**Intelligent Scissors Project**

**Team [173]**

Amr Ezzat Hanafy 20191700428

Amr Mahmoud Zakaria 20191700429

Omar Hany Abdelfatah Said 20191700420

**Priority Queue** (Minimum Heap) **:**

**public** **class** PeriorityQueue *// O(log (V))*

{

**private** List<Arc> H = **new** List<Arc>(); *// θ(1)*

**private** **int** LeftChild(**int** Node) *// θ(1)*

{

**return** Node \* 2 + 1; *// θ(1)*

}

**private** **int** RightChild(**int** Node) *// θ(1)*

{

**return** Node \* 2 + 2; *// θ(1)*

}

**private** **int** Parent(**int** Node) *// θ(1)*

{

**return** (Node - 1) / 2; *// θ(1)*

}

**private** **void** ShiftUp(**int** Node) *// θ(1)*

{

**if** (Node == 0 || H[Node].Cost >= H[Parent(Node)].Cost) *// θ(1)*

**return**; *// θ(1)*

Arc temp = H[Parent(Node)]; *// θ(1)*

H[Parent(Node)] = H[Node]; *// θ(1)*

H[Node] = temp; *// θ(1)*

ShiftUp(Parent(Node)); *// θ(1)*

}

**private** **void** ShiftDown(**int** Node) *// θ(1)*

{

**if** (LeftChild(Node) >= H.Count

|| (LeftChild(Node) < H.Count && RightChild(Node) >= H.Count && H[LeftChild(Node)].Cost >= H[Node].Cost)

|| (LeftChild(Node) < H.Count && RightChild(Node) < H.Count && H[LeftChild(Node)].Cost >= H[Node].Cost &&

H[RightChild(Node)].Cost >= H[Node].Cost)) *// θ(1)*

**return**; *// θ(1)*

**if** (RightChild(Node) < H.Count && H[RightChild(Node)].Cost <= H[LeftChild(Node)].Cost) *// θ(1)*

{

Arc temp = H[RightChild(Node)]; *// θ(1)*

H[RightChild(Node)] = H[Node]; *// θ(1)*

H[Node] = temp; *// θ(1)*

ShiftDown(RightChild(Node)); *// θ(1)*

}

**else**

{

Arc temp = H[LeftChild(Node)]; *// θ(1)*

H[LeftChild(Node)] = H[Node]; *// θ(1)*

H[Node] = temp; *// θ(1)*

ShiftDown(LeftChild(Node)); *// θ(1)*

}

}

**public** **void** Push(Arc Node) *// O(log (V))*

{

H.Add(Node); *// θ(1)*

ShiftUp(H.Count - 1); *// θ(1)*

}

**public** Arc Pop() *// O(log (V))*

{

Arc temp = H[0]; *// θ(1)*

H[0] = H[H.Count - 1]; *// θ(1)*

H.RemoveAt(H.Count - 1); *// O(log (V))*

ShiftDown(0); *// θ(1)*

**return** temp; *// θ(1)*

}

**public** **bool** IsEmpty() *// θ(1)*

{

**if** (H.Count == 0) *// θ(1)*

**return** **true**; *// θ(1)*

**return** **false**; *// θ(1)*

}

**public** Arc Top() *// θ(1)*

{

**return** H[0]; *// θ(1)*

}

}

Which means that a parent node can't have a greater value than itschildren. Thus, the minimum element is located at the root, and the maximum elements are located in the leaves. Depending on the type of heap used, the heap property may have additional requirements. So its easier to apply dijsktra algorithm.

**Dijkstra Algorithm :**

**public** **static** List<Point> Inverting\_Path(List<**int**> ParentList, **int** Dest, **int** matrix\_width) *// O(E)*

{

List<Point> ShortestPath = **new** List<Point>(); *// θ(1)*

Stack<**int**> ReversedPath = **new** Stack<**int**>(); *// θ(1)*

ReversedPath.Push(Dest); *// θ(1)*

**int** Parent = ParentList[Dest]; *// θ(1)*

**while** (Parent != -1) *// θ(E)*

{

ReversedPath.Push(Parent); *// θ(1)*

Parent = ParentList[Parent]; *// θ(1)*

}

**while** (ReversedPath.Count != 0) *// O(E)*

{

**var** TwoD = Functions.oneDtoTwoD(ReversedPath.Pop(), matrix\_width); *// θ(1)*

Point point = **new** Point((**int**)TwoD.X, (**int**)TwoD.Y); *// θ(1)*

ShortestPath.Add(point); *// θ(1)*

}

**return** ShortestPath; *// θ(1)*

}

Fill a stack with the selected nodes from the anchor point until reach the begging anchor point. Then Pop all the stack one by one and fill the shortest path list. So it has all the anchor points from the begging to the last anchor point (Destination)

Note: Using stack to easily reverse the anchor nodes to get the right path with low complexity.

