**Project: Intelligent scissor**

**Team (6)**

**Abdelrahman Hamdy El-Sayed Radwan**

**Khaled Mohammed**

**Mohammed Saeed**

**Ahmed Salah**

**Islam Magdy**

**Project documentation:**

**User guide:**

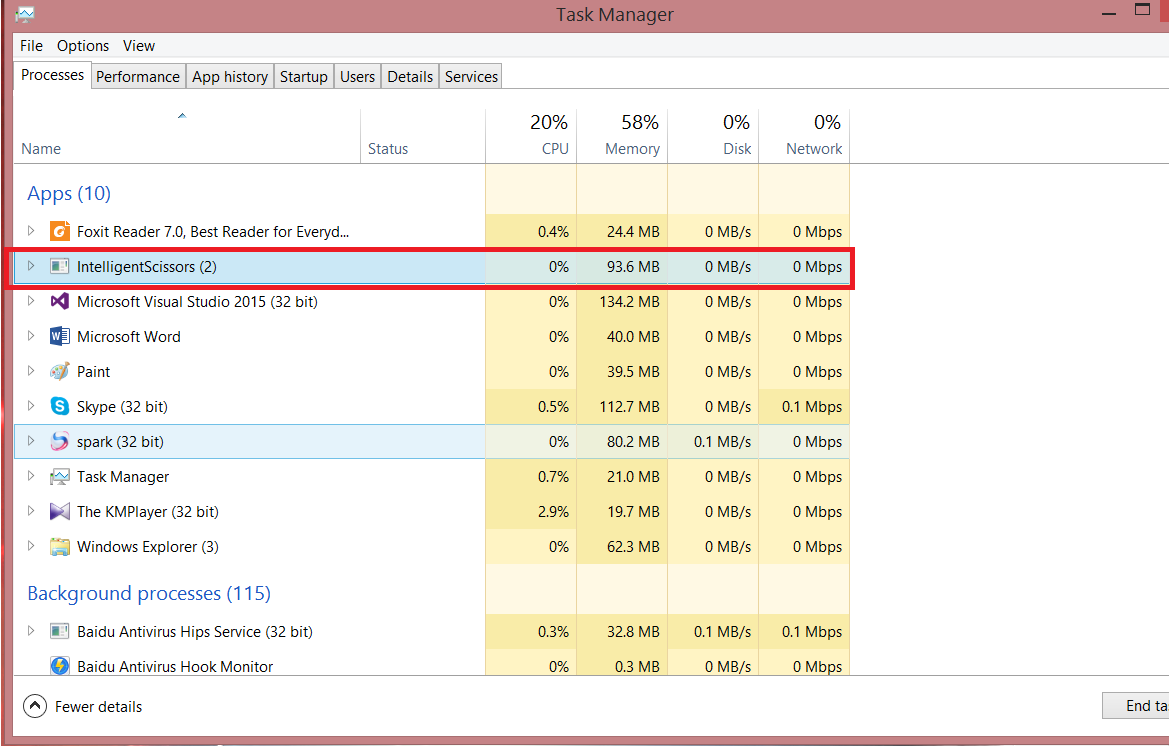
**Intro:**

This product is an intelligent scissors, it’s a selection tool, which can be used to select objects in an image to use them as separate objects, and it gives you the ability to automatically snap to the objects’ boundaries.

**Minimum requirements:**

To run this product you need to have:

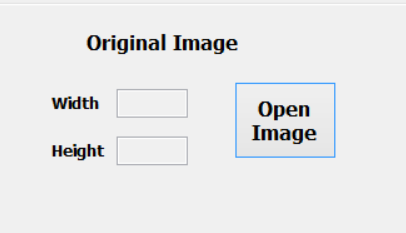
* PC
* 166 MHz processor
* 126 MB RAM
* 1 MB of free hard disk space



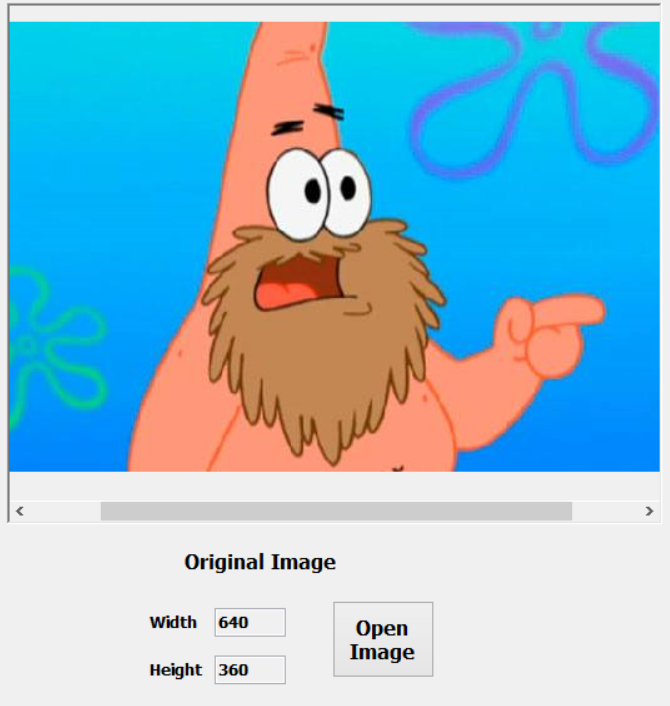
**It just costs you 93.6MB of memory to crop a 2.02MB 2496X1664 pixels picture.**

