Anexo 1. Código fuente

Script 1. *App_Script.cs*

Descripción: El primer *script* se encarga mayormente de las acciones que el usuario puede realizar en la ventana principal de la aplicación y al funcionamiento en general de ésta, que corresponde con la especificación del problema.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System;
using System.Collections.Generic;

public class Rectangle
{
    private int id;
    public float x, y, w, h;

    public Rectangle(float x, float y, float w, float h)
    {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }

    public void set_id(int id)
    {
        this.id = id;
    }

    public int get_id()
    {
        return id;
    }

    public Boolean fits_in(Rectangle outer)
    {
        return outer.w >= this.w && outer.h >= this.h;
    }

    public Boolean same_size_as(Rectangle other)
    {
        return this.w == other.w && this.h == other.h;
    }

}

public class Node
{
    public Node left;
    public Node right;
    public Rectangle rect;
    public Boolean filled;


    public Node()
    {
        this.left = null;
        this.right = null;
        this.rect = null;
        this.filled = false;
    }

    public Node insert_rect(Rectangle rect)
    {


        // if it is not a leaf, then
        if(this.left != null)
        {
            Node newNode = this.left.insert_rect(rect);
            // try insert into first child
            if (newNode != null)
            {
                return newNode;
            }

            else return this.right.insert_rect(rect);
        }

        if (this.filled)
            return null;

        if (!rect.fits_in(this.rect))
            return null;

        if (rect.same_size_as(this.rect))
        {
            // same size means it's filled when inserted
            this.filled = true;
            return this;
        }

        // When a call to this methods ends up here, it means that has get to a leaf,
and this one must branch:

        // Branch
        this.left = new Node();
        this.right = new Node();

        float width_diff = this.rect.w - rect.w;
        float height_diff = this.rect.h - rect.h;

        var me = this.rect;

        if (width_diff > height_diff)
        {
            // split literally into left and right, putting the rect on the left.
            Debug.Log("Split into left and right. Putting the rect on the left");
            this.left.rect = new Rectangle(me.x, me.y, rect.w, me.h);
            this.right.rect = new Rectangle(me.x + rect.w, me.y, me.w - rect.w, me.h);

        }
        else
        {
            // split into top and bottom, putting rect on top.
            Debug.Log("Split into top and bottom, putting rect on top");
            this.left.rect = new Rectangle(me.x, me.y, me.w, rect.h);
            this.right.rect = new Rectangle(me.x, me.y - rect.h, me.w, me.h - rect.h);
        }

        return this.left.insert_rect(rect);

    }

}

public class App_Script : MonoBehaviour
{
    private const float MIN_PIECE_HEIGHT = 25;
    private const float MAX_PIECE_HEIGHT = 125;

    private const float MIN_PIECE_WIDTH = 25;
    private const float MAX_PIECE_WIDTH = 125;
    private const float MIN_SHEET_HEIGHT = 200;
    private const float MAX_SHEET_HEIGHT = 420;

    private const float MIN_SHEET_WIDTH = 320;
    private const float MAX_SHEET_WIDTH = 600;




    // Starting window elements
    public Text text_population_size;
    public Slider slider_population_size;
    public Canvas canvas_starting_window;
    public Button button_starting_window;
    public Canvas canvas_problem_setup_window;



    // Problem setup window elements


    // Piece related elements
    public GameObject panel_piece_canvas;

    public Scrollbar scrollbar_piece_width;
    public Scrollbar scrollbar_piece_height;
    public GameObject panel_piece;
    public InputField inputField_piece_height;
    public InputField inputField_piece_width;

    public Text text_restriction_comment_height;
    public Text text_restriction_comment_width;

    public Text text_piece_area;

    public Slider slider_RGB_R;
    public Slider slider_RGB_G;
    public Slider slider_RGB_B;

    public Button button_add_piece;


    // Sheet related elements
    public GameObject panel_canvas_sheet;
    public GameObject panel_sheet;
    public Scrollbar scrollbar_canvas_sheet_width;
    public Scrollbar scrollbar_canvas_sheet_height;


    public InputField inputField_sheet_height;
    public InputField inputField_sheet_width;

    public Text text_restriction_comment_height_sheet;
    public Text text_restriction_comment_width_sheet;

    public Text text_sheet_area;

    public Text text_sheet_remaining_capacity;

    public Text text_warning;
    public Button button_help_at_warning;
    public GameObject panel_warning_and_help;

    public Button button_sheet_clear;
    public Button button_sheet_rearrange;



    // Virtual list of pieces added to the sheet
    public List<GameObject> list_pieces = new List<GameObject>();

    // Current dimensions of the sheet
    public float current_sheet_width;
    public float current_sheet_height;
    // Starting node representing the whole sheet space
    Node start_node = new Node();


    private bool automatic_rearrange_upon_resize = true;



    // Use this for initialization
    void Start()
    {
        // Set screen resolution for UI consistency purposes
        Screen.SetResolution(1210, 658, false);


        // Starting window (text) setup
        text_population_size.text = "Population size: 50";


        // Starting window delegates
        slider_population_size.onValueChanged.AddListener(delegate {
ValueChangeAction(); });
        button_starting_window.onClick.AddListener(delegate {
starting_window_button_Clicked_Action(); });


        // Problem setup window delegates
        scrollbar_piece_height.onValueChanged.AddListener(delegate {
scrollbar_piece_height_onValueChanged(); });
        scrollbar_piece_width.onValueChanged.AddListener(delegate {
scrollbar_piece_width_onValueChanged(); });
        inputField_piece_height.onEndEdit.AddListener(delegate {
inputField_piece_height_onEndEdit(); });
        inputField_piece_width.onEndEdit.AddListener(delegate {
inputField_piece_width_onEndEdit(); });

        // Add piece button delegate
        button_add_piece.onClick.AddListener(delegate { button_add_piece_onClick(); });

        // Scrollbar of canvas of the sheet delegates
        scrollbar_canvas_sheet_height.onValueChanged.AddListener(delegate {
scrollbar_canvas_sheet_height_onValueChanged(); });
        scrollbar_canvas_sheet_width.onValueChanged.AddListener(delegate {
scrollbar_canvas_sheet_width_onValueChanged(); });

        // RGB sliders delegates
        slider_RGB_R.onValueChanged.AddListener(delegate {
slider_RGB_R_onValueChanged(); });
        slider_RGB_G.onValueChanged.AddListener(delegate {
slider_RGB_G_onValueChanged(); });
        slider_RGB_B.onValueChanged.AddListener(delegate {
slider_RGB_B_onValueChanged(); });


        // Sheet InputField delegates
        inputField_sheet_height.onEndEdit.AddListener(delegate {
inputField_sheet_height_onEndEdit(); });
        inputField_sheet_width.onEndEdit.AddListener(delegate {
inputField_sheet_width_onEndEdit(); });

        // Sheet clear button delegate
        button_sheet_clear.onClick.AddListener(delegate { button_sheet_clear_onClick();
});

        // Sheet rearrange button delegate
        button_sheet_rearrange.onClick.AddListener(delegate {
button_sheet_rearrange_onClick(); });



        // Setup the starting text inside the piece's InputFields
        inputField_piece_height.text = MIN_PIECE_HEIGHT.ToString();
        inputField_piece_width.text = MIN_PIECE_WIDTH.ToString();

        // Setup the starting text inside the sheet's InputFields
        inputField_sheet_height.text = MAX_SHEET_HEIGHT.ToString();
        inputField_sheet_width.text = MAX_SHEET_WIDTH.ToString();

        // Setup the restriction comments accordingly to the restrictions
        text_restriction_comment_height.text = "Between " + MIN_PIECE_HEIGHT + " and "
+ MAX_PIECE_HEIGHT;
        text_restriction_comment_width.text = "Between " + MIN_PIECE_WIDTH + " and " +
MAX_PIECE_WIDTH;

        text_restriction_comment_height_sheet.text = "Between " + MIN_SHEET_HEIGHT + "
and " + MAX_SHEET_HEIGHT;
        text_restriction_comment_width_sheet.text = "Between " + MIN_SHEET_WIDTH + "
and " + MAX_SHEET_WIDTH;