**Generate Dijkstra Algorithm :**

**public** **static** List<**int**> Dijkstra(**int** src, **int** dest, RGBPixel[,] ImageMatrix) *// O(EV)*

{

**double** infinity = 9999999999999999999; *// θ(1)*

**int** MinValue = -1; *// θ(1)*

**int** W = ImageOperations.GetWidth(ImageMatrix); *// θ(1)*

**int** H = ImageOperations.GetHeight(ImageMatrix); *// θ(1)*

**int** Pixel\_number = W \* H; *// θ(1)*

List<**double**> Distance = **new** List<**double**>(); *// θ(1)*

Distance = Enumerable.Repeat(infinity, Pixel\_number).ToList(); *//O(E)*

List<**int**> ParentsPath = **new** List<**int**>(); *// θ(1)*

ParentsPath = Enumerable.Repeat(MinValue, Pixel\_number).ToList(); *//O(E)*

PeriorityQueue Shortest\_Distances = **new** PeriorityQueue(); *// θ(1)*

Shortest\_Distances.Push(**new** Arc(-1, src, 0)); *// θ(1)*

**if** (dest == 0) *// O(EV)*

{

**while** (!Shortest\_Distances.IsEmpty()) *// O(EV)*

{

Arc CurrentEdge = Shortest\_Distances.Top(); *// θ(1)*

Shortest\_Distances.Pop(); *// O(Log(V))*

**if** (CurrentEdge.Cost < Distance[CurrentEdge.dest]) *// O(V)*

{

Distance[CurrentEdge.dest] = CurrentEdge.Cost; *// θ(1)*

ParentsPath[CurrentEdge.dest] = CurrentEdge.src; *// θ(1)*

List<Arc> neibours = Functions.GetSibling(CurrentEdge.dest, ImageMatrix);

**int** i = 0; *// θ(1)*

**var** neiboursNUm = neibours.Count; *// θ(1)*

**while** (i < neiboursNUm) *// O(V)*

{

**if** (Distance[neibours[i].dest] > Distance[neibours[i].src] + neibours[i].Cost) *// θ(1)*

{

neibours[i].Cost += Distance[neibours[i].src]; *// θ(1)*

Shortest\_Distances.Push(neibours[i]); *// θ(1)*

}

i++; *// θ(1)*

}

}

}

**return** ParentsPath; *// θ(1)*

}

**else**

{

**while** (!Shortest\_Distances.IsEmpty()) *// O(N^3)*

{

Arc CurrentEdge = Shortest\_Distances.Top(); *// θ(1)*

Shortest\_Distances.Pop(); *// O(N)*

**if** (CurrentEdge.Cost < Distance[CurrentEdge.dest]) *// O(N^2)*

{

ParentsPath[CurrentEdge.dest] = CurrentEdge.src; *// θ(1)*

Distance[CurrentEdge.dest] = CurrentEdge.Cost; *// θ(1)*

**if** (CurrentEdge.dest == dest) **break**; *// θ(1)*

List<Arc> neibours = Functions.GetSibling(CurrentEdge.dest, ImageMatrix);

**int** i = 0; *// θ(1)*

**while** (i < neibours.Count) *// O(N^2)*

{

**if** (Distance[neibours[i].dest] > Distance[neibours[i].src] + neibours[i].Cost) *// Ω(1) O(N)*

{

neibours[i].Cost += Distance[neibours[i].src]; *// θ(1)*

Shortest\_Distances.Push(neibours[i]); *// Ω(1) O(N)*

}

i++; *// θ(1)*

}

}

}

**return** ParentsPath; *// θ(1)*

}

}

**public** **static** List<Point> Create\_ShortestPath(**int** src, **int** dest, RGBPixel[,] ImageMatrix) *// O(EV)*

{

List<**int**> ParentList = Dijkstra(src, dest, ImageMatrix); *// O(EV)*

**return** Inverting\_Path(ParentList, dest, ImageOperations.GetWidth(ImageMatrix)); *// O(V)*

}

Using Dijkstra algorithm to find the shortest path by connecting the anchor nodes throw the least cost path to be easily back tracked. By calculating the energy difference between the pixels and their nearby pixels in four directions. And return a list holding all the pixels of the least cost shortest path.