**How to use:**

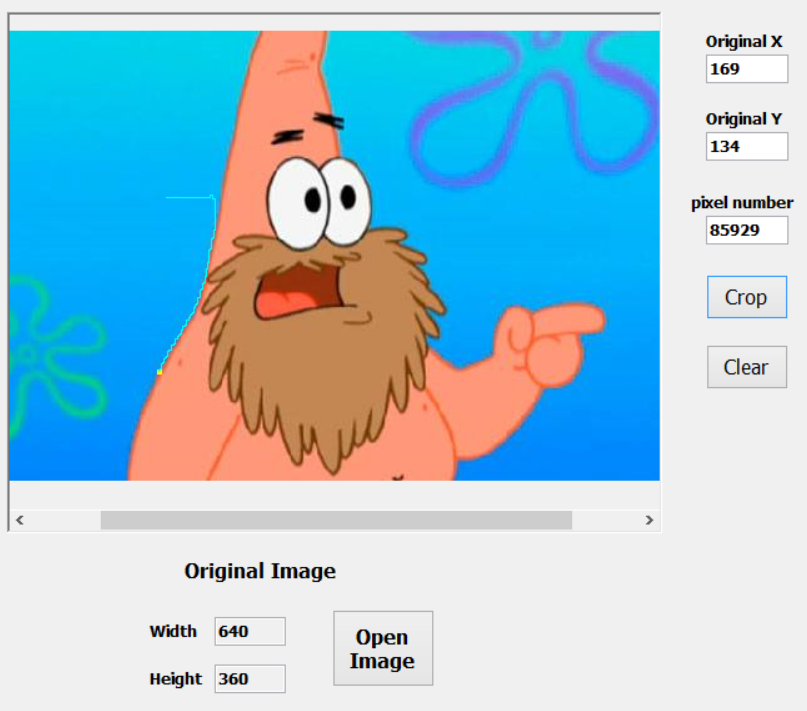
* **Open the image**



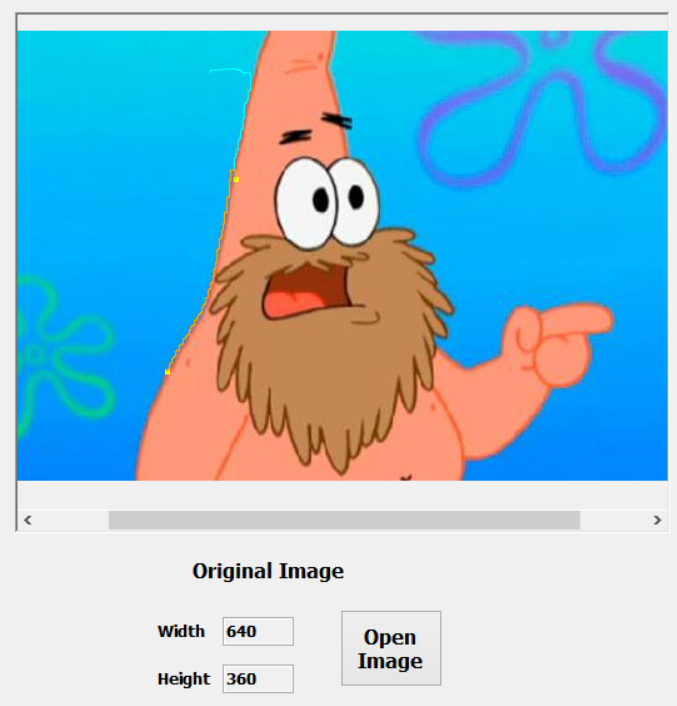
* The image will be open on the #, through which you can navigate using your mouse



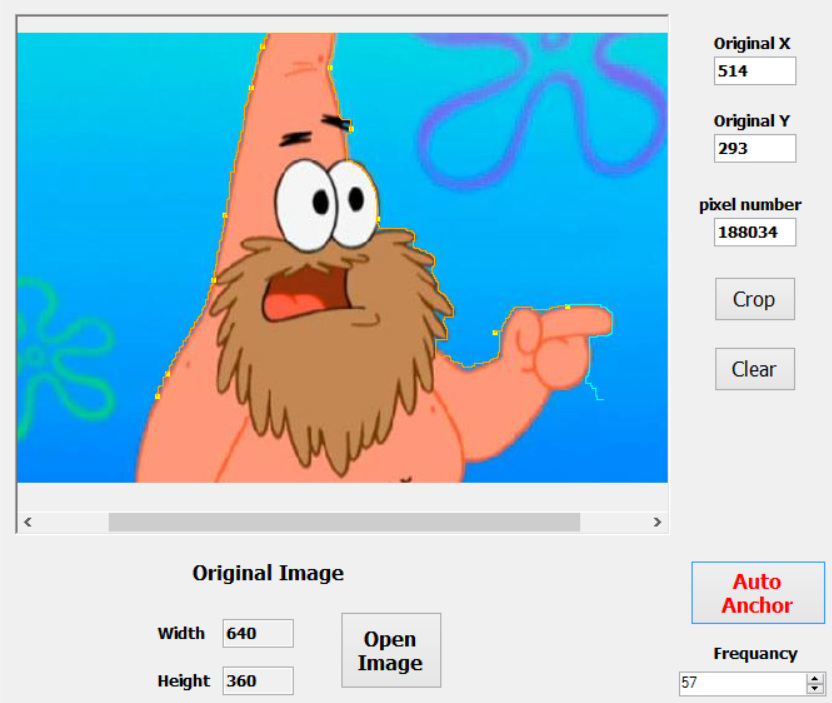
* Place an anchor point and then move by your mouse (while not pressing), you will watch a live wire cope with your motion using the mouse.



* Click where you want to set another anchor point (it’s supposed to be on the bounders of the object you want to select (, a line colored **orange** will appear, denotes the bounders of the selected part of the image.



* You can also use the automatic anchor points’ setter, just activate the mode by clicking on the button, then move freely while keeping the left mouse button clicked, anchor points will be set automatically at accurate places from the boundaries, by the specified frequency ( the distance after which a new anchor point is been placed).



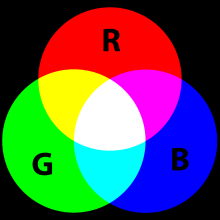
* After finishing blundering the object, press crop button, it will continue selecting the boundaries of your object then it will crop it in a new window giving you the ability to save the image on your hard disk.



**Technical part:**

**Idea beyond the program:**

Usually we can use a 2D-array to represent a 2D-image, because every image consists of number of small units called pixels ordered in columns, and every column contains almost the same number of pixels (just like the 2D-array), and every pixel has a color, this color is just a combination of the tree primary colors (red, green, blue) which is called The RGB color model.



Therefore, we can consider the image as a 2D-Array of RGB, as RGB is a class in which we holds the intense of the 3 colors [1].

Now, let’s remember our target, it was to select an object out of a picture, how can we do this just using that array we had constructed above?

I think it’s possible, but it’s a little bit harder that the following idea:

As long as we want to select an object by going through its boundaries, here we can make use of the change in color that happened through every pixel!

I mean if I have a pixel colored in white, and another pixel also colored in white, there will not can exist an edge between them because there is no difference in the intensity of color.

Therefore, we can say that, if the change of intensity between two pixels in small, we can consider them (the 2 nodes) have a bridge with no cost, and here I mean by no cost that, you will not make much effort to change the first pixel RGB to the second one or vice versa.

Now, we can construct a graph to represent our idea [2], in which we save the values of the first

**[1] Here the RGBPixel [,] is the 2D-Array of RGB**

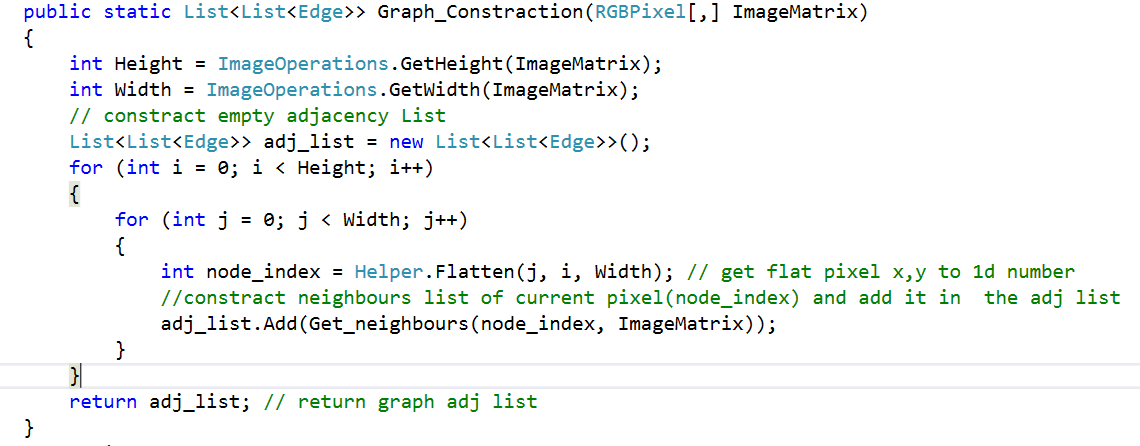
**[2]Graph can be list of list of edges List<List<Edge>>**

Pixel number, the value of the second pixel’s number, and the cost to change this to that.

Now, let’s return back to the idea we wanted to implement, we wanted to classify the edges of the actual [3] picture! But here in our graph, the path from a node to another one will have a big cost if and only if they have a big intensity in color, I mean the cost to go from white pixel to another will be very low, but to go from black pixel to white pixel will be very high, so if we tried to get the shortest path from one node to another we will go through the path which has the same color, and this is not the definition of the edge![3].