        // Area text of piece set to MIN_HEIGHT*MIN_WIDTH
        text_piece_area.text = "Area: " + MIN_PIECE_HEIGHT * MIN_PIECE_WIDTH;


        panel_warning_and_help.SetActive(false);
        text_warning.text = "Warning: There are one or more pieces that do not fit in
the current conditions";


        // Area text set of sheet set to MAX*MAX
        text_sheet_remaining_capacity.text = "Capacity: " + MAX_SHEET_HEIGHT *
MAX_SHEET_WIDTH + " / " + MAX_SHEET_HEIGHT * MAX_SHEET_WIDTH + "  ( 100.00 % )";

        // Problem setup window objects' size, depending on pieces' max/min possible
size
        panel_piece_canvas.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_PIECE_WIDTH, MAX_PIECE_HEIGHT);
        scrollbar_piece_width.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_PIECE_WIDTH, 10);
        scrollbar_piece_height.GetComponent<RectTransform>().sizeDelta = new
Vector2(10, MAX_PIECE_HEIGHT);

        // Change scrollbars of canvas sheet depending of its size
        scrollbar_canvas_sheet_width.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_SHEET_WIDTH - MIN_SHEET_WIDTH+30, 10);
        scrollbar_canvas_sheet_height.GetComponent<RectTransform>().sizeDelta = new
Vector2(10, MAX_SHEET_HEIGHT - MIN_SHEET_HEIGHT+20);

        panel_canvas_sheet.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_SHEET_WIDTH+15, MAX_SHEET_HEIGHT+15);

        panel_sheet.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_SHEET_WIDTH, MAX_SHEET_HEIGHT);




        panel_sheet.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_SHEET_WIDTH, MAX_SHEET_HEIGHT);

        // Set up the virtual calculations for the rectangles:

        // The rectangle of the starting node is the one representing the whole sheet.
It will change if the sheet redimensionates
        start_node.rect = new Rectangle(0, 0,
panel_sheet.GetComponent<RectTransform>().rect.width,
panel_sheet.GetComponent<RectTransform>().rect.height);



        // Start the value of the piece to its minimum
        panel_piece.GetComponent<RectTransform>().sizeDelta = new
Vector2(MIN_PIECE_WIDTH, MIN_PIECE_HEIGHT);


        // Set current dimensions of the sheet to its maximum
        current_sheet_width = MAX_SHEET_WIDTH;
        current_sheet_height = MAX_SHEET_HEIGHT;

        // Start the value of the sheet to its maximum
        panel_sheet.GetComponent<RectTransform>().sizeDelta = new
Vector2(MAX_SHEET_WIDTH, MAX_SHEET_HEIGHT);
```

```
    }

    // Update is called once per frame
    void Update()
    {

    }

    // Starting window Delegated methods
    public void ValueChangeAction()
    {
        text_population_size.text = "Population size: " + slider_population_size.value;
    }

    public void starting_window_button_Clicked_Action()
    {
        canvas_starting_window.enabled = false;
        canvas_problem_setup_window.enabled = true;
    }


    // Problem setup window Delegated methods
    public void scrollbar_piece_height_onValueChanged()
    {
        // Change value of InputField
        float proportion = MIN_PIECE_HEIGHT / MAX_PIECE_HEIGHT;

        float normalized_value = (scrollbar_piece_height.value * (1 - proportion) +
proportion) * MAX_PIECE_HEIGHT;


        inputField_piece_height.text = normalized_value.ToString();

        // and change height of the piece
        RectTransform panel_piece_RectTransform =
panel_piece.GetComponent<RectTransform>();

        panel_piece_RectTransform.sizeDelta = new
Vector2(panel_piece_RectTransform.rect.width, normalized_value);

        // Recalculate area and change text
        text_piece_area.text = "Area: " + panel_piece_RectTransform.rect.width *
normalized_value;
    }



    public void scrollbar_piece_width_onValueChanged()
    {
        // Change value of InputField
        float proportion = MIN_PIECE_WIDTH / MAX_PIECE_WIDTH;
        float normalized_value = (scrollbar_piece_width.value * (1 - proportion) +
proportion) * MAX_PIECE_WIDTH;
        inputField_piece_width.text = normalized_value.ToString();

        // and change width of the piece
        RectTransform panel_piece_RectTransform =
panel_piece.GetComponent<RectTransform>();

        panel_piece_RectTransform.sizeDelta = new Vector2(normalized_value,
panel_piece_RectTransform.rect.height);

        // Recalculate area and change text
        text_piece_area.text = "Area: " + normalized_value *
panel_piece_RectTransform.rect.height;
    }



    public void inputField_piece_height_onEndEdit()
    {
        try
        {
            int value = Convert.ToInt32(inputField_piece_height.text);

            if (value < MIN_PIECE_HEIGHT)
            {
                value = (int) MIN_PIECE_HEIGHT;
                inputField_piece_height.text = MIN_PIECE_HEIGHT.ToString();
            }

            if (value > MAX_PIECE_HEIGHT)
            {
                value = (int) MAX_PIECE_HEIGHT;
                inputField_piece_height.text = MAX_PIECE_HEIGHT.ToString();
            }

            // Change scrollbar position according to value
            scrollbar_piece_height.value = ((float)value - 
MIN_PIECE_HEIGHT)/(MAX_PIECE_HEIGHT-MIN_PIECE_HEIGHT);

            // Change piece size according to value
            RectTransform panel_piece_RectTransform = 
panel_piece.GetComponent<RectTransform>();
            panel_piece_RectTransform.sizeDelta = new 
Vector2(panel_piece_RectTransform.rect.width, value);
        }

        catch (Exception)
        {
            Console.WriteLine("El valor introducido no es válido");
        }
    }


    public void inputField_piece_width_onEndEdit()
    {
        try
        {
            int value = Convert.ToInt32(inputField_piece_width.text);

            if (value < MIN_PIECE_WIDTH)
            {
                value = (int)MIN_PIECE_WIDTH;
                inputField_piece_width.text = MIN_PIECE_WIDTH.ToString();

            }

            if (value > MAX_PIECE_WIDTH)
            {
                value = (int)MAX_PIECE_WIDTH;
                inputField_piece_width.text = MAX_PIECE_WIDTH.ToString();
            }

            // Change scrollbar position according to value
            scrollbar_piece_width.value = ((float)value - MIN_PIECE_WIDTH) / 
(MAX_PIECE_WIDTH - MIN_PIECE_WIDTH);

            // Change piece size according to value
            RectTransform panel_piece_RectTransform = 
panel_piece.GetComponent<RectTransform>();
            panel_piece_RectTransform.sizeDelta = new Vector2(value, 
panel_piece_RectTransform.rect.height);
        }

        catch (Exception)
        {
            Console.WriteLine("El valor introducido no es válido");
        }
    }



    public void inputField_sheet_height_onEndEdit()
    {
        try
        {
            int value = Convert.ToInt32(inputField_sheet_height.text);

            if (value < MIN_SHEET_HEIGHT)
            {
                value = (int)MIN_SHEET_HEIGHT;
                inputField_sheet_height.text = MIN_SHEET_HEIGHT.ToString();
            }

            if (value > MAX_SHEET_HEIGHT)
            {
                value = (int)MAX_SHEET_HEIGHT;
                inputField_sheet_height.text = MAX_SHEET_HEIGHT.ToString();
            }

            // Change scrollbar position according to value
            scrollbar_canvas_sheet_height.value = ((float)value - MIN_SHEET_HEIGHT) /
(MAX_SHEET_HEIGHT - MIN_SHEET_HEIGHT);

            // Change sheet size according to value
            RectTransform panel_sheet_RectTransform =
panel_sheet.GetComponent<RectTransform>();
            panel_sheet_RectTransform.sizeDelta = new
Vector2(panel_sheet_RectTransform.rect.width, value);


        }

        catch (Exception)
        {
            Console.WriteLine("El valor introducido no es válido");
        }
    }


    public void inputField_sheet_width_onEndEdit()
    {
        try
        {
            int value = Convert.ToInt32(inputField_sheet_width.text);

            if (value < MIN_SHEET_WIDTH)
            {
                value = (int)MIN_SHEET_WIDTH;
                inputField_sheet_width.text = MIN_SHEET_WIDTH.ToString();

