Capítulo 7

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Palma, España
joan.vilella1@estudiant.uib.cat

La complejidad de muchos problemas informáticos ha hecho que gran parte de los algoritmos previamente estudiados sean inviables dada la duración de las ejecuciones. Los algoritmos probabilísticos surgen como una posible alternativa ya que utilizan un factor de aleatoriedad que permite que dichas ejecuciones sean considerablemente más rápidas. Con un buen control sobre el posible introducido y aplicado sobre problemas adecuados, se sitúan como una de las mejores opciones.

Keywords—MVC, rendimiento, algoritmos probabilísticos, Banderas.

I. INTRODUCCIÓN

Los algoritmos probabilísticos se han posicionado como una de las opciones más viables ante la solución de distintos problemas dado su funcionamiento. Son especialmente útiles cuando la complejidad de un problema tiene un coste asintótico muy elevado. Además, al ser "menos pesados" este tipo de algoritmos, permiten que puedan ser ejecutados en una mayor cantidad de máquinas, por lo que permite no depender de ordenadores especializados para la solución de problemas. Esto ha permitido que su uso en multitud de dispositivos móviles.

Ahora bien, estos algoritmos no son perfectos, ya que si no serían los únicos que se usarían. Los algoritmos probabilísticos fallan, lo cual es algo a priori extraño ya que no es un comportamiento que se haya considerado admisible hasta ahora. Pero si se logra controlar estos fallos, o al menos, acotarlos dentro de un percentil de error, se obtienen unos tiempos de ejecución considerablemente más cortos. Permitiendo así, múltiples ejecuciones hasta encontrar la solución.

Más adelante se ahondará en profundidad en los principios teóricos de este algoritmo, por el momento es importante destacar que se utilizará esta técnica nueva (desde el punto de vista del estudiante) para la identificación de banderas.

Para ello se han usado técnicas de colorimetría (ciencia que estudia el color) juntamente con los algoritmos probabilísticos. Con esto se busca clasificar cada uno de los píxeles de la imagen dentro de un rango de colores definidos. Una vez obtenidos estos datos, se comparará con la base de datos propia de las banderas.

Todo este proyecto se enmarca en el patrón de diseño MVC (Modelo Vista Controlador), permitiendo así aprovechar muchas de sus ventajas: mejor detección de errores, mayor escalabilidad del proyecto, código mejor estructurado, entre otros. Con el pequeño contra de que esto suponga una mayor complejidad a la hora de programar la solución.

II. CONTEXTO Y ENTORNO DE ESTUDIO

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre varios IDE's (*Integrated Development Environment*).

- NetBeans
- IntelliJ
- VS Code

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su versión de escritorio *Github Desktop* por su facilidad de uso mediante interfaz gráfica.

III. DESCRIPCIÓN DEL PROBLEMA

El problema a resolver es que, dada una bandera, el programa sea capaz de "adivinar" a que país pertenece consultando previamente una base de datos.

A la hora de entender el acercamiento probabilístico del problema, es importante entender que diferencias existen ante un acercamiento que se podría considerar más "tradicional".

El enfoque tradicional buscaría recorrer todos y cada uno de los píxeles, por lo que se tendría un conteo exacto. A posteriori se compararía con la base de datos, la cual se debería haber hecho ese proceso previamente, y atendiendo a ciertos umbrales de posible error, se seleccionaría la bandera correspondiente.

Este acercamiento sería imposible de llevar a la práctica o al menos, muy costoso computacionalmente. Dentro de esta aproximación se podrían buscar otras alternativas como por ejemplo escalar la imagen hasta cierto punto donde se pueda seguir haciendo la distinción. Pero como se demostrará más adelante, los algoritmos probabilísticos consultando un 1% de los píxeles, se pueden obtener resultados notablemente buenos. Difícilmente se puedan conseguir esos resultados escalando la imagen a un 1%.

1) Tipos de algoritmos probabilísticos

Aunque todos los algoritmos probabilísticos tienen una serie de características en común, dependiendo de su funcionamiento se pueden distinguir distintos tipos:

- Algoritmos de Montecarlo
- Algoritmos de Las Vegas
- Algoritmos de aproximación numérica
- Algoritmos de Sherwood

Como los últimos no han sido tratados en la asignatura, no se explicarán.