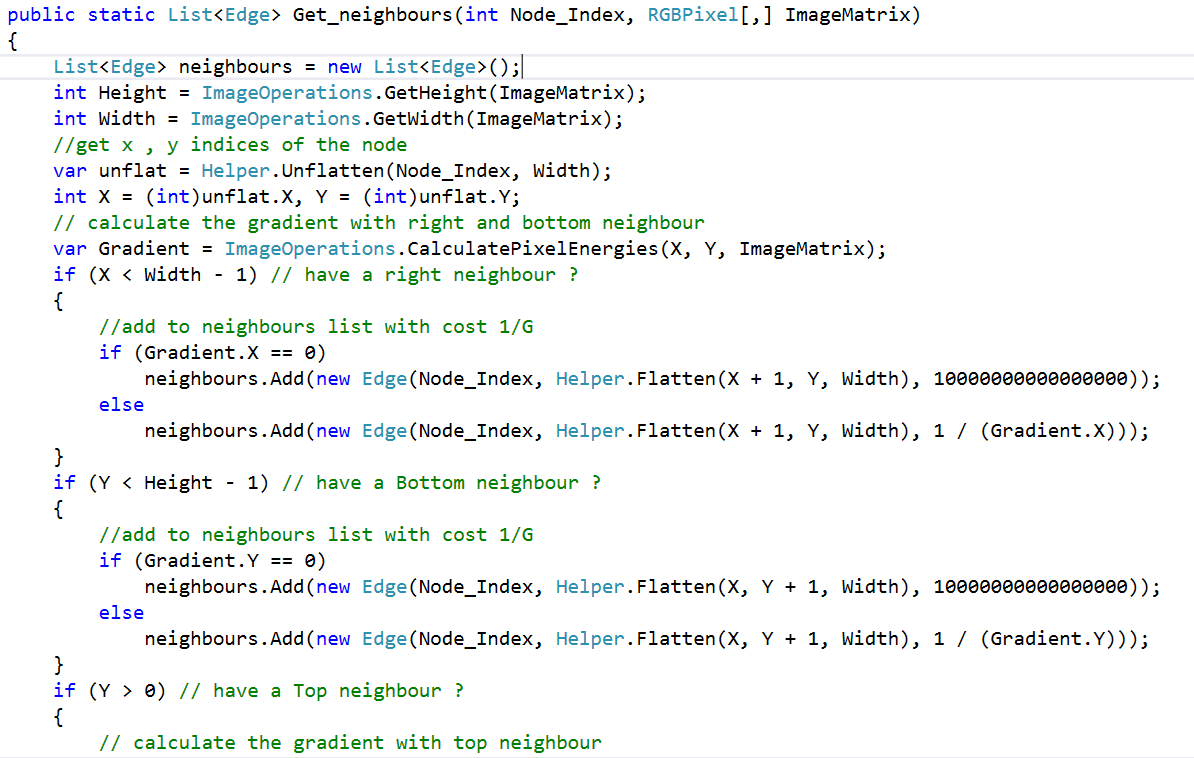
Therefore as long as we want to keep track with the biggest change in color to oscillate around the actual object in the picture not to get deeply inside or deeply outside, we will keep on the value of (1/density) as the density can be considered as the cost I have to pay to go from the first pixel criteria to the another.

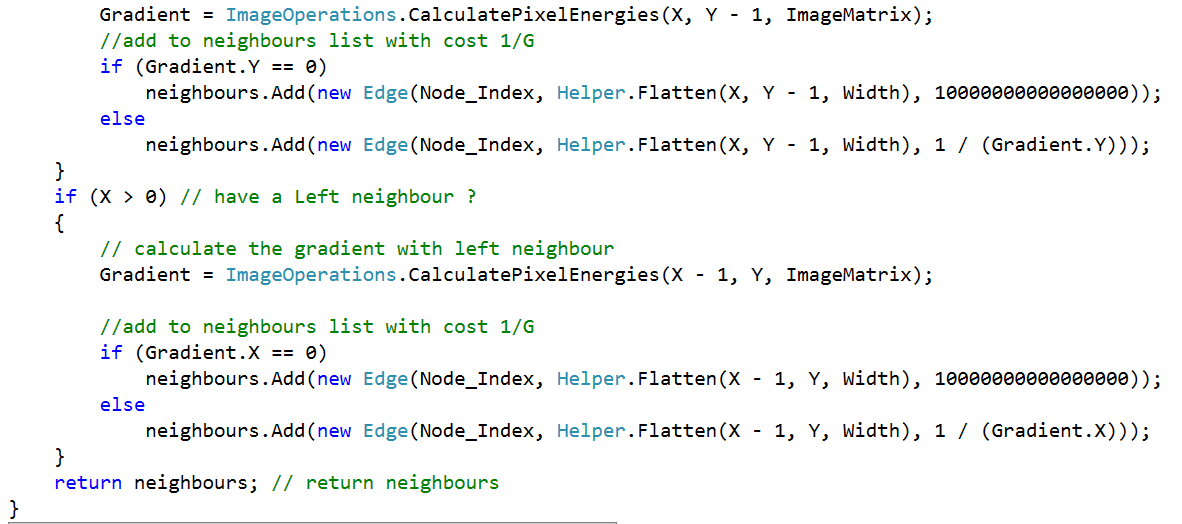
**And that was how to get a weighted graph throughout a picture, which can help us classify the edges of the actual picture, let’s see a piece of code!**

****

You just need to loop throughout your 2D-array which holds the picture RGB (ImageMatrix), then get her neighbors in a list, then add this list to our graph, we can get neighbors through:

[3] The edge in a picture, is the place which has a big density difference in RGB.





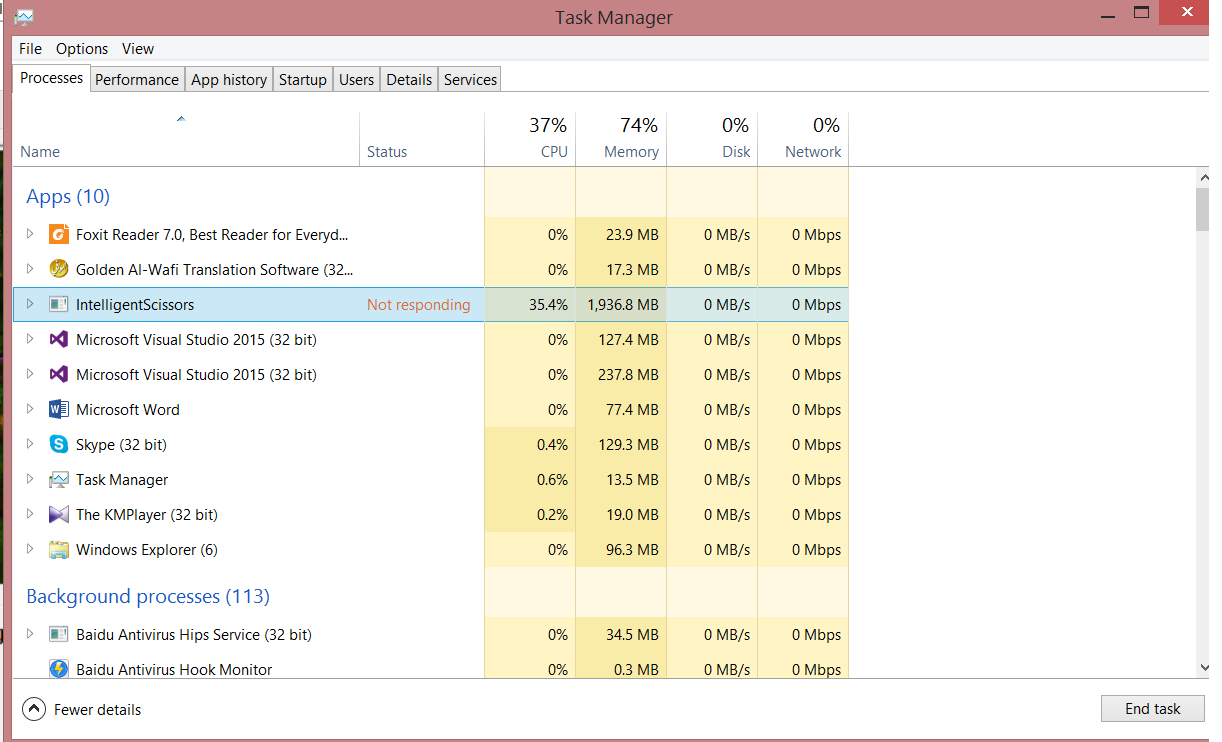
As we check if we have a right neighbor, left, upper, and lower, if there is a neighbor, we will add an edge contains the 2 pixels and the cost we talked about before.