            }

            if (value > MAX_SHEET_WIDTH)
            {
                value = (int)MAX_SHEET_WIDTH;
                inputField_sheet_width.text = MAX_SHEET_WIDTH.ToString();
            }

            // Change scrollbar position according to value
            scrollbar_canvas_sheet_width.value = ((float)value - MIN_SHEET_WIDTH) /
(MAX_SHEET_WIDTH - MIN_SHEET_WIDTH);

            // Change sheet size according to value
            RectTransform panel_sheet_RectTransform =
panel_sheet.GetComponent<RectTransform>();
            panel_sheet_RectTransform.sizeDelta = new Vector2(value,
panel_sheet_RectTransform.rect.height);
        }

        catch (Exception)
        {
            Console.WriteLine("El valor introducido no es válido");
        }
    }


    public void scrollbar_canvas_sheet_height_onValueChanged()
    {
        RectTransform panel_sheet_RectTransform =
panel_sheet.GetComponent<RectTransform>();

        float proportion = MIN_SHEET_HEIGHT / MAX_SHEET_HEIGHT;
        float normalized_value = ((scrollbar_canvas_sheet_height.value * (1 -
proportion) + proportion) * MAX_SHEET_HEIGHT);

        // Change value of InputField
        inputField_sheet_height.text = normalized_value.ToString();

        // change height of the sheet
        panel_sheet_RectTransform.sizeDelta = new
Vector2(panel_sheet_RectTransform.rect.width, normalized_value);

        // change height of the rect in node
        start_node.rect = new Rectangle(0, 0,
panel_sheet.GetComponent<RectTransform>().rect.width, normalized_value);

        // Change value of current sheet height
        current_sheet_height = normalized_value;


        // change text of remaining capacity:
        update_capacity_text();



        // change text of total sheet area

        if(automatic_rearrange_upon_resize) rearrange();


    }

    public void scrollbar_canvas_sheet_width_onValueChanged()
    {
        RectTransform panel_sheet_RectTransform =
panel_sheet.GetComponent<RectTransform>();

        float proportion = MIN_SHEET_WIDTH / MAX_SHEET_WIDTH;
        float normalized_value = ((scrollbar_canvas_sheet_width.value * (1 -
proportion) + proportion) * MAX_SHEET_WIDTH);

        // Change value of InputField
        inputField_sheet_width.text = normalized_value.ToString();

        // change height of the sheet
        panel_sheet_RectTransform.sizeDelta = new Vector2(normalized_value,
panel_sheet_RectTransform.rect.height);

        // Change value of current sheet height
        current_sheet_width = normalized_value;

        // change text of remaining capacity:
        update_capacity_text();

        // change text of total sheet area
        if (automatic_rearrange_upon_resize) rearrange();

    }

    public void button_add_piece_onClick()
    {


        // Make another instance of the piece to be added
        Debug.Log("Size of sheet canvas -> Width:" +
panel_sheet.GetComponent<RectTransform>().rect.width + "... Height: " +
panel_sheet.GetComponent<RectTransform>().rect.height);
        GameObject piece_to_add = Instantiate(panel_piece);
        piece_to_add.transform.SetParent(panel_sheet.transform);

        set_tag_to_piece(piece_to_add, list_pieces.Count + 1);



        // Create virtual rectangle
        Rectangle piece_to_add_RECT = new Rectangle(0, 0,
piece_to_add.GetComponent<RectTransform>().rect.width,
piece_to_add.GetComponent<RectTransform>().rect.height);

        Node node = start_node.insert_rect(piece_to_add_RECT);
        if (node != null)
        {
            Rectangle r = node.rect;
            piece_to_add.transform.localPosition = new Vector2(r.x, r.y);



            // The piece must be only added if it's added to the tree
            list_pieces.Add(piece_to_add);

            Debug.Log("The added piece has its starting point at: ( " + r.x + ", " +
r.y + "), its upper-right corner at " + get_upper_right_position(piece_to_add) + ", its
lower-left corner at " + get_lower_left_position(piece_to_add) + ", and its lower-right
corner at " + get_lower_right_position(piece_to_add));
        }

        update_capacity_text();

    }

    // Gets the point (relative coordinates to GameObject's parent) of the upper left
corner, i.e, (x, y)
    public Vector2 get_upper_left_position(GameObject piece)
    {
        return new Vector2(piece.transform.localPosition.x,
piece.transform.localPosition.y);
    }

    public Vector2 get_upper_right_position(GameObject piece)
    {
        return new Vector2(piece.transform.localPosition.x +
piece.GetComponent<RectTransform>().rect.width, piece.transform.localPosition.y);
    }

    public Vector2 get_lower_left_position(GameObject piece)
    {
        return new Vector2(piece.transform.localPosition.x,
piece.transform.localPosition.y - piece.GetComponent<RectTransform>().rect.height);
    }

    public Vector2 get_lower_right_position(GameObject piece)
    {
        return new Vector2(piece.transform.localPosition.x +
piece.GetComponent<RectTransform>().rect.width, piece.transform.localPosition.y -
piece.GetComponent<RectTransform>().rect.height);
    }




    public void slider_RGB_R_onValueChanged()
    {
        Image panel = panel_piece.GetComponent<Image>();
        panel.color = new Color(slider_RGB_R.value, panel.color.g, panel.color.b);
    }

    public void slider_RGB_G_onValueChanged()
    {
        Image panel = panel_piece.GetComponent<Image>();
        panel.color = new Color(panel.color.r, slider_RGB_G.value, panel.color.b);
    }

    public void slider_RGB_B_onValueChanged()
    {
        Image panel = panel_piece.GetComponent<Image>();
        panel.color = new Color(panel.color.r, panel.color.g, slider_RGB_B.value);
    }


    public void update_capacity_text()
    {
        float total_current_area = current_sheet_width * current_sheet_height;
        float pieces_area = 0;

        foreach(GameObject piece in list_pieces)
        {
            pieces_area += piece.GetComponent<RectTransform>().rect.width * 
piece.GetComponent<RectTransform>().rect.height;
        }

        float remaining_area = total_current_area - pieces_area;
        float percentage = remaining_area / total_current_area;

        text_sheet_remaining_capacity.text = "Capacity: " + remaining_area + " / " + 
total_current_area + "  ( " + String.Format("{0:0.00}", percentage*100) + " % )";
    }

    public void rearrange()
    {
        bool all_pieces_have_been_put = true;

        clear_sheet();

        foreach (GameObject piece in list_pieces)
        {

            // Create virtual rectangle
            Rectangle piece_to_add_RECT = new Rectangle(0, 0, 
piece.GetComponent<RectTransform>().rect.width, 
piece.GetComponent<RectTransform>().rect.height);

            Node node = start_node.insert_rect(piece_to_add_RECT);


            if (node != null)
            {
                Rectangle r = node.rect;
                piece.transform.localPosition = new Vector2(r.x, r.y);
                piece.SetActive(true);
            }

            else
            {
                panel_warning_and_help.SetActive(true);
                piece.SetActive(false);
                all_pieces_have_been_put = false;
            }

            if (all_pieces_have_been_put) panel_warning_and_help.SetActive(false);

            }

    }


    public void clear_sheet()
    {
        foreach (GameObject piece in list_pieces)
        {
            // Disable pieces from sheet
            piece.SetActive(false);
        }