Algoritmos de Montecarlo: Estos algoritmos siempre dan un resultado, pero no siempre será el correcto. Se intenta minimizar la probabilidad de un resultado incorrecto, y utilizando el factor de aleatoriedad, con múltiples ejecuciones se reduce en gran nivel la probabilidad de error (selección modal). Uno de los inconvenientes es que no se puede saber si se han equivocado o no.

Algoritmos de Las Vegas: Una de las principales ventajas de este tipo de algoritmos es que nunca devuelven un resultado incorrecto, a cambio, habrá ejecuciones en las que no devolverá ningún tipo de resultado. Al igual que en el anterior tipo de algoritmo, se busca reducir la probabilidad de un "no resultado", con múltiples ejecuciones se reduce dicha probabilidad.

A diferencia del anterior, este algoritmo es capaz de discernir si se ha equivocado o no, por lo que puede repetir el cálculo hasta hallar una solución correcta. Dada a su alta velocidad, son muy convenientes.

Algoritmos numéricos: Los elementos aleatorios permiten aproximar resultados numéricos, en gran cantidad de ocasiones, más rápido que métodos tradicionales o "directos". Al igual que el resto, muchas ejecuciones permitirán una aproximación mayor.

El elemento aleatorio requerido en este tipo de algoritmos es un "generador de números aleatorios". Por motivos vistos, esta clase de generadores no son capaces de producir números propiamente aleatorios, sino que generan números pseudoaleatorios.

2) Colorimetría

Al estar trabajando con colores, se ha tenido que entrar en la teoría del color (Colorimetría). Aquí, en el aspecto informático, el espacio de color más utilizado es el RGB (Componente roja, verde y azul).
El problema del RGB es que es un poco complejo a la hora de clasificar colores, debido a que no tenemos concepto de saturación o brillo, solo niveles de estos tres colores.

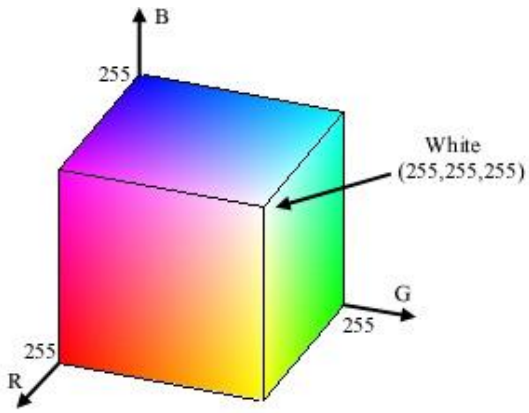

Ilustración 1: Espacio de color RGB

Este es el espacio de color RGB, como se puede observar, es un Cubo de lado 256. Para clasificar colores en este espacio, tendríamos que dividir de alguna manera el cubo en cubos más pequeños, para lo que necesitaríamos las coordenadas (valores R,G,B) de las dos esquinas de los sub-cubos. Muy complicado.

Debido a esto, hemos decidido usar el modelo de color HSB (o HSV). Este se base en tres componentes: *Hue* (Matiz), *Saturation* y *Brightness*. Esto permite clasificar los colores de una forma mucho más sencilla dependiendo de su matiz (que color realmente es), la saturación (el tono), y el brillo. Debido a esto, es muy fácil clasificar los colores, ya que ahora dependen principalmente del Hue (Matiz), y los blancos y negros comparten las características de la saturación y brillo que explicaremos más adelante.

IV. SOLUCIÓN PROPUESTA

En este apartado se comentará a modo de carácter general el planteamiento seguido en la solución implementada. Los detalles técnicos serán comentados en apartados posteriores, especialmente en la implementación del modelo.

Los dos métodos que más relacionados están con la propia resolución del problema son:

- *getColorPercentages*
- *findCountry*

Se podría decir de alguna manera que el resto son métodos auxiliares que permiten la ejecución de estos dos métodos. Por lo que serán los que se comenten en este apartado.

a) GetColorPercentages

Este es el método encargado de obtener los porcentajes que cada uno de los colores definidos que hay en una bandera. Una vez obtenidos estos datos, se podrá consultar con la base de datos y obtener el nombre de la bandera correspondiente.