The time complexity of getting the neighbors is O (1) so the cost of constructing the graph will be just O(N^) with a memory complexity of O (V) as V is the number of vertices, because in our adjacency list, we have every Node, connected to at most 4 neighbors, so the cost is 4edges\*v vertices.

OH, that’s great huh…!

But how would you get the path that can classify the edge? Of course now we will run a shortest path algorithm throughout our constructed graph, won’t we?

You seem to like big full HD pictures, don’t you?

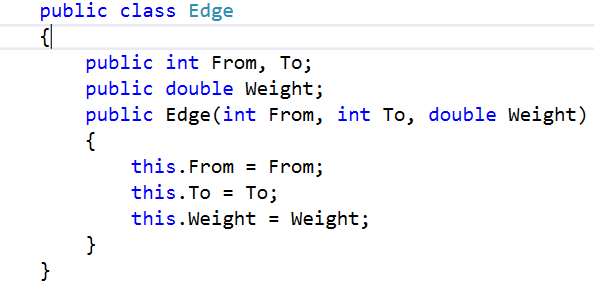


When you open a 3.9MB 2560X1600 Picture

But your memory didn’t share your likes!

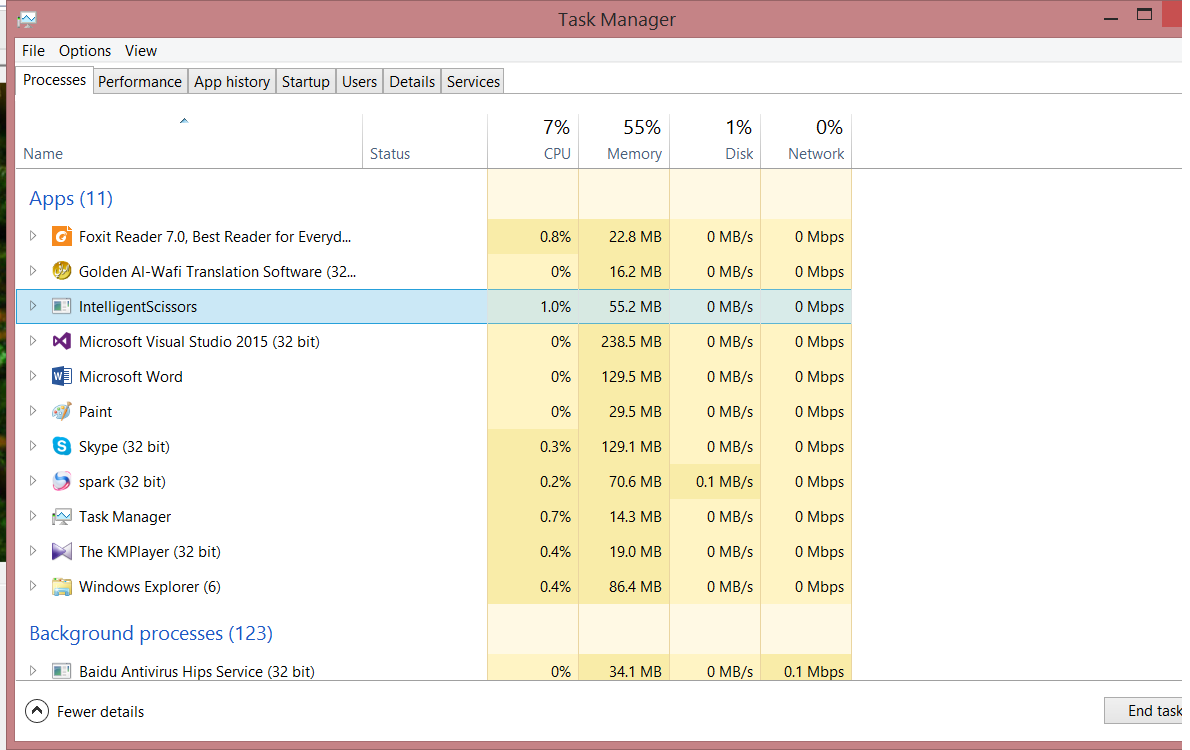
What happened!

Suppose you have a 105 X 105 Picture, then we will have 1010 node, if every nod have a list of at most 4 edges, we will allocate memory of 4X1010 edge, which is consists of 4 bytes for every int variables and 8 bytes for the double variable which is total of (4+4+8=16) byte for the edge:



So we need 64X1010 byte just to construct the graph!

Can we do better?



The same 3.9MB 2560X1600 Picture

What do you think of this?

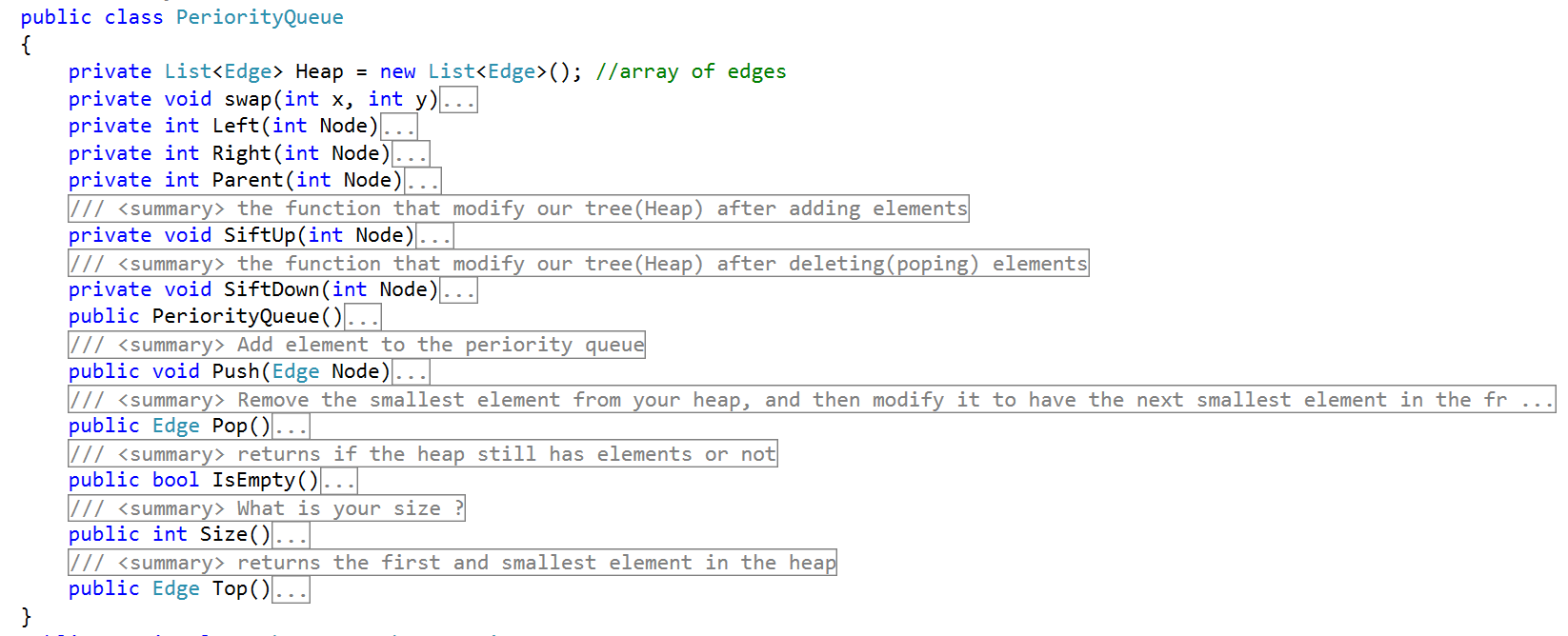
We just get rid of the graph construction idea and then get the neighbors we need directly.