        // Restart the tree that represents the space
        start_node = new Node();
        // Reinitialize the space in that node with the current sheet proportions
        start_node.rect = new Rectangle(0, 0, 
panel_sheet.GetComponent<RectTransform>().rect.width, 
panel_sheet.GetComponent<RectTransform>().rect.height);
        panel_warning_and_help.SetActive(false);
    }

    public void button_sheet_clear_onClick()
    {
        clear_sheet();
        list_pieces.Clear();
    }

    public void button_sheet_rearrange_onClick()
    {
        rearrange();
    }


    public void set_tag_to_piece(GameObject piece_to_add, int id)
    {
        // Create text to identify the piece
        GameObject text_piece_id = new GameObject();
        text_piece_id.AddComponent<Text>();
        // Set size as the piece
        text_piece_id.GetComponent<RectTransform>().sizeDelta = new
Vector2(piece_to_add.GetComponent<RectTransform>().rect.width,
piece_to_add.GetComponent<RectTransform>().rect.height);
        text_piece_id.GetComponent<Text>().text = id.ToString();
        text_piece_id.GetComponent<Text>().font =
Resources.GetBuiltinResource(typeof(Font), "Arial.ttf") as Font;
        text_piece_id.GetComponent<Text>().alignment = TextAnchor.MiddleCenter;
        text_piece_id.transform.SetParent(piece_to_add.transform);
        // Set position as the piece so they perfectly overlap
        text_piece_id.transform.localPosition = new
Vector2(piece_to_add.GetComponent<RectTransform>().rect.width / 2, -
piece_to_add.GetComponent<RectTransform>().rect.height / 2);
    }