Este método genera un número aleatorio por cada coordenada, en este caso serán dos (eje x e y). Estos números aleatorios al estar entre 0 y 1 se multiplican por el ancho y largo de la imagen (respectivamente). Con esto se logra elegir un píxel al azar de la imagen, ahora solo quedará obtener el color de dicho píxel e incrementar el contador de los colores definidos. Este cálculo se hace tantas veces como se indique por parámetro.

Finalmente se calculan los porcentajes y se devuelve en el array correspondiente.

b) FindCountry

Este método depende del anterior explicado, ya que recibe como parámetro la salida del método. Como su nombre indica, este método es el encargado de encontrar aquel país cuya bandera guarda mayor similitud con los porcentajes obtenidos.

Para encontrar dicho país, se itera por toda la base de datos (detalles de implementación comentados más adelante) y se calcula la distancia euclidiana que existe entre los porcentajes de colores con los de cada una de las banderas.

De entre estas distancias euclidianas, se devuelve el país que tenga la menor, es decir, la que guarde mayor grado de similitud con la base de datos.

Una vez conocidos los distintos algoritmos probabilísticos que existen y el propio funcionamiento de la solución implementada, se puede concluir que la solución propuesta es del tipo “algoritmos numéricos”. Esto se debe principalmente a que el factor de aleatoriedad introducido en el algoritmo proviene de un generador de números aleatorios, específicamente del de la clase *Random* de Java (Ilustración 1).

```
public double[] getColorPercentages(final int N_TESTS) {
    if (flag == null || database == null) {
        return null;
    }
    double x, y;
    int[] colorCount = new int[N_COLORS];
    for (int i = 0; i < N_TESTS; i++) {
        x = Math.random() * flag.getWidth();
        y = Math.random() * flag.getHeight();
        Color color = new Color(flag.getRGB((int) x, (int) y));
        int pos = FlagColor.getColor(color);
        colorCount[pos]++;
    }
    double[] percentages = new double[COLORS.length];
    for (int i = 0; i < colorCount.length; i++) {
        percentages[i] = (double) colorCount[i] / N_TESTS;
    }
    return percentages;
}
```

Ilustración 2: Uso de generadores aleatorios

V. PATRÓN MVC

El Modelo-Vista-Controlador (MVC) [1] es un patrón de diseño de software[1] en el que se divide la lógica del programa de su representación gráfica, además se hace uso de un controlador para los eventos y comunicaciones entre las distintas partes.

Este patrón de arquitectura se basa en las ideas de reutilización de código y la separación de conceptos

[1] Un patrón de diseño es un conjunto de técnicas utilizadas para resolver problemas comunes en el desarrollo de software.

distintos, estas características pretenden facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. Para ello se proponen la construcción de tres componentes:

- Modelo:
  Es donde se almacena la información necesaria para la ejecución de la aplicación. Gestiona el acceso a dicha información mediante peticiones a través del controlador. También contiene los procedimientos lógicos que hagan uso de esa información.

- Vista:
  Contiene el código que muestra la aplicación, es decir, que va a producir la visualización tanto de la interfaz del usuario como de los resultados

- Controlador:
  Responde a eventos usualmente generados por el usuario y los convierte en comandos para la vista o modelo. Se podría decir que hace la función de intermediario entre la vista y el modelo encargándose de la lógica del programa.

Como se ha mencionado anteriormente se ha decidido utilizar este patrón debido a la facilidad que aporta al programar la separación de conceptos. Además, se obtiene una capa más de abstracción ya que es posible utilizar diferentes vistas con un mismo modelo.

Las ventajas principales del MVC son: Escalabilidad, facilidad de tratamiento de errores y reutilización de componentes. Existen otras ventajas, pero esta arquitectura se aprovecha en mayor medida en aplicaciones web, el cual no es este caso.

La principal desventaja que existe del patrón MVC es la complejidad que añade a la programación. Ya que, para el mismo problema, hay que modificar el acercamiento para que quepa dentro de esta arquitectura. Lo que implica una mayor complejidad.

Hoy en día, es muy común el uso de la programación orientada a objetos (POO) como paradigma principal, por ello cada componente del patrón MVC suele implementarse como una clase independiente. A continuación, se explica que proceso se ha seguido y como se ha implementado cada parte.