So, how can you get the path that classify the edge?

Using an algorithm called **Dijkstra's algorithm**, it is an algorithm for finding the shortest paths between nodes in a non-negative weighted graph.

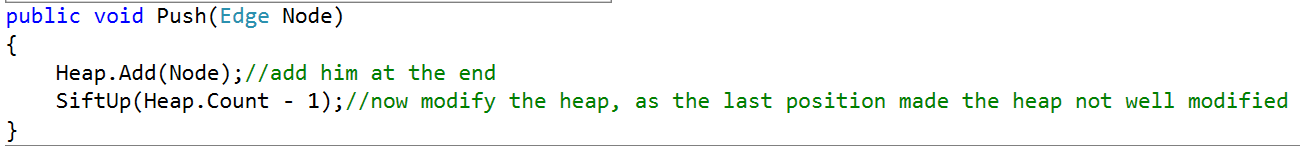
The implementation based on a min-priority queue implemented by a binary heap and running   
  
in O(|E|+|V|\log|V|) (where |E| is the number of edges) and  |V| is the number of nodes

Let’s first walkthrough the min-priority queue:

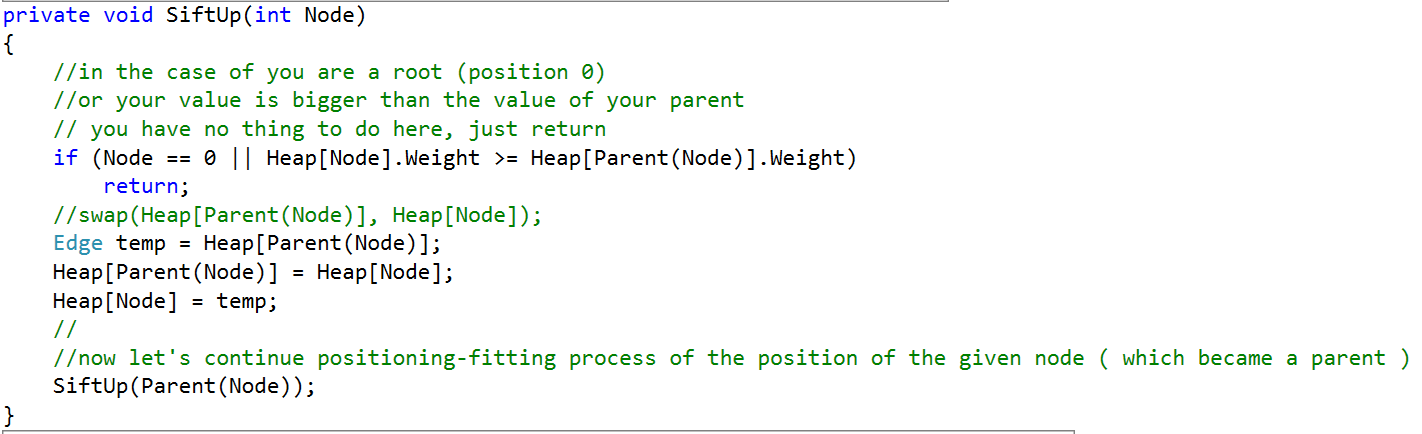


The min-priority-queue is a data structure for maintaining a set S of elements, each with an associated value called key, to have the minimum value at its top.it based on the heap data structure, which is an array object that we can view as a nearly complete binary tree, as each node in the tree corresponds to an element of the array.

As you see there is to main functions, the first one is:

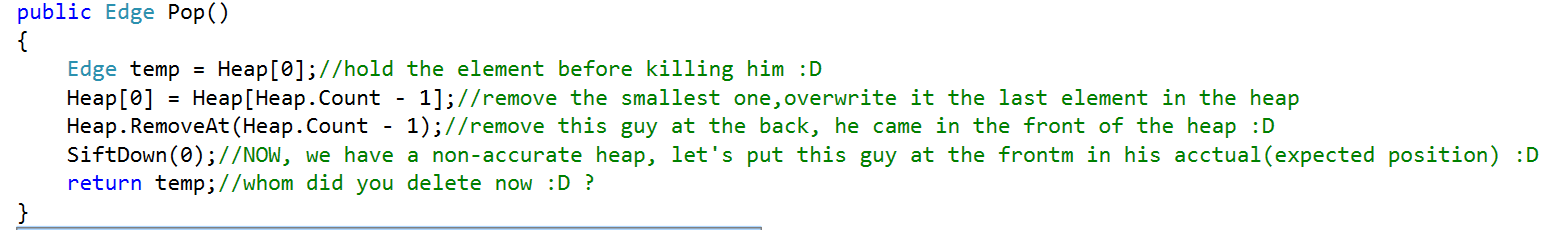


It’s the responsible to add new node to the binary heap tree (the node is an edge), then maintain the minimum-prosperity of the heap (to have the minimum elements on the top), and this is by the function:

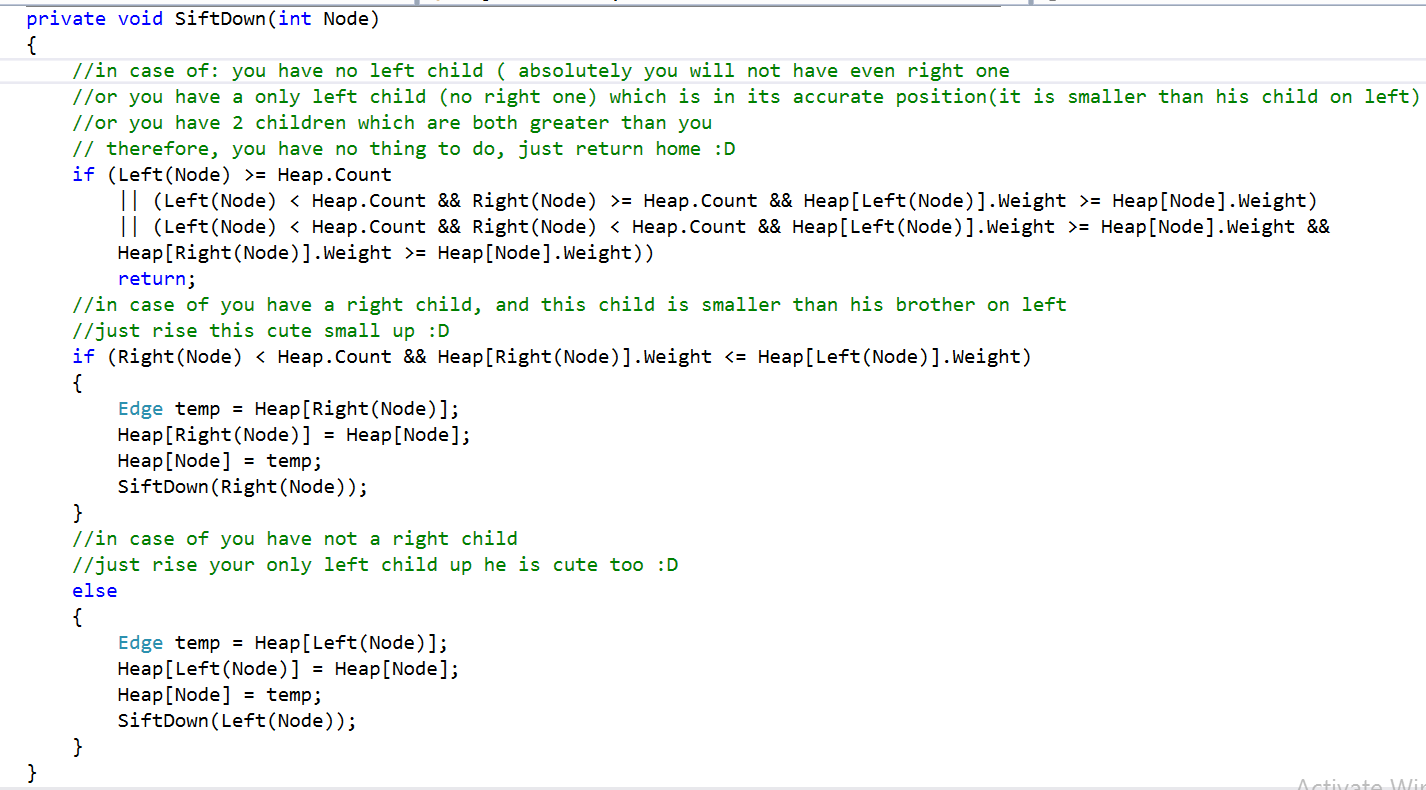


Which is a tail recursive function, which is responsible to swap the new node with her parent, until reaching the position in which it will be minimum than all their children.

The second main function is:



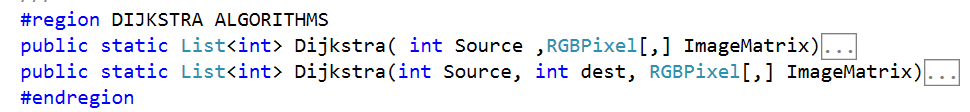
Which is responsible to delete the minimum elements of the heap, and maintain the heap well organized again, by swapping the last element in the heap with the first one, we can say we deleted the root, then removing the last element in the heap, then it’s time to make the heap well organized again, so we call sift down function:



And it’s a tail recursive function too, which keep swapping the node with the smallest of her children, until it reaches the place in which it’s less than their children.

**Now let’s return back to DIJKSTRA algorithm.**

**Dijkstra** is a single source shortest path algorithm, in which I have a source, and I will find the shortest path from this source to any or all destinations, here we have overloaded Dijkstra algorithms so:



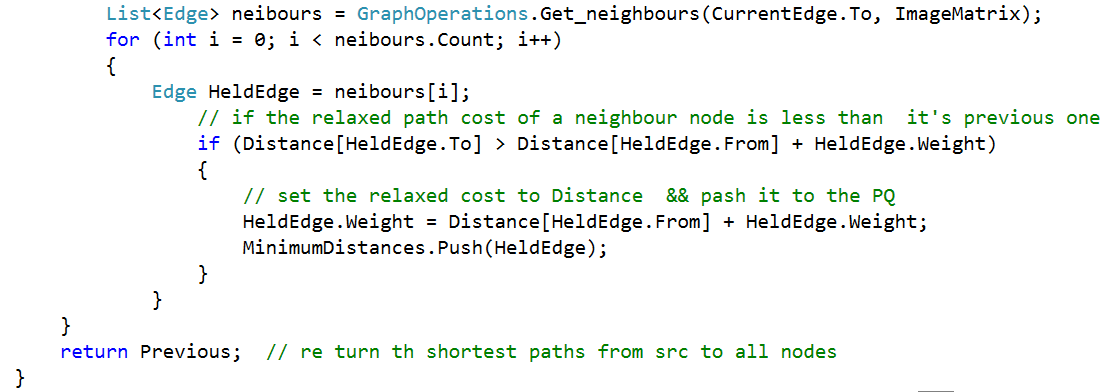
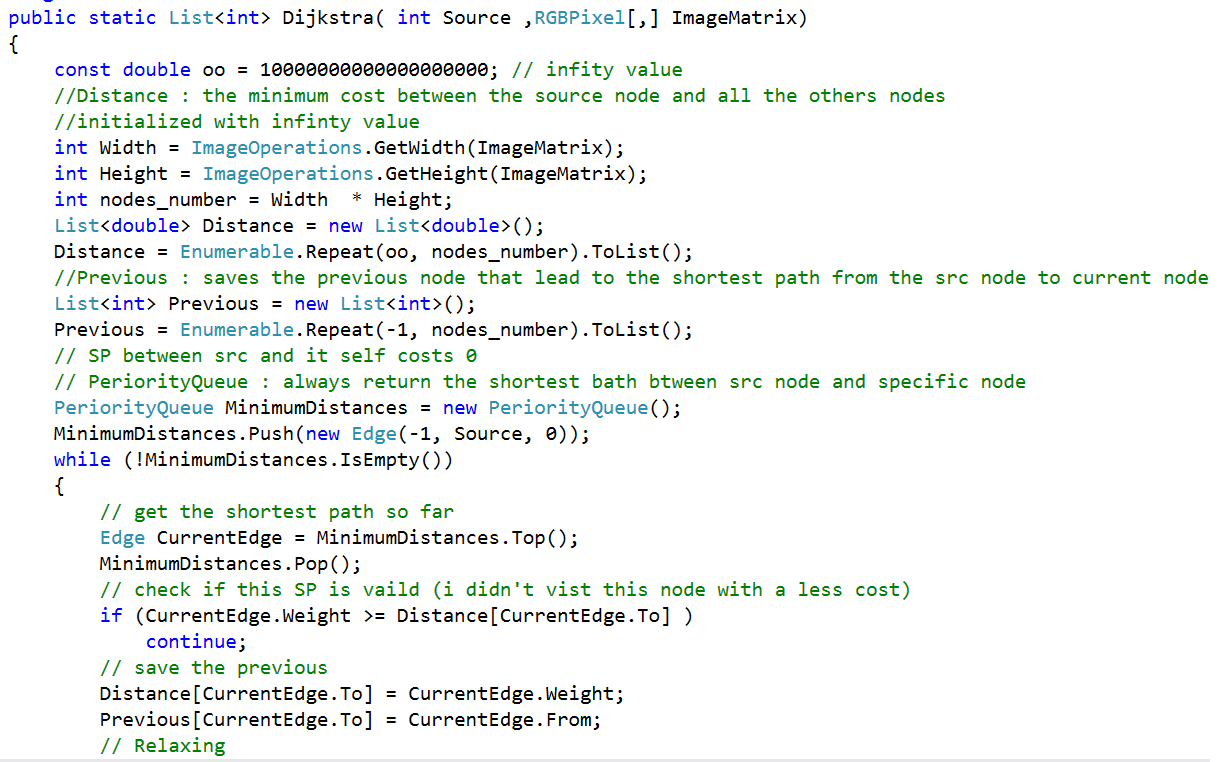
They both have the same logic, but the first have no exact destination, but the second have.

The algorithm based on the idea that, if I ask you can you tell me the shortest path from a node Start to any node, any path that you are 100% sure that it’s actually the shortest path, you will say yes, the minimum cost of the edges attached to the node Start, is leading to a node with the minimum cost, because, taking another path, which has greater cost, and costs are non-negative, you just will continue going greater and bigger.

Now, relax that node (the destination) and I mean by relaxing, is to act that that node is not exist!