}
```

Script 2. Genetic_Process_Script.cs

Descripción: El segundo *script* se encarga principalmente de las acciones ocurridas en la segunda ventana de la aplicación, es decir, se encarga de procesar el algoritmo genético y mostrar los resultados obtenidos.

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System;
using System.Collections.Generic;


public class Rectangle_Node
{
    public Rectangle_Node left;
    public Rectangle_Node right;
    public Rectangle rect;

    public String integration_type;



    public Rectangle_Node(Rectangle rect)
    {
        this.rect = rect;
    }

    public void set_right_child(Rectangle_Node node)
    {
        right = node;
    }

    public void get_right_child(Rectangle_Node node)
    {
        right = node;
    }

    public void set_left_child(Rectangle_Node node)
    {
        left = node;
    }

    public void get_left_child(Rectangle_Node node)
    {
        left = node;
    }

    public String get_integration_type()
    {
        return integration_type;
    }

    public void set_integration_type(String type)
    {
        if (type != "H" || type != "V")
        {
            throw new System.ArgumentException("The call of the method
set_integration_type() does not admit a type that is not H or V: The parameter is set
to: " + type);
        }
        integration_type = type;
    }


    public bool is_integrated(Rectangle_Node node)
    {
        return left == null ? false : true;
    }
```

```
}

public class Chromosome
{
    public List<String> gen_list;
    public double fitness;
    int length;
    private Rectangle_Node tree;


    // Constructors

    // Void constructor
    public Chromosome()
    {

    }

    public void set_phenotype(Rectangle_Node tree)
    {
        this.tree = tree;
    }

    public Rectangle_Node get_phenotype()
    {
        return tree;
    }

    // Defined constructor
    public Chromosome(List<String> chromosome)
    {
        gen_list = chromosome;
    }

    // Introduces TO the chromosome -> List<String>
    // a random PART gene taken from the list
    public void introduce_random_part_gene(List<String> my_chromosome, List<String>
my_list_of_available_pieces, System.Random rnd_obj)
    {


        // Select the piece
        int piece_number = rnd_obj.Next(0, my_list_of_available_pieces.Count);
        String piece_gen = my_list_of_available_pieces[piece_number];

        // Add the piece to the chromosome
        my_chromosome.Add(piece_gen);

        // Remove the piece from the list of pieces so it does not get added again in
the future
        my_list_of_available_pieces.Remove(piece_gen);
    }

    // Introduces at the end of the chromosome, an operator gene
    public void introduce_random_operator_gene(List<String> my_chromosome, System.Random
rnd_obj)
    {


        // Select the operator
        int random_choice = rnd_obj.Next(1, 3);

        if (random_choice == 1)
        {
            my_chromosome.Add("H");
        }
        if (random_choice == 2)
        {
            my_chromosome.Add("V");
        }
    }


    // Random constructor (takes length as argument)
    // Creates a valid, yet random chromosome
    public Chromosome(int length)
    {
        List<String> chromosome = new List<String>();

        // Given a length (number of genes in the chromosome), we know the exact number
of pieces, since [ Ng = Np + No = 2Np - 1 ] -> Np = (Ng + 1) / 2

        int ng = length; // Number of genes in the chromosome
        int np = (ng + 1) / 2; // Number of pieces
        int no = ng - np; // Number of operators


        // Generate a list which will contain the pieces
        List<String> list_pieces = new List<String>();

        System.Random rnd = new System.Random();

        // Fill list with piece numbers
        for (int i = 1; i <= np; i++)
        {
            list_pieces.Add(i.ToString());
        }

        // The first two pieces will always be two part numbers

        introduce_random_part_gene(chromosome, list_pieces, rnd);
        introduce_random_part_gene(chromosome, list_pieces, rnd);
        introduce_random_operator_gene(chromosome, rnd);



        // From now on, we need to complete the chromosome while satisfying the
restrictions

        // For the next empty genes of the chromosome
        for (int i = chromosome.Count; i < ng; i++)
        {
            // Check if no >= np+1 (only on the left)
            int actual_np = 0;
            int actual_no = 0;

            int n;
            for (int j = 0; j < chromosome.Count; j++)
            {
                if (Genetic_Process_Script.is_a_number(chromosome[j], out n))
                    actual_np++;
                else actual_no++;
            }

            // Once counted, we decide:
            if (no == np + 1)
            {
                // Limit of the restriction, we need to put a piece
                introduce_random_part_gene(chromosome, list_pieces, rnd);
            }

            // If the restriction is not on its limit, then we put with a 50% chance a
part or an operator, if there is still pieces
            else
            {
                // if there is not pieces left, fill with operator

                if (list_pieces.Count == 0)
                {
                    introduce_random_operator_gene(chromosome, rnd);




                }
                else
                {

                    // 50% for part or operator
                    int random_choice = rnd.Next(1, 3);
                    // introduce part
                    if (random_choice == 1)
                    {
                        introduce_random_part_gene(chromosome, list_pieces, rnd);
                    }
                    // introduce operator
                    if (random_choice == 2)
                    {
                        introduce_random_operator_gene(chromosome, rnd);


                        // The built chromosome could not be valid, we must check:
                        try
                        {
                            Stack<System.Object> test_stack = new
Stack<System.Object>();
                            build_stack_from_chromosome_simple(chromosome, test_stack);
                        }
                        catch (Exception)
                        {
                            chromosome.RemoveAt(chromosome.Count-1);

                            introduce_random_part_gene(chromosome, list_pieces, rnd);
                        }


                    }

                }
            }
        }


        gen_list = chromosome;
    }




    // Setters and getters
    public void set_fitness(double new_fitness)
    {
        fitness = new_fitness;
    }

    public double get_fitness()
    {
        return fitness;
    }



    // Stack operations to resemble stack building process. Needed to check if the
random chromosome is valid
    public void stack_simple(Stack<System.Object> my_stack)
    {
        System.Object o = new System.Object();
        my_stack.Push(o);
    }


    public void unstack_operate_and_stack_simple(Stack<System.Object> my_stack)
    {
        // Unstack
        System.Object first_node = my_stack.Pop();
        System.Object second_node = my_stack.Pop();

        System.Object o = new System.Object();
        my_stack.Push(o);


    }

    public void build_stack_from_chromosome_simple(List<String> chromosome,
Stack<System.Object> my_stack)
```

```
    {

        for (int i = 0; i < chromosome.Count; i++)
        {

            int n = -1;
            // if the gen of the chromosome is a part number, we need to retrieve that
number
            if (Genetic_Process_Script.is_a_number(chromosome[i], out n))
            {
                stack_simple(my_stack);
            }
            else // not a number -> V or H
                unstack_operate_and_stack_simple(my_stack);
        }
    }


}

public class Population
{
    public int population_size;
    public double mutation_chance;
    public int generation;

    public List<Chromosome> population_list = new List<Chromosome>();

    // Creates a new population with its individuals randomly generated
    public Population(double mutation_chance, int population_size, int gene_length)
    {
        this.mutation_chance = mutation_chance;
        this.population_size = population_size;


        for(int i = 0; i < population_size; i++)
        {
            population_list.Add(new Chromosome(gene_length));
        }
    }

    // Empty constructor. Creates population without chromosomes
    public Population(double mutation_chance, int population_size)
    {

        this.mutation_chance = mutation_chance;
        this.population_size = population_size;
    }


    public static List<Chromosome> crossover(Chromosome parent_1, Chromosome parent_2)
    {
        List<Chromosome> children = new List<Chromosome>();
        Chromosome child_1 = new Chromosome();
        Chromosome child_2 = new Chromosome();

        List<String> parent_1_gen_list = parent_1.gen_list;
        List<String> parent_2_gen_list = parent_2.gen_list;

        // Part lists will not have any gaps and will only represent order relationships
between parts
        List<String> parent_1_part_list = new List<String>();
        List<String> parent_2_part_list = new List<String>();

        // Operator lists will have gaps in which we put the part numbers
        List<String> parent_1_op_list = new List<String>();
        List<String> parent_2_op_list = new List<String>();




        // First, separate part numbers from operators of both parents
        int n;
        foreach (String gen in parent_1_gen_list)
        {
            // if is a number
```

```
           if (Genetic_Process_Script.is_a_number(gen, out n))
            {
                // add that part number to part list
                parent_1_part_list.Add(n.ToString());

                // and add nothing to the op list
                parent_1_op_list.Add("");
            }
            else
            {
                // and add op letter to op list
                parent_1_op_list.Add(gen);
            }
        }

        // Do the same for the other parent
        foreach (String gen in parent_2_gen_list)
        {
            // if is a number
            if (Genetic_Process_Script.is_a_number(gen, out n))
            {
                // add that part number to part list
                parent_2_part_list.Add(n.ToString());

                // and add nothing to the op list
                parent_2_op_list.Add("");
            }
            else
            {
                // and add op letter to op list
                parent_2_op_list.Add(gen);
            }
        }



                // Create the dictionary beforehand only by going through the part list
and adding the strings as keys

                Dictionary<string, List<string>> mapping_relationships = new
Dictionary<string, List<string>>();

                // Initialize dictionary with empty lists on values for keys
                foreach (string el in parent_1_part_list)
                {
                    mapping_relationships.Add(el, new List<string>());
                }



        // Now, we must choose two numbers that will delimit the random segment limits
of the part list that will be swapped between parents to form child
        // 1st number will be from 0 to part list length - 2
        // 2nd number will be from 1st number to part list length - 1

        System.Random rnd = new System.Random();

        int first = rnd.Next(0, parent_1_part_list.Count - 2);
        int second = rnd.Next(first, parent_1_part_list.Count - 1);



        // crossover for the operators

        // Determine which operators of the first will be swapped for the operators in
the second
        List<bool> to_be_changed = new List<bool>();
        int count = 0; // this will count the pieces