A. Implementación del Modelo

Para la implementación del modelo en primer lugar hemos definido una clase auxiliar *FlagColor* que es un enum que contiene una lista de los colores definidos para las banderas, cada color tiene un valor mínimo y máximo que utilizaremos para compararlos posteriormente. Esta clase tiene un método estático llamado *getColor()* que pasado un Color de java nos devuelve el índice del *FlagColor* con el color más cercano.

El cálculo de la distancia se hace en modelo de Color HSB ya que con el Hue se puede identificar más fácilmente el tono del color. Como se ha comentado antes cada *FlagColor* tiene un min y un max, si el hue está dentro de ese rango lo clasificamos como ese color. Para detectar los negros comprobamos si el brightness es menor a un threshold y de forma similar para el blanco, pero añadiendo la saturación.
Por lo tanto, tenemos en total 14 colores distintos.

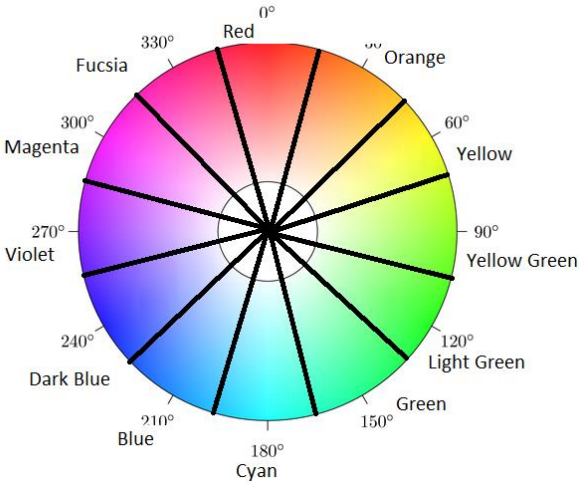

Ilustración 3: Colores según el Hue

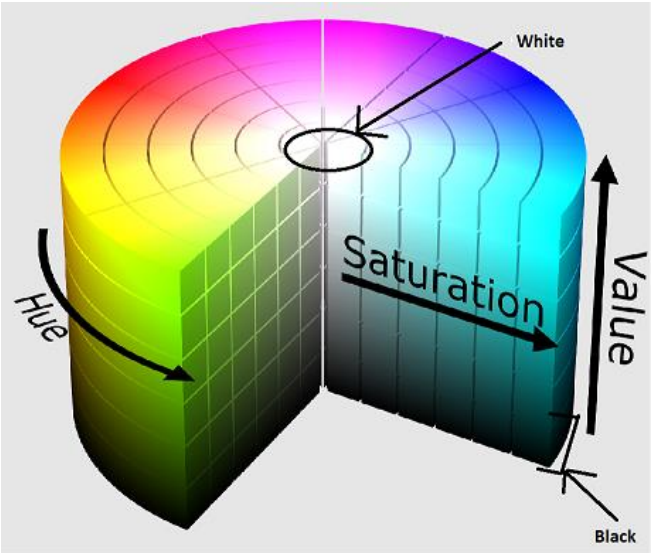

Ilustración 4: Blanco y Negro en HSB

Además de esta clase, se ha definido la clase principal *Model* que contiene todos los métodos necesarios para el algoritmo.

Esta clase tiene definidas una serie de variables para almacenar la bandera del país, el nombre del país y un *Hashmap* que será la "Base de Datos" y es indexado por el nombre del país y contiene un array de doubles que representa el porcentaje de cada color en la bandera de dicho país. Para crear esta base de datos tenemos el método loadDatabase() que pasado el directorio de las banderas, comprueba si la base de datos ya existe. En caso contrario se crea el hashmap y una entrada por cada bandera del directorio utilizando el método *getRealPercentages()* que devuelve los porcentajes de color recorriendo todos los pixeles de la bandera, finalmente se escribe el hashmap en un archivo para poder ser reutilizado en posteriores ejecuciones. Cabe destacar que utilizamos las funcionalidades de Java Locale para convertir los nombres de código ISO al nombre del país. Por ello los ficheros de las banderas deben tener de nombre el código ISO.