So, what does it help us to do? Now, you can got the minimum path from the remaining ones, and do the last two steps.

After relaxing all nodes, we will return an array called previous, which holds the answer for the question, from where did you come?

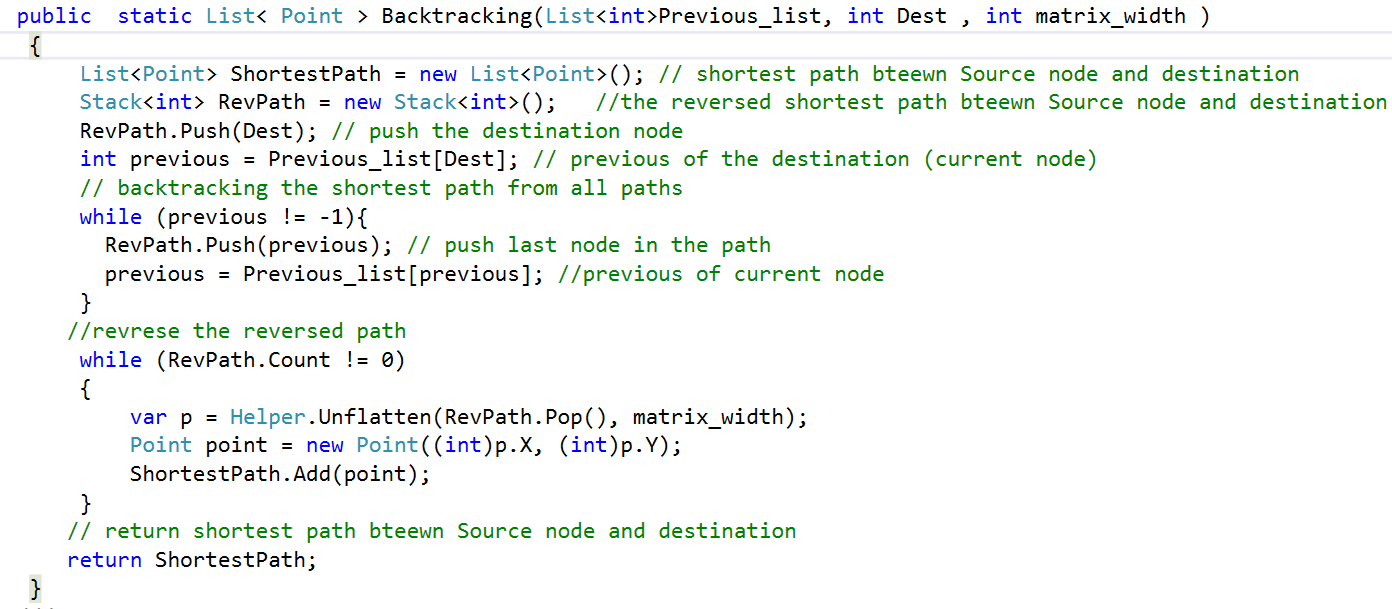


We have a distance array, to keep the minimum distance from source to node I in.

As long as there is edges in the priority queue, we will loop (that takes O (E) as I will go through all edges), then in every step, I will get my neighbors in an array, then I will go through them to relax by them ( actually I relax by valuables ones only and valuable relaxation according to my prospective is the one which maintain small distance from source to the destination J as J is the a neighbor of the node I want to relax by it), through this, every node will be manipulated once, so I will totally loop V, in every loop, I will push a new node, which costs log (n) (because of the sifting up operation of the priority queue ).

So, briefly, as we mentioned before, because each vertex will get pulled out by at most once, so that the inner for loop (across all iterations of the while loop) will select each edge at most once, so it takes O (V), by every V we take Log (v) to pus in the priority queue, so it’s O(V LOG (V) ) and we will go through every edge, so it’s O( E + V LOG (V) ).

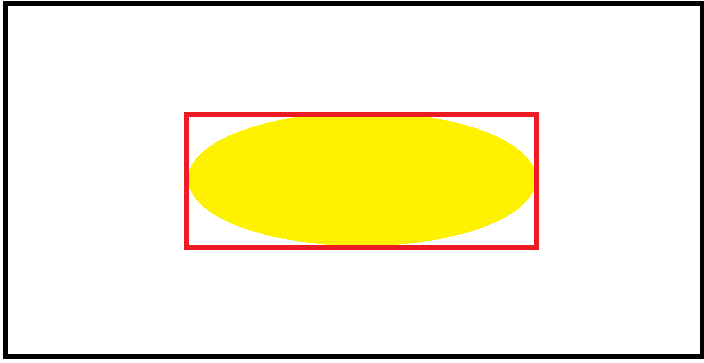
**After running Dijkstra algorithm, we will have the previous list, which denotes who came from where, by backtracking it, we can know the nodes we should connect together, and this is the technique we could use to draw the path, and the live wire.**



As we can see, we have a shortest path list of points which will contain the list of points on the picture box which we will connect together by the drawing tools, by continue going in reverse coping with the values in the previous list, we can construct a stack contains the points, then we reverse them to get the path, that takes O (E) time complexity as E is the number of nodes we will draw.

**Now let’s move to the bonus section: D**

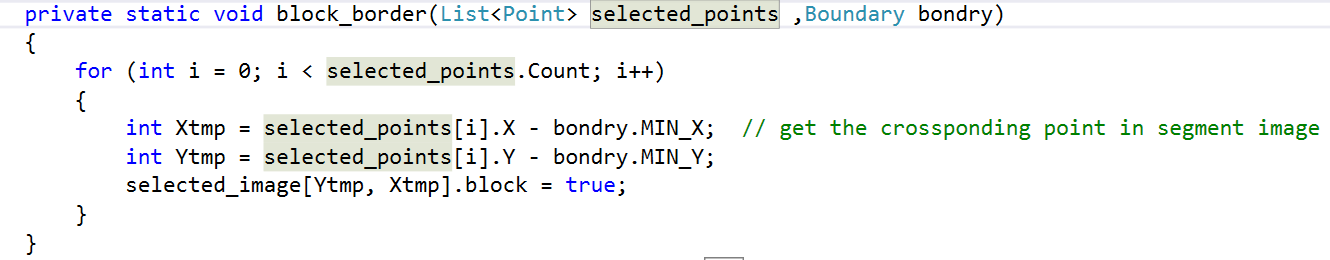
**1- Cropping the image and save it:**

****

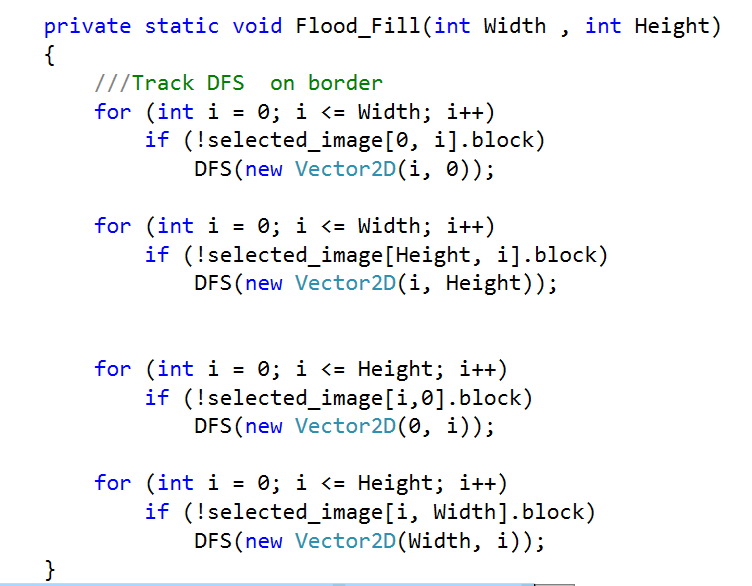
**Suppose the actual picture is the in-black box above, and the object is the in yellow shape, if we can find such red rectangle in which we are completely sure that the object is included in, we can then run the flood fill algorithm using DFS algorithm to find the connected components (which here are the blocks of pixels which are not in our object) then we will color them with a transparent, so you will see the rectangle in red color above, only contains your object without any other thing.**