        for (int i = 0; i < parent_1_op_list.Count; i++)
        {
            if (parent_1_op_list[i] == "") count++;
            if (count >= first && count < second && parent_1_op_list[i] != "")
to_be_changed.Add(true);
            else to_be_changed.Add(false);
        }

        int number_of_operators_found;
        string aux_2;

        for(int i = 0; i < parent_1_op_list.Count; i++)
        {
            if (to_be_changed[i])
            {
                // count the number of operators that the gene has until there
                number_of_operators_found = 0;
                for(int j = 0; j <= i; j++)
                {
                    if (parent_1_op_list[j] != "") number_of_operators_found++;
                }

                // once counted, we get the one that we will be changed in the second
parent
                for(int j = 0; j < parent_2_op_list.Count; j++)
                {
                    // first, check if it is an operator, if it is an operator, take 1
from found
                    if (parent_2_op_list[j] != "") number_of_operators_found--;
                    if (number_of_operators_found == 0)
                    {
                        // swap
                        aux_2 = parent_1_op_list[i];
                        parent_1_op_list[i] = parent_2_op_list[j];
                        parent_2_op_list[j] = aux_2;
                        break;
                    }


                }


            }
        }

            // crossover for the part numbers

            // Exchange subgroup between parents
            string aux;

        for (int i = first; i <= second; i++)
        {
            // Save the element of 1 to aux
            aux = parent_1_part_list[i];

            // Remove the element of 1
            parent_1_part_list.RemoveAt(i);

            // To insert the element of 2 in that place
            parent_1_part_list.Insert(i, parent_2_part_list[i]);

            // Remove the element of 2
            parent_2_part_list.RemoveAt(i);

            // To insert the element of 1 (that is in aux) in that place
            parent_2_part_list.Insert(i, aux);
        }



        // Now they are swapped, and we must legalize them
        // Determine mapping relationship. Now we will use the dictionary
        for (int i = first; i <= second; i++)
        {
            mapping_relationships[parent_1_part_list[i]].Add(parent_2_part_list[i]);
            mapping_relationships[parent_2_part_list[i]].Add(parent_1_part_list[i]);
        }

        Utilities.unfold_relationships(mapping_relationships);


        // The elements from 0 to first (not included) and from second + 1 to count (not
included) will change* if they are in the range [first, second]
        //      * They will be changed to a number of its related that are not in the
whole gene


        for (int i = 0; i < first; i++) // outer left
        {
            for(int j = first; j <= second; j++) // inner
            {
                if (parent_1_part_list[i] == parent_1_part_list[j]) // it means the
inner one should be swapped by an element of relationships, that is not in the string
                {
                    // for each element of the relationship
                    foreach(string el in mapping_relationships[parent_1_part_list[j]])
                    {
                        // if that element is not in the gene, then swap it
                        if (!parent_1_part_list.Contains(el))
                        {
                            // in "el" we have the element that can be swapped with the
"repeated number in 1", we must find it in outter 2 to make the swap
                            for(int k = 0; k < first; k++)
                            {
                                if(el == parent_2_part_list[k])
                                {
                                    // Save the element of 1 to aux
                                    aux = parent_1_part_list[i];

                                    parent_1_part_list[i] = el;
                                    parent_2_part_list[k] = aux;
                                }
                            }

                            // check also at the right side of the selection
                            for (int l = second + 1; l < parent_2_part_list.Count; l++)
                            {
                                if (el == parent_2_part_list[l])
                                {
                                    // Save the element of 1 to aux
                                    aux = parent_1_part_list[i];

                                    parent_1_part_list[i] = el;
                                    parent_2_part_list[l] = aux;
                                }
                            }
                        }
                    }
                }
            }
        }

        for (int i = second + 1; i < parent_1_part_list.Count; i++) // outer right
        {
            for (int j = first; j <= second; j++) // inner
            {
                if (parent_1_part_list[i] == parent_1_part_list[j]) // it means the
inner one should be swapped by an element of relationships, that is not in the string
                {
                    // for each element of the relationship
                    foreach (string el in mapping_relationships[parent_1_part_list[j]])
                    {
                        // if that element is not in the gene, then swap it
                        if (!parent_1_part_list.Contains(el))
                        {
                            // in "el" we have the element that can be swapped with the
"repeated number in 1", we must find it in outter 2 to make the swap
                            for (int k = 0; k < first; k++)
                            {
                                if (el == parent_2_part_list[k])
                                {
                                    // Save the element of 1 to aux
                                    aux = parent_1_part_list[i];

                                    parent_1_part_list[i] = el;
                                    parent_2_part_list[k] = aux;
                                }
                            }
                            // check also at the right side of the selection
                            for (int l = second + 1; l < parent_2_part_list.Count; l++)
                            {
                                if (el == parent_2_part_list[l])
                                {
                                    // Save the element of 1 to aux
                                    aux = parent_1_part_list[i];

                                    parent_1_part_list[i] = el;
                                    parent_2_part_list[l] = aux;
                                }
                            }
                        }
                    }
                }
            }
        }



        // Now we must refill the operator lists and they will form the genes of the
offsprings
        for(int i = 0; i < parent_1_op_list.Count; i++)
        {
            if (parent_1_op_list[i] == "")
            {
                parent_1_op_list[i] = parent_1_part_list[0];
                parent_1_part_list.RemoveAt(0);
            }
        }

        for (int i = 0; i < parent_2_op_list.Count; i++)
        {
            if (parent_2_op_list[i] == "")
            {
                parent_2_op_list[i] = parent_2_part_list[0];
                parent_2_part_list.RemoveAt(0);
            }
        }



        child_1.gen_list = parent_1_op_list;
        child_2.gen_list = parent_2_op_list;

        children.Add(child_1);
        children.Add(child_2);

        return children;
    }


    public static bool mutation(Chromosome myChr)
    {
        List<string> gene = myChr.gen_list;
        System.Random rnd = new System.Random();

        // Choose 2 random different gene positions
        int first = rnd.Next(0, gene.Count);
        int second = rnd.Next(0, gene.Count);



        while (first == second)
        {
            second = rnd.Next(0, gene.Count);
        }


        // If first is not the left one, we force it
        if (first > second)
        {
            int aux = first;
            first = second;
            second = aux;
        }

        // Now, we must check what are they

        // If they are both part numbers, we can switch without problems
        if (Utilities.is_a_number(gene[first]) && Utilities.is_a_number(gene[second]))
        {
            string aux = gene[first];
            gene[first] = gene[second];
            gene[second] = aux;
        }

        else if (!Utilities.is_a_number(gene[first]))
        {
            string aux = gene[first];
            gene[first] = gene[second];
            gene[second] = aux;
        }

        // if p1 is a part number and p2 is an operator
        else if (Utilities.is_a_number(gene[first]) && 
!Utilities.is_a_number(gene[second]))
        {

            int operators = 0;
            int parts = 0;

            // we should check that every position between p1 and p2 verifies  No <= Np 
- 3
            for (int i = 0; i < gene.Count; i++)
            {
                if (Utilities.is_a_number(gene[i])) parts++;
                else operators++;

                if(i >= first && i <= second)
                {
                    if(! (operators <= parts - 3))
                    {

                        return false;
                    }
                }
            }
            // swap is possible
            string aux = gene[first];
            gene[first] = gene[second];
            gene[second] = aux;
        }

        myChr.gen_list = gene;

        return true;

    }

    public static int compare_fitness(Chromosome chr1, Chromosome chr2)
    {
        return chr1.fitness.CompareTo(chr2.fitness);
    }

    public void sort_population_by_fitness()
    {
        Comparison<Chromosome> comp_chr = new 
Comparison<Chromosome>(Population.compare_fitness);
        population_list.Sort(comp_chr);
        population_list.Reverse();
    }




    public void compute_one_generation()
    {
        double sum_of_fitnesses = 0;
        System.Random nature = new System.Random();

        Population survivors = new Population(mutation_chance, population_size);
        generation++;

        Chromosome parent_1 = new Chromosome();
        Chromosome parent_2 = new Chromosome();
        Chromosome offspring_1;
        Chromosome offspring_2;
        List<Chromosome> offspring_list;

        // elitist for first n chromosomes

        for (int i = 0; i < 1; i++)
        {
            survivors.population_list.Add(population_list[i]);
        }


        //  Calculate sum of all chromosome fitnesses in population  sum S.
        foreach (Chromosome chr in population_list)
        {
            sum_of_fitnesses += chr.fitness;
        }



        while (survivors.population_list.Count < population_size)
        {

            // Generate random number from interval (0,S) - r
            double random = Utilities.get_random_double_between(0, sum_of_fitnesses,
nature);

            double s = 0;

            // Go through the population and sum fitnesses from 0 - sum s. When the sum
s is greater than r, stop and return the chromosome where you are.
            foreach (Chromosome chr in population_list)
            {
                s += chr.fitness;

                if (s > random)
                {
                    parent_1 = chr;
                    break;
                }

            }


            // do all again for parent 2

            random = Utilities.get_random_double_between(0, sum_of_fitnesses, nature);
            s = 0;

            // Go through the population and sum fitnesses from 0 - sum s. When the sum
s is greater than r, stop and return the chromosome where you are.