Además de la carga de la BD, hay definidos algunos métodos auxiliares que se explican a continuación:

- *loadFlag():* Este método se encarga de settear en la variable *flag* la imagen del archivo pasado como parámetro.
- *getRandomFlag():* Este método devuelve un archivo de bandera aleatorio dado el directorio de las banderas.
- *getFlagImage():* Este método devuelve la bandera actual, o la bandera del país indicado por parámetro.
- *distanceBetweenColorPercentatges():* Este método devuelve la distancia euclidiana entre dos vectores de porcentajes de colores, esta distancia se calcula realizando el módulo de la diferencia.
- *getRealPercentatges():* Este método es parecido al *getPercentatges()* pero en lugar de calcular los porcentajes de color de N pixeles, realiza un recorrido de toda la imagen, se utiliza para crear la BD.

Por último, cabe destacar que el modelo utiliza el *propertyChange* para notificar de cambios al controlador y de esta forma poder actualizar la información en la vista. Esto se usa principalmente durante la creación de la base de datos para mostrar un dialogo y actualizar la respectiva barra de progreso.

B. Implementación de la Vista

Para la implementación de la vista se ha creado la clase *View*.

La clase *View* es la ventana principal del programa que contiene tanto los componentes de la interfaz de usuario como toda la lógica que hay detrás de estos componentes.

La ventana principal contiene los siguientes elementos:

- **JButtons**:
  - "Random Flag": pone en la vista una bandera aleatoria con su nombre, para luego poder adivinarla.
  - "Guess Country": hace las correspondientes llamadas al modelo para adivinar la bandera, actualiza los porcentajes de colores, y pone el país de la solución propuesta con su bandera.
- **JPanel**: dos JPanel para mostrar la bandera actual y la bandera de la solución propuesta
- **JLabel**: Dos JLabel para los nombres del país actual y el propuesto, y luego unos JLabels para los colores.
- **JProgressBar**: una para cada color, para mostrar el porcentaje de ese color en esa bandera.

Hay definidos varios métodos, pero solo explicaremos con mayor detalle los más interesantes y que aportan valor a la documentación:

1) setFlagImage

Este método se encarga de poner la imagen y nombre del país pasados por parámetro en el JPanel de la vista que contendrá la bandera a adivinar.

2) setGuessImage

Este método se encarga de poner la imagen y nombre del país propuesto, así como actualizar las barras de porcentajes de los colores. Si la solución propuesta es correcta, el nombre del país se pondrá en verde. En caso contrario, se pondrá en rojo.

Este devuelve un entero, 1 si es la solución correcta y 0 si es incorrecto. Este valor no se usa para nada en el programa como tal, solo se usa para hacer las ejecuciones del estudio de rendimiento. En los estudios de rendimiento, sumamos el valor que nos devuelva esta función para hacer el cálculo del "Score". Así que, si ha

acertado, sumaremos 1 en el denominador, y si no, sumaremos 0.

C. *Implementación del Controlador*

Para la implementación se ha creado la clase *Controller* que contiene todos los métodos necesarios para poder comunicar ambas partes del patrón MVC.

En primer lugar, se han definido dos atributos de la clase que contienen las instancias del modelo y la vista, estos se reciben directamente como parámetros a través del constructor. A continuación se ha definido el método start() que es al que debe llamar la aplicación principal para comenzar la ejecución de la aplicación.
Este método únicamente añade los listeners correspondientes al modelo y a la vista, posteriormente hace la vista visible al usuario.

Finalmente hay definidos dos métodos que son los que se han proporcionado como listeners. Estos métodos se ejecutarán cuando el modelo o la vista lancen el evento pertinente. A continuación, se explican en mayor detalle:

- viewActionPerformed():
  Dependiendo del evento que se reciba, se hará lo siguiente:
  - "Random Flag": se genera una bandera aleatoria y se carga en el modelo (para los cálculos posteriores a la hora de adivinar). Tambíen se carga en la vista.
  - "Guess Country": en un nuevo hilo, cargamos la base de datos de las banderas (si no se ha hecho ya antes), se calculan los porcentajes de la bandera con el número de pixeles de muestreo, se encuentra la bandera, y se actualiza la solución propuesta en la vista.

