.

```
1 reference
public static  Dictionary<int, int> DisjkstraDistance(int src, int dist, Dictionary<int, Dictionary<int, double>> graphDict,RGBPixel[,] ImageMatrix)
{
    //graph output will be queue's input
    SimplePriorityQueue<int, double> priority_queue = new SimplePriorityQueue<int, double>();
    priority_queue.Enqueue(src, 0);

    //Dictionry of distances for each vertex
    Dictionary<int, double> distances = new Dictionary<int, double>...;
   //Dictionary of parent for each vertex
   // key-> child value->parent
    Dictionary<int, int> parent = new Dictionary<int, int>...;
    //   parent = Enumerable.Repeat(-1, size).ToArray();

    Dictionary<int, string> dequeued = new Dictionary<int, string>();
    while (!(priority_queue.Count == 0))
    {
        int value;
        string status;
        value = priority_queue.Dequeue();
        status = "black";
        dequeued.Add(value, status);
        //black(visited) dont visit agian
        //white(not visited ) weight infinity
        //grey not sure

        if (value == dist)
```

```
        if (value == dist)
        {
            break;
        }

       DijFile.WriteLine(dist + " Node: " + parent[dist] + " at position X = " + parent[dist]%width + ", and  position Y= " + parent[dist]/width);

        foreach (var neighbors in graphDict[value])
        {

            if (!dequeued.ContainsKey(neighbors.Key)) // check if it not black
            {
                if (priority_queue.Contains(neighbors.Key)) //if true it means that is grey
                {
                    //check if path is less than stored
                    if (distances[neighbors.Key] > neighbors.Value + distances[value])
                    {
                        //update in distance and parent
                        priority_queue.UpdatePriority(neighbors.Key, neighbors.Value + distances[value]);
                        distances[neighbors.Key] = neighbors.Value + distances[value];
                        parent[neighbors.Key] = value;
                    }
                }
                else // white
                {

                    //update value from infitinty
                    priority_queue.Enqueue(neighbors.Key, neighbors.Value + distances[value]);
                    distances.Add(neighbors.Key, neighbors.Value + distances[value]);
                    parent.Add(neighbors.Key, value);
                }
            }
        }
```

Complexity : O(E Log(v))