Crop Image Boundaries :

**public** **static** RGBPixel[,] CropedImageFrame(RGBPixel[,] ImageMatrix, Boundary border) *// O(V^2)*

{

**int** W = border.X\_max - border.X\_min; *// θ(1)*

**int** H = border.Y\_max - border.Y\_min; *// θ(1)*

RGBPixel[,] CropedImage = **new** RGBPixel[H + 1, W + 1]; *// θ(1)*

**int** i = 0; *// θ(1)*

**while** (i <= H) *// O(V^2)*

{

**int** j = 0; *// θ(1)*

**while** (j <= W) *// O(V)*

{

CropedImage[i, j] = ImageMatrix[border.Y\_min + i, border.X\_min + j]; *// θ(1)*

j++; *// θ(1)*

}

i++; *// θ(1)*

}

**return** CropedImage; *// θ(1)*

}

This function finds the width and height of the selected shortest path nodes and crop the original image into a Square or Rectangle.

Nearby Pixels :

**public** **static** List<Arc> GetSibling(**int** Pixel, RGBPixel[,] ImageMatrix) *// θ(1)*

{

List<Arc> Sibling = **new** List<Arc>(); *// θ(1)*

**int** H = ImageOperations.GetHeight(ImageMatrix); *// θ(1)*

**int** W = ImageOperations.GetWidth(ImageMatrix); *// θ(1)*

Vector2D TwoD = oneDtoTwoD(Pixel, W); *// θ(1)*

**int** X = (**int**)TwoD.X; *// θ(1)*

**int** Y = (**int**)TwoD.Y; *// θ(1)*

**var** G = ImageOperations.CalculatePixelEnergies(X, Y, ImageMatrix);

**if** (X < W - 1) *// θ(1)*

{

**if** (G.X != 0) *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X + 1, Y, W), 1 / (G.X))); *// θ(1)*

**else** *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X + 1, Y, W), 10000000000000000)); *// θ(1)*

}

**if** (Y < H - 1) *// θ(1)*

{

**if** (G.Y != 0) *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X, Y + 1, W), 1 / (G.Y))); *// θ(1)*

**else** *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X, Y + 1, W), 10000000000000000)); *// θ(1)*

}

**if** (Y > 0) *// θ(1)*

{

**if** (G.Y != 0) *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X, Y - 1, W), 1 / (G.Y))); *// θ(1)*

**else** *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X, Y - 1, W), 10000000000000000)); *// θ(1)*

}

**if** (X > 0) *// θ(1)*

{

**if** (G.X != 0) *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X - 1, Y, W), 1 / (G.X))); *// θ(1)*

**else** *// θ(1)*

Sibling.Add(**new** Arc(Pixel, twoDtoOneD(X - 1, Y, W), 10000000000000000)); *// θ(1)*

}

**return** Sibling; *// θ(1)*

}

This function calculate difference between all the nearby energies of the current selected node in its four directions ( up, down, left, right ).

DFS And Cropping Image :

**private** **static** RGBPixel[,] CropedImage; *// θ(1)*

**public** **static** RGBPixel[,] Croped\_Image(List<Point> MainSelection, RGBPixel[,] ImageMatrix) *// O(E^2 + V)*

{

Boundary border = Border\_Limits(MainSelection); *// Boundary of the main selection*

CropedImage = Functions.CropedImageFrame(ImageMatrix, border); *// get croped image*

**int** counter = MainSelection.Count; *// θ(1)*

**int** i = 0; *// θ(1)*

**while** (i < counter) *// O(V)*

{

**int** X = MainSelection[i].X - border.X\_min; *// θ(1)*

**int** Y = MainSelection[i].Y - border.Y\_min; *// θ(1)*

CropedImage[Y, X].visited = **true**; *// θ(1)*

i++; *// θ(1)*

}

filtering\_Image(ImageOperations.GetWidth(CropedImage) - 1, ImageOperations.GetHeight(CropedImage) - 1); *// O(E^2)*