1) *Programación Concurrente*

La programación concurrente es una forma de cómputo en la que el trabajo se divide en varios hilos de ejecución distintos que trabajan simultáneamente. Esto suele mejorar el rendimiento de una aplicación al poder realizar cálculos largos en un hilo aparte. Para este proyecto se ha decidido utilizar este concepto sobre el modelo para realizar los cálculos en segundo plano.

Como se ha comentado anteriormente en el método viewActionPerformed() se crea un nuevo hilo donde se ejecutaran los cálculos. Para ello se hace uso de la interfaz Runnable que nos permite crear un método run() que será el que ejecute el nuevo hilo. Dentro del hilo se ejecuta el algoritmo probabilístico, se obtienen los resultados y se pasan a la vista para que puedan ser visualizados.

VI. Estudio de Rendimiento

En este apartado se mostrarán los resultados de unas pruebas de rendimiento que se han realizado.

La variable principal en esta práctica era el número de muestras (píxeles) que se tomaban de las imágenes. Así que se ha probado de realizar ejecuciones con distintos números de muestras, para ver qué porcentaje de acierto tenían.

El número de píxeles (N_PIXELS) ha variado desde 10 hasta 50.000. Tenemos que recordar, que cuántos más píxeles se muestrean, más tarda el algoritmo, pero también más certero es.
El objetivo de esto era ver el gráfico resultante de esta correlación para poder decidir, dependiendo del % de acierto buscado, el mejor (mínimo) número de muestras.

Una vez realizados los tests, hemos trasladado los datos a una hoja de cálculo y hemos obtenido el siguiente resultado:

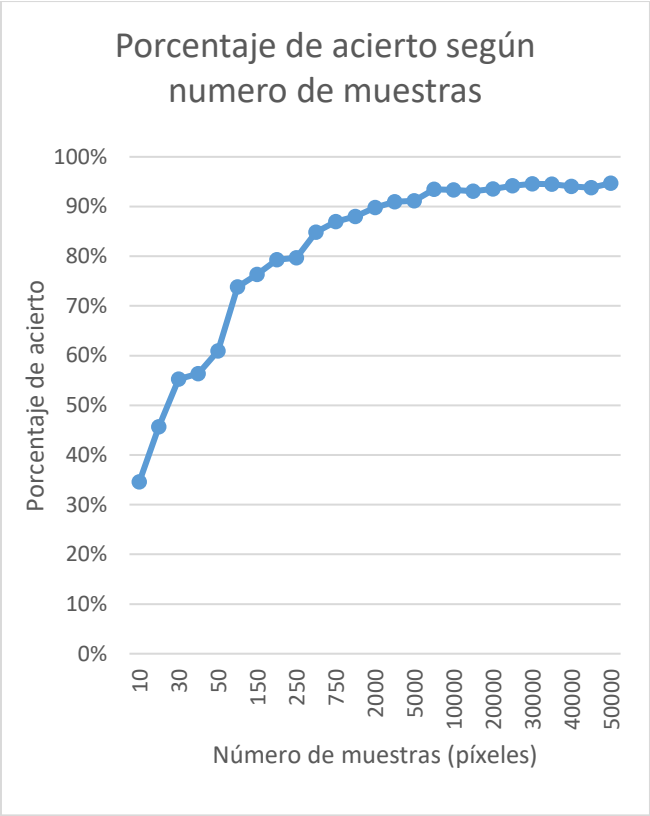

Gráfico 1

Como se puede observar, sobrepasamos el 90% de acierto en las 2000 muestras. Teniendo en cuenta que las imágenes tienen entre 150.000 y 200.000 píxeles, eso es apenas un 1% de la cantidad total de información de la imagen.

Cabe destacar que, con el algoritmo implementado, es imposible llegar al 100% de acierto, incluso muestreando todos los píxeles, debido a que hay un par de países que tienen las banderas casi iguales, y los porcentajes de colores exactamente iguales. Estos podrían ser: Indonesia, Mónaco y Polonia, por una parte, y Suiza y Dinamarca por otra.

Un posible estudio que se podría realizar es mirar el cambio en el rendimiento dependiendo de la cantidad de Colores disponibles al clasificar. Es decir, nosotros ahora estamos clasificando los píxeles en uno de 14 colores (blanco, negro y los 12 explicados más arriba).