            foreach (Chromosome chr in population_list)
            {
                s += chr.fitness;

                if (s > random)
                {
                    parent_2 = chr;
                    break;
                }

            }




            // once we have both parents, we crossover and get two offsprings
            offspring_list = crossover(parent_1, parent_2);
            offspring_1 = offspring_list[0];
            offspring_2 = offspring_list[1];

            //survivors.population_list.Add(parent_1);
            //survivors.population_list.Add(parent_2);
            survivors.population_list.Add(offspring_1);
            survivors.population_list.Add(offspring_2);


        }





        for (int i = 2; i < survivors.population_list.Count; i++)
        {
            if (nature.NextDouble() < mutation_chance)
            {
                while (!mutation(survivors.population_list[i])) ;

                //mutation(survivors.population_list[i]);
            }
        }


        population_list = survivors.population_list;

    }
}



    public class Genetic_Process_Script : MonoBehaviour {

    public List<GameObject> list_pieces;

    public Button button_start_process;

    public Canvas canvas_problem_setup_window;

    public Canvas canvas_genetic_results_window;

    public GameObject panel_canvas_sheet_process_window;
    public GameObject panel_sheet_process_window;

    public List<String> chromosome;

    // Declare variables of the sheet's size
    public float current_sheet_width;
    public float current_sheet_height;

    public Stack<Rectangle_Node> node_Stack;

    public Text text_generation_number;
    public Text text_fitness_of_best;
    public Text text_time;

    public Button my_button;
    public Button button_compute_next_generation;
    public Button button_compute_50_generations;
    public Button button_debug_population;
    public Button button_compute_until_change;

    public InputField field_min_rect_factor;
    public InputField field_sq_factor;
    public InputField field_mutation_chance;
    public InputField field_population_size;

    public Population my_population;

    double time_elapsed;
    void Start()
    {
        // Starting delegate
        button_start_process.onClick.AddListener(delegate {
button_start_process_onClick(); });

    }


    // Use THIS for initialization!
    public void button_start_process_onClick()
    {
        // Share list of pieces (gameobjects)
        list_pieces =
GameObject.Find("Script_Holder").GetComponent<App_Script>().list_pieces;

        // Switch window
        canvas_problem_setup_window.enabled = false;
        canvas_genetic_results_window.enabled = true;

        // Share sheet values through scripts
        current_sheet_width =
GameObject.Find("Script_Holder").GetComponent<App_Script>().current_sheet_width;
        current_sheet_height =
GameObject.Find("Script_Holder").GetComponent<App_Script>().current_sheet_height;

        // Share canvas of the sheet values through scripts
        GameObject canvas_sheet =
GameObject.Find("Script_Holder").GetComponent<App_Script>().panel_canvas_sheet;
        float current_canvas_width =
canvas_sheet.GetComponent<RectTransform>().rect.width;
        float current_canvas_height =
canvas_sheet.GetComponent<RectTransform>().rect.height;

        my_button.onClick.AddListener(delegate { my_button_onClick(); });
        button_compute_next_generation.onClick.AddListener(delegate {
button_compute_next_generation_onClick(); });
        button_compute_50_generations.onClick.AddListener(delegate {
button_compute_50_generations_onClick(); });
        button_debug_population.onClick.AddListener(delegate {
button_debug_population_onClick(); });
        button_compute_until_change.onClick.AddListener(delegate {
button_compute_until_change_onClick(); });


        // And change size
        panel_canvas_sheet_process_window.GetComponent<RectTransform>().sizeDelta = new
Vector2(current_canvas_width, current_canvas_height);

        // Change sheet size accordingly to the set in the previous window
        panel_sheet_process_window.GetComponent<RectTransform>().sizeDelta = new
Vector2(current_sheet_width, current_sheet_height);


        time_elapsed = 0;

        int chromosome_length = list_pieces.Count * 2 - 1;

        // Create population

        my_population = new
Population(Double.Parse(field_mutation_chance.text),int.Parse(field_population_size.text
), chromosome_length);
        my_population.generation = 0;

        foreach (Chromosome chrom in my_population.population_list)
        {
            calculate_chromosome_phenotype_and_set_phenotype(chrom);
        }

        my_population.sort_population_by_fitness();

        text_fitness_of_best.text = "Fitness: " +
my_population.population_list[0].fitness.ToString();
```

```
        // Take out the best and draw it:
        Chromosome best = my_population.population_list[0];


        draw_pieces(best.get_phenotype(), panel_sheet_process_window);





    }

    public void my_button_onClick()
    {

        Chromosome test = new Chromosome(new List<string>() { "1", "5", "H", "7", "6",
"V", "2", "H", "4", "H", "V", "9", "8", "V", "3", "V", "H" });

        // stack initialization
        node_Stack = new Stack<Rectangle_Node>();
        build_stack_from_chromosome(test.gen_list);
        // pop the tree once it is built
        Rectangle_Node tree = node_Stack.Pop();

        // once the tree is built, now we can set the positions of the pieces depending
of their relationship with other pieces (H or V)
        calculate_positions(tree);

        calculate_and_set_fitness(test, tree, list_pieces);

        test.fitness = test.fitness * Math.Pow(10, 6);

        // draws the pieces on the sheet given the layout in tree
        draw_pieces(tree, panel_sheet_process_window);

    }

    public void button_compute_50_generations_onClick()
    {

        var watch = System.Diagnostics.Stopwatch.StartNew();
        // the code that you want to measure comes here


        for (int i = 0; i < 1000; i++)
        {
            my_population.compute_one_generation();
            text_generation_number.text = "Generation " + my_population.generation;

            foreach (Chromosome chrom in my_population.population_list)
            {
                calculate_chromosome_phenotype_and_set_phenotype(chrom);
            }


        }

        my_population.sort_population_by_fitness();

        watch.Stop();
        var elapsedMs = watch.ElapsedMilliseconds;

        text_time.text = "Time: " + elapsedMs + " ms.";



        text_fitness_of_best.text = "Fitness: " +
my_population.population_list[0].fitness.ToString();




        // Take out the best and draw it:
        Chromosome best = my_population.population_list[0];
        draw_pieces(best.get_phenotype(), panel_sheet_process_window);


    }


    public void button_debug_population_onClick()
    {
        for (int i = 0; i < my_population.population_list.Count; i++)
        {
            Debug.Log("Chromosome number " + i + ": FITNESS = " + 
my_population.population_list[i].fitness + ", GENOTYPE -> " + 
Utilities.list_toString(my_population.population_list[i].gen_list));
        }
    }


    public void button_compute_next_generation_onClick()
    {


        my_population.compute_one_generation();
        text_generation_number.text = "Generation " + my_population.generation;


        var watch = System.Diagnostics.Stopwatch.StartNew();
        // the code that you want to measure comes here

        foreach (Chromosome chrom in my_population.population_list)
        {
            calculate_chromosome_phenotype_and_set_phenotype(chrom);
        }

        my_population.sort_population_by_fitness();

        watch.Stop();
        var elapsedMs = watch.ElapsedMilliseconds;

        text_time.text = "Time to calculate_chromosome_phenotype_and_set_phenotype: " + 
elapsedMs + " ms.";




        text_fitness_of_best.text = "Fitness: " + 
my_population.population_list[0].fitness.ToString();




        // Take out the best and draw it:
        Chromosome best = my_population.population_list[0];

        // stack initialization
        node_Stack = new Stack<Rectangle_Node>();
        // we build the stack depending on how the chromosome specifies it
        build_stack_from_chromosome(best.gen_list);
        // pop the tree once it is built
        Rectangle_Node tree_best = node_Stack.Pop();
        // once the tree is built, now we can set the positions of the pieces depending 
of their relationship with other pieces (H or V)
        calculate_positions(tree_best);
        calculate_and_set_fitness(best, tree_best, list_pieces);
        draw_pieces(tree_best, panel_sheet_process_window);

    }


    public void button_compute_until_change_onClick()
    {
        // get current best fitness
        double best_fitness_old = my_population.population_list[0].fitness;
        double best_fitness_new = my_population.population_list[0].fitness;
        var watch = System.Diagnostics.Stopwatch.StartNew();
        // the code that you want to measure comes here

        int i = 0;
        while (best_fitness_old == best_fitness_new && i < 10000)
        {
            my_population.compute_one_generation();

            text_generation_number.text = "Generation " + my_population.generation;

            foreach (Chromosome chrom in my_population.population_list)
            {
                calculate_chromosome_phenotype_and_set_phenotype(chrom);
            }

            my_population.sort_population_by_fitness();

            best_fitness_new = my_population.population_list[0].fitness;

            i++;
        }

        watch.Stop();
        var elapsedMs = watch.ElapsedMilliseconds;

        text_time.text = "Time: " + elapsedMs + " ms.";

        text_fitness_of_best.text = "Fitness: " + 
my_population.population_list[0].fitness.ToString();

        // if it goes out of the loop, it means a best solution has been found, draw it 
and end
        draw_pieces(my_population.population_list[0].get_phenotype(), 
panel_sheet_process_window);

    }



    public void calculate_chromosome_phenotype_and_set_phenotype(Chromosome chr)
    {
        // stack initialization
        node_Stack = new Stack<Rectangle_Node>();
        // we build the stack depending on how the chromosome specifies it
        build_stack_from_chromosome(chr.gen_list);
        // pop the tree once it is built
        Rectangle_Node tree = node_Stack.Pop();
        // once the tree is built, now we can set the positions of the pieces depending 
of their relationship with other pieces (H or V)
        calculate_positions(tree);
        chr.set_phenotype(tree);
        calculate_and_set_fitness(chr, tree, list_pieces);
    }

    // Stacks the element that is in the position "id" of "list_pieces" onto the stack, 
as a node (tree)
    // This method gets a List of GameObjects as paramater, it is expected to be the 
list of pieces

    // It does not calculate positions, since they are calculated at the end, when the 
tree is fully built
    // The ID is meant to be the position of the piece in the list, starting its index 
at 0
    public void stack(List<GameObject> list_pieces, int id)
    {
        // Get dimensions of the piece
        float rect_width = list_pieces[id].GetComponent<RectTransform>().rect.width;
        float rect_height = list_pieces[id].GetComponent<RectTransform>().rect.height;

        // Create virtual rectangle given the piece GameObject
        Rectangle piece_rectangle = new Rectangle(0, 0, rect_width, rect_height);
        piece_rectangle.set_id(id);

        // Create the node with the virtual rectangle and no childs
        Rectangle_Node node = new Rectangle_Node(piece_rectangle);
        // Push the node onto the stack
        node_Stack.Push(node);