**Let’s give a look:**

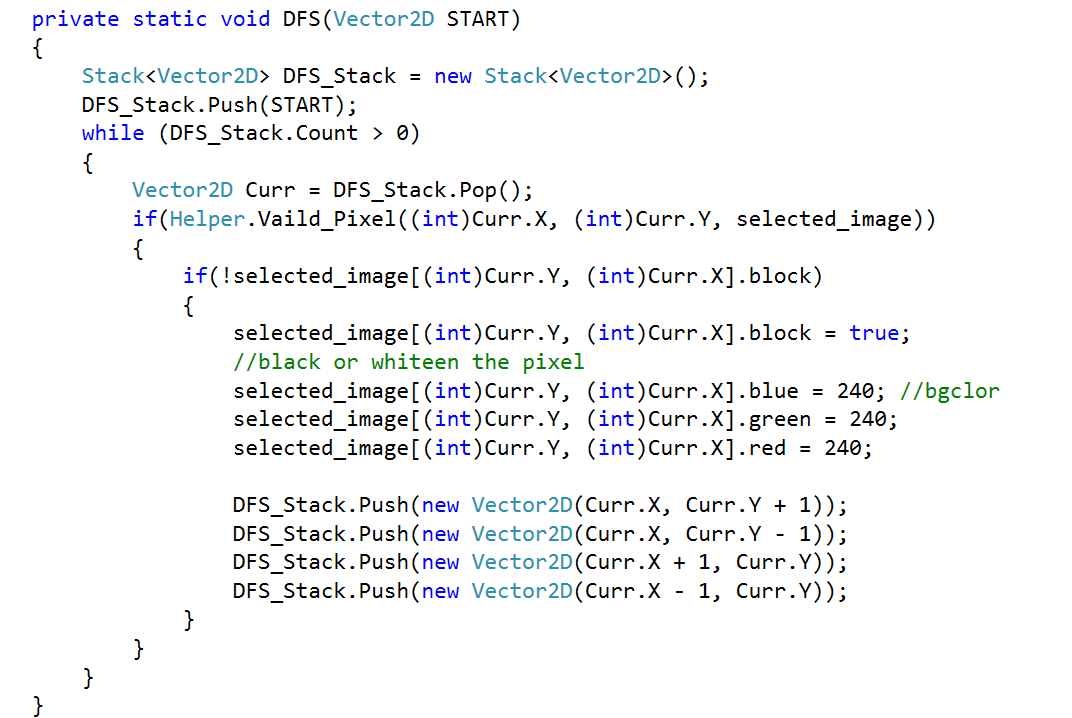
Firstly let’s define a new variable called blocked and initialized by false, to refer to is this pixel visited or not, and let’s go through our path (which we had drawn and mark it as blocked)



Now let’s flood fill our non-necessary parts of our box:



Simply, we will run DFS algorithm through every point of the borders of the box we had bounded our object in, and we will stop DFS-ing when we meet non-valid pixel or visited pixel or blocked pixel (which will refer to the boundaries of the object we want to select.

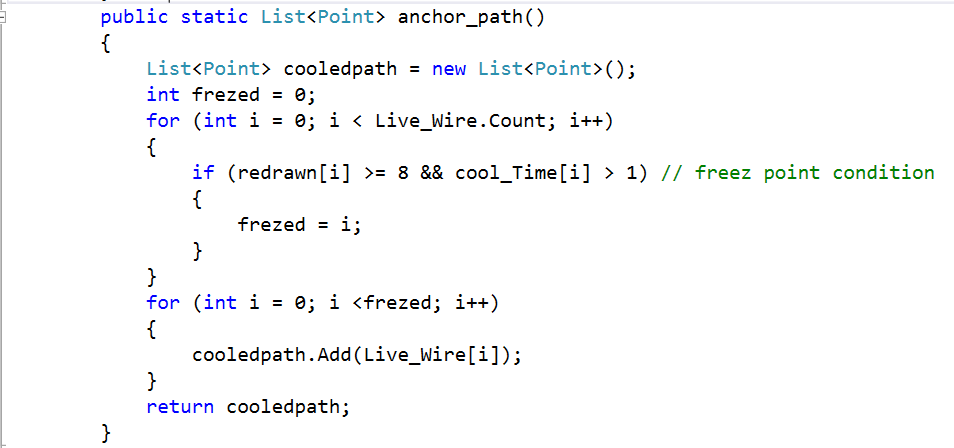


The whole complexity of that part, is the complexity of the DFS which is O (V + E)

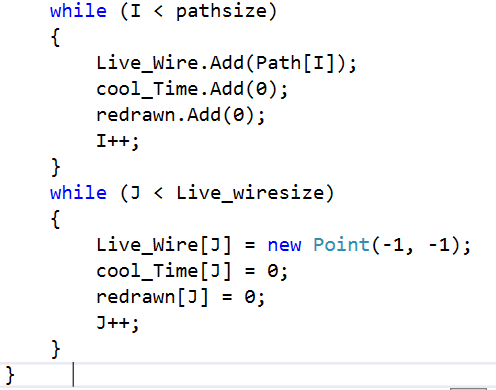
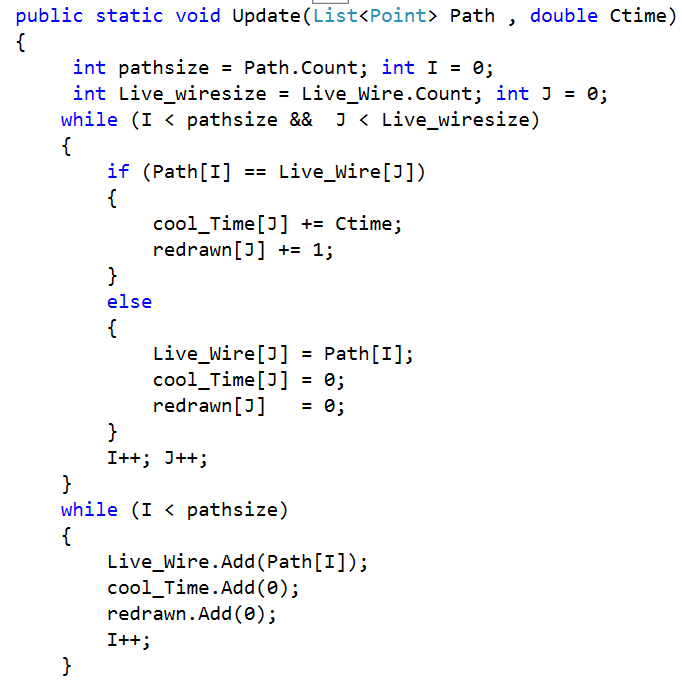
**2- Now, let’s move to the auto anchor point setter algorithmic technique:**

Firstly, we need to define what we called cooled path, it’s the path, which occurred many times (accurate number of times depends on the frequency).

Going through the path so far and the marked line so far, then calculating the maximum intersection between them, that intersection is the cooled path list, it’s the line which it can be drawn and it’s guaranteed to be on the object as far as the actual path marked so far is on the object.



The returned cooled path, is the path which we will draw, going through the live wire, and checking the cooling time, if this node is cooled and re-drowned, then it can be the last frozen node, and so on.



And the whole complexity of this part is O (N) as N is the number of max number of points out of those of the live wire and the drawn line so far.