Se podría mirar de hacer que la cantidad de colores fuese dinámica, e ir probando qué número de colores da mejores resultados sin llegar a los 360º del Hue de HSV.

VII. Posibles mejoras

Como se ha explicado en el párrafo anterior, hay algunas banderas que no son diferenciables con el algoritmo explicado. Una posible mejora que se le podría añadir al algoritmo sería tener información sobre los porcentajes de colores de según que zonas más pequeñas y concretas de las imágenes. Esto podría arreglar el problema de las banderas con mismas densidades de colores, ya que normalmente los colores no están dispuestos de la misma manera sobre la imagen. Esto, en cambio, haría que el algoritmo fuese un poco más lento, ya que tendríamos que hacer dos búsquedas, una para la bandera entera y otra para el trocito más pequeño, y también haría que la base de datos fuese más grande, ya que debería contener también las distribuciones de los colores de esa parte de la imagen.

VIII. Guía de usuario

A continuación, se explicará como ejecutar el programa mostrando el flujo del programa de cara al usuario y todas las funcionalidades de las que dispone.

1. Inicio del programa:

Nos aparece la siguiente ventana con los siguientes elementos:

- "Random Flag": simplemente elige una bandera al azar y la pone en el recuadro grande, juntamente con el nombre del país.
- "Guess Country": Ejecutará el algoritmo implementado.

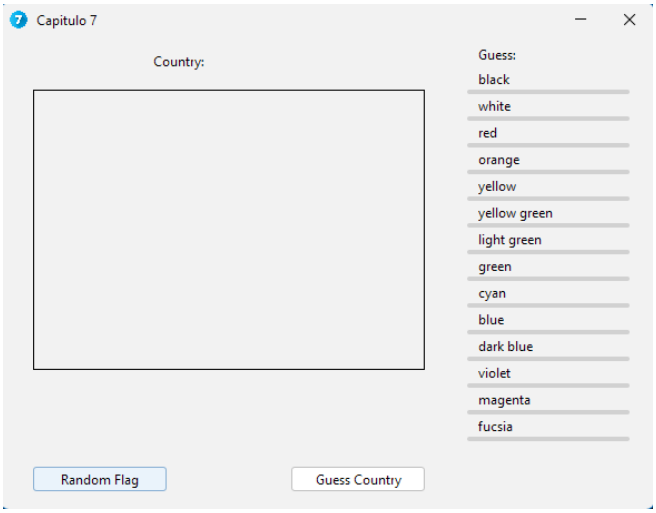

Ilustración 5

2. Inicio del algoritmo:

Al mostrar un país, se puede ejecutar el algoritmo. Dándole al botón "Guess Country", se iniciará el algoritmo.

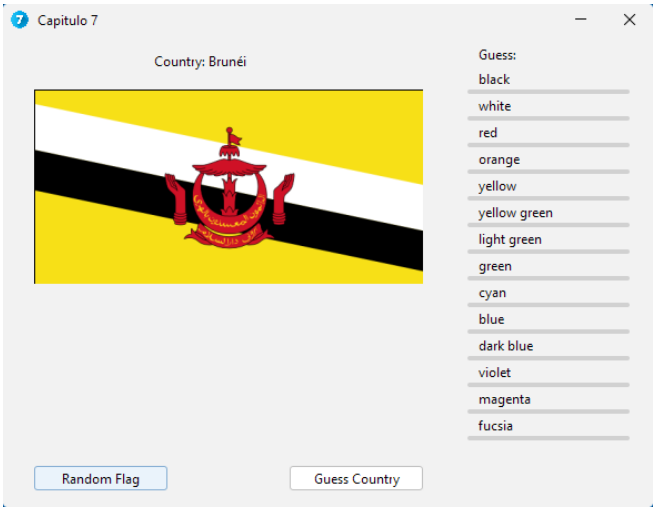

Ilustración 6

3. Carga de la base de datos:

Si es la primera vez que se hace en el dispositivo, puede que el programa tenga que generar la base de datos con los valores de las banderas de cada país. En ese caso, aparecerá una ventana de carga mostrando el progreso.