     }

    // Unstacks the last two nodes, operates on them with the given operator, and stacks
the result, as an integrated rectangle
    public void unstack_operate_and_stack(String op)
    {
        // Unstack two nodes to get the dimensions of their rectangles
        Rectangle_Node first_node = node_Stack.Pop();
        Rectangle_Node second_node = node_Stack.Pop();

        Rectangle integrated_piece;

        // The procedure depends on the type of operator given
        if (op == "H")
        {
            float integrated_width = first_node.rect.w + second_node.rect.w;
            float integrated_height = Math.Max(first_node.rect.h, second_node.rect.h);
            integrated_piece = new Rectangle(0, 0, integrated_width, integrated_height);
        }

        else if (op == "V")
        {
            float integrated_width = Math.Max(first_node.rect.w, second_node.rect.w);
            float integrated_height = first_node.rect.h + second_node.rect.h;
            integrated_piece = new Rectangle(0, 0, integrated_width, integrated_height);
        }

        else
        {
            throw new System.ArgumentException("Parameter can only be \"H\" or \"V\"",
"op (operator)");
        }


        // Create a rectangle node with the integrated rectangle
        Rectangle_Node integrated_rectangle_node = new Rectangle_Node(integrated_piece);

        // Set how has been made to set positions afterwards
        integrated_rectangle_node.integration_type = op;

        // Make the childs be the rectangles it has been made of
        integrated_rectangle_node.set_left_child(second_node);
        integrated_rectangle_node.set_right_child(first_node);

        node_Stack.Push(integrated_rectangle_node);

    }

    // sets the positions of the pieces doing a inorder traversal to the tree (once
built)
    public void calculate_positions(Rectangle_Node root)
    {


        // BASE CASE: if this node is a leaf, do not do anything
        if (root.left == null)
        {
            return;
        }



        // GENERAL CASE: if this node has childs, set its positions depending on this
node's integration type
        // Note: Since the positions are always relative to parent, they always start in
(0, 0) and it is the parent which changes their actual position
        // the traversal does not matter in this case (?)
        else
        {
            // Recursive calls first for a post-order traversal
            calculate_positions(root.left);
            calculate_positions(root.right);
            // Then, do the actions
            root.left.rect.x = 0;
            root.left.rect.y = 0;

            if (root.integration_type == "V")
            {
                // if it is V integrated, the second (right child) piece is put under
the first (left child) one
                root.right.rect.x = 0;
                root.right.rect.y = 0 - root.left.rect.h;
            }
            else if (root.integration_type == "H")
            {
                // if it is H integrated, the second piece is put to the right of the
first one
                root.right.rect.x = 0 + root.left.rect.w;
                root.right.rect.y = 0;
            }


        }

    }





    public static bool is_a_number(String str, out int n)
    {
        return int.TryParse(str, out n);
    }

    // Draws the pieces into the sheet given the layout representation as a tree

    // The first call should have the sheet_canvas as the parent_piece parameter
    public void draw_pieces(Rectangle_Node tree, GameObject parent_piece)
    {
        // The traversal type does not matter
        // Remember that the Rectangle_node contains a Rectangle that contains the id of
the piece
        // It is not suposed that we go back from this point, so we do not instantiate
the pieces -> We just use them directly



        int piece_id = tree.rect.get_id(); // We need to know what piece is the one we
are treating in this node of the tree

        // once we know what piece it is, we can retrieve it
        // Note: Depending of it is a pure piece or a integrated one, we will get the
piece from the list or create an empty game object to hold it, respectively

        GameObject piece;
        if (tree.left == null && tree.right == null)
        {
            // if it is a piece, get it
            piece = list_pieces[piece_id];
        }
        else {
            // if it is an integrated, create empty game object to hold the other
pieces/integrated rectangles
            piece = new GameObject();
        }

        // Set the child pieces' parent to be the actual node, so the positions are
always relative
        piece.transform.SetParent(parent_piece.transform);


        // Set the Local position depending of what the tree representation says
        piece.transform.localPosition = new Vector2(tree.rect.x, tree.rect.y);
        // Finally, if this node is a leaf, stop recursion
        if (tree.left == null && tree.right == null)
        {
            return;
        }

        else
        {
            // else, recursive call on both childs
            draw_pieces(tree.left, piece);
            draw_pieces(tree.right, piece);
        }
    }

    public void build_stack_from_chromosome(List<String> chromosome)
    {

        for(int i = 0; i < chromosome.Count; i++)
        {

            int n = -1;
            // if the gen of the chromosome is a part number, we need to retrieve that
number
            if (is_a_number(chromosome[i], out n))
            {

                // remember: the second parameter of stack is the piece id, starting at
0 (that is why we take out 1)
                // for example, if the first get of the chromosome is the number "3", we
want to stack the piece number 2
                stack(list_pieces, n-1);
            }
            else // not a number -> V or H
                unstack_operate_and_stack(chromosome[i]);
        }
    }

    // checks if a chromosome is valid
    // if it is valid, it returns 0
    // else, it returns an int depending on the error code

    // Error code 1: [ No = Np - 1 ] does not apply
    // Error code 2: There is a point in which [  1 ≤ No ≤ Np - 1  ] does not apply
    public int validate_chromosome(List<String> chromosome)
    {
        // Variables to check case 1
        int _No = 0;
        int _Np = 0;

        int n;
        // Check case 2 at any given point
        foreach (String gen in chromosome)
        {
            if (is_a_number(gen, out n)) _Np++; else _No++;
            if (!(_No <= _Np - 1)) return 2;

        }

        // Check case 1 at the end
        if (_No != _Np - 1) return 1;

        return 0;
    }



    // For now, there is only one tipe of fitness function. In the future there will
maybe be more
    // Sets the fitness of the chromosome given the tree that represents the layout
given by the chromosome



    // profited area
    public void calculate_and_set_fitness(Chromosome chr, Rectangle_Node tree,
List<GameObject> list_pieces)
    {

        double proportion_weight = Double.Parse(field_min_rect_factor.text);
        double square_factor_weight= Double.Parse(field_sq_factor.text);


        // get area of all pieces
        double area_of_pieces = 0;
        double w;
        double h;

        foreach (GameObject piece in list_pieces)
        {
            w = piece.GetComponent<RectTransform>().rect.width;
            h = piece.GetComponent<RectTransform>().rect.height;
            area_of_pieces += w * h;
        }


        // get area of whole rect
        double big_area_width = tree.rect.w;
        double big_area_height = tree.rect.h;
        double big_area = big_area_width * big_area_height;

        // get proportion
        double proportion = area_of_pieces / big_area;




        // calculate how squared is the integrated piece. This is, min value between W
and H, divided by the other one (the max)
        double square_factor = Math.Min(big_area_width, big_area_height) /
Math.Max(big_area_width, big_area_height);


        double fitness = (proportion * proportion_weight) + (square_factor *
square_factor_weight);



        chr.fitness = fitness;
        //fitness = fitness * Math.Pow(10, 6);


    }









    // Deprecated: Inverse of the area needed to store the whole piece set, multiplied
by how-squared factor

    public void calculate_and_set_fitness_old(Chromosome chr, Rectangle_Node tree)
    {
        if (chr.gen_list.Count < 2)
        {
            new System.InvalidOperationException("It is not possible to calculate the
fitness of an empty chromosome. Are you forgetting its initialization?");
        }

        float width = tree.rect.w;
        float height = tree.rect.h;
        float area_needed = width * height;
        double fitness = 1 / area_needed;

        // calculate how squared is the integrated piece. This is, min value between W
and H, divided by the other one (the max)
        float square_factor = Math.Min(width, height) / Math.Max(width, height);
        fitness = fitness * square_factor * 0.5;

        fitness = fitness* Math.Pow(10, 6);

        chr.fitness = fitness;


    }


}


public static class Utilities
{

    public static void unfold_relationships(Dictionary<string, List<string>> myDic)
    {
        foreach (KeyValuePair<string, List<string>> entry in myDic)
        {
            // entry.Value is the 2nd List
            List<string> not_processed = new List<string>();

            // copy list in entry.Value to not_processed
            foreach (string el in entry.Value)
            {
                not_processed.Add(el);
            }

            // Now, in not_processed we have the first list, we now need to iterate over
it until it is clear

            while(not_processed.Count != 0)
            {
                // Take and remove
                string element = not_processed[0];
                not_processed.RemoveAt(0);

                // add to not processed and the entry.Value the elements of the list
VALUE (peek dict value) if they are not in the 2nd List
                foreach (string second_el in myDic[element])
                {
                    if (!entry.Value.Contains(second_el) && second_el != entry.Key)
                    {
                        not_processed.Add(second_el);
                        entry.Value.Add(second_el);
                    }
                }

            }

        }


    }



    public static void print_dict(Dictionary<string, List<string>> myDic)
    {
        foreach (KeyValuePair<string, List<string>> entry in myDic)
        {
            Debug.Log("KEY: " + entry.Key);

            foreach (string result in entry.Value)
            {
                Debug.Log("     VALUE: " + result);
            }
        }
    }

    public static void print_list(List<string> myList)
    {
        string list = "";
        for (int i = 0; i < myList.Count; i++)
        {
            list += myList[i];
            list += " ";
        }
        Debug.Log(list);
    }

    public static string list_toString(List<string> myList)
    {
        string list = "";
        for (int i = 0; i < myList.Count; i++)
        {
            list += myList[i];
            list += " ";
        }
        return list;
    }

    public static void print_list_old(List<string> myList)
    {
        for (int i = 0; i < myList.Count; i++)
        {
            Debug.Log("ELEMENT Number " + i + ": " + myList[i]);
        }
    }

    public static bool is_a_number(String str, out int n)
    {
        return int.TryParse(str, out n);
    }

    public static bool is_a_number(String str)
    {
        int n;
        return int.TryParse(str, out n);
    }

    public static double get_random_double_between(double minValue, double maxValue,
System.Random rnd)
    {
         return rnd.NextDouble() * (maxValue - minValue) + minValue;
    }
}
```