**return** CropedImage; *// θ(1)*

}

**private** **static** **void** filtering\_Image(**int** Width , **int** Height) *// O(E^2)*

{

**int** i = 0; *// θ(1)*

**while** (i <= Width) *// O(E^2)*

{

**if** (!CropedImage[0, i].visited) *// O(E)*

DFS(**new** Vector2D(i, 0)); *// O(E)*

i++; *// θ(1)*

}

i = 0; *// θ(1)*

**while** (i <= Width) *// O(E^2)*

{

**if** (!CropedImage[Height, i].visited) *// O(E)*

DFS(**new** Vector2D(i, Height)); *// O(E)*

i++; *// θ(1)*

}

i = 0; *// θ(1)*

**while** (i <= Height) *// O(E^2)*

{

**if** (!CropedImage[i, 0].visited) *// O(E)*

DFS(**new** Vector2D(0, i)); *// O(E)*

i++; *// θ(1)*

}

i = 0; *// θ(1)*

**while** (i <= Height) *// O(E^2)*

{

**if** (!CropedImage[i, Width].visited) *// O(E)*

DFS(**new** Vector2D(Width, i)); *// O(E)*

i++; *// θ(1)*

}

}

**private** **static** **void** DFS(Vector2D SelectedPixel) *// O(E)*

{

Queue<Vector2D> queue = **new** Queue<Vector2D>(); *// θ(1)*

queue.Enqueue(SelectedPixel); *// θ(1)*

**while** (queue.Count > 0) *// O(E)*

{

Vector2D Cpixel = queue.Dequeue(); *// θ(1)*

**bool** FoundX = (Cpixel.X >= 0 && Cpixel.X < ImageOperations.GetWidth(CropedImage)); *// θ(1)*

**bool** FoundY = (Cpixel.Y >= 0 && Cpixel.Y < ImageOperations.GetHeight(CropedImage)); *// θ(1)*

**if** (FoundX && FoundY && !CropedImage[(**int**)Cpixel.Y, (**int**)Cpixel.X].visited) *// θ(1)*

{

*//make these pixels white*

CropedImage[(**int**)Cpixel.Y, (**int**)Cpixel.X].visited = **true**; *// θ(1)*

CropedImage[(**int**)Cpixel.Y, (**int**)Cpixel.X].red = 255; *// θ(1)*

CropedImage[(**int**)Cpixel.Y, (**int**)Cpixel.X].green = 255; *// θ(1)*

CropedImage[(**int**)Cpixel.Y, (**int**)Cpixel.X].blue = 255; *// θ(1)*

*//push arounded pixels*

queue.Enqueue(**new** Vector2D(Cpixel.X, Cpixel.Y + 1)); *// θ(1)*

queue.Enqueue(**new** Vector2D(Cpixel.X, Cpixel.Y - 1)); *// θ(1)*

queue.Enqueue(**new** Vector2D(Cpixel.X + 1, Cpixel.Y)); *// θ(1)*

queue.Enqueue(**new** Vector2D(Cpixel.X - 1, Cpixel.Y)); *// θ(1)*

}

}

}

**private** **static** Boundary Border\_Limits(List<Point> selected\_points) *// O(V)*

{

Boundary border; *// θ(1)*

border.X\_max = border.Y\_max = -999999999; *// θ(1)*

border.X\_min = border.Y\_min = 999999999; *// θ(1)*

**int** counter = selected\_points.Count; *// θ(1)*

**int** i = 0; *// θ(1)*

**while** ( i < counter) *// O(V)*

{

**if** (selected\_points[i].X > border.X\_max) *// θ(1)*

border.X\_max = selected\_points[i].X; *// θ(1)*

**if** (selected\_points[i].X < border.X\_min) *// θ(1)*

border.X\_min = selected\_points[i].X; *// θ(1)*

**if** (selected\_points[i].Y > border.Y\_max) *// θ(1)*

border.Y\_max = selected\_points[i].Y; *// θ(1)*

**if** (selected\_points[i].Y < border.Y\_min) *// θ(1)*

border.Y\_min = selected\_points[i].Y; *// θ(1)*

i++; *// θ(1)*

}

**return** border; *// θ(1)*

}

After cropping the boundaries of the selected anchor points as a default value of False (Non-Visited) turning the part bounded by the anchor nodes into True (Visited) and keeping the rest as False (Non-Visited) then color each non-visited pixel with white background color and turning it form false (Non-Visited) into true (Visited).