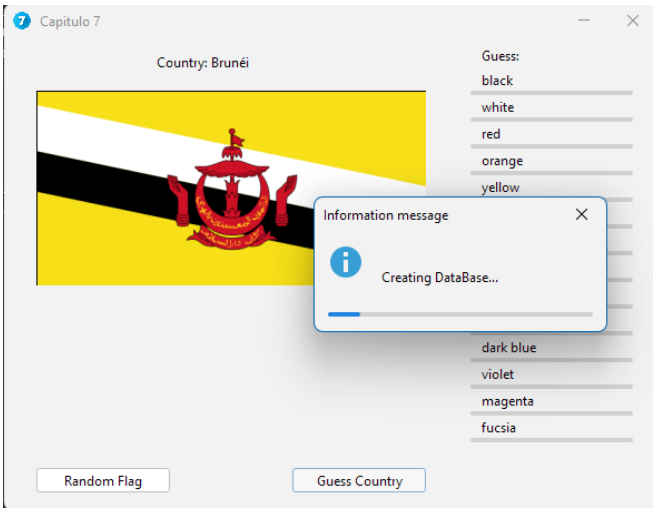

Ilustración 7

4. Muestra de resultados:

Cuando se haya generado la base de datos, el algoritmo se ejecutará, y cuando haya terminado, mostrará el resultado de la siguiente manera:

- Respuesta correcta: El nombre saldrá en verde.

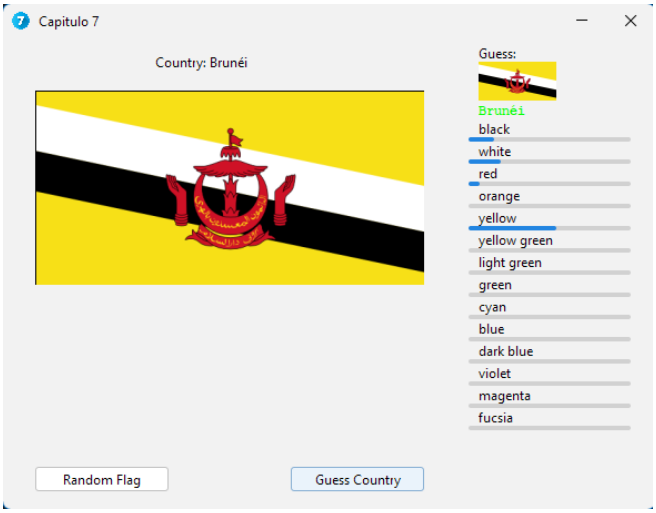

Ilustración 8

- Respuesta incorrecta: El nombre saldrá en rojo.

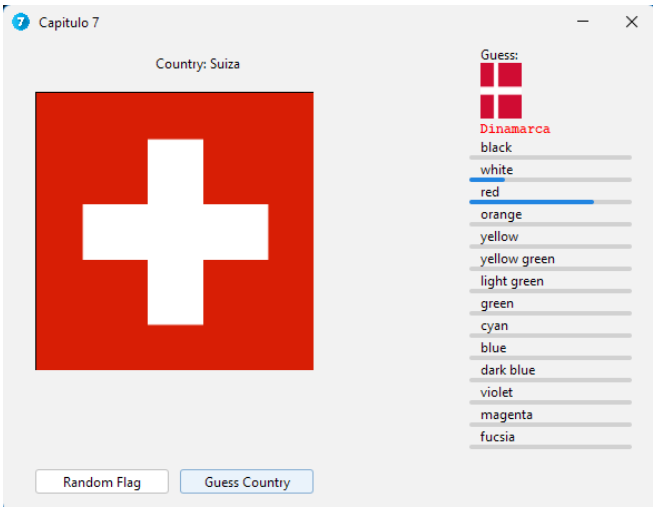

Ilustración 9

También podemos ver unas barras con los porcentajes de cada color de las muestras obtenidas (es decir, del algoritmo probabilístico).

5. Nueva ejecución:

Se puede volver a clicar en “Guess Country”, y se ejecutará el algoritmo otra vez sobre la imagen, lo que puede dar un resultado distinto al estar cogiendo muestras aleatorias.

Si se desea cambiar de imagen, simplemente le damos a “Random Flag” y se pondrá otra bandera, sobre la que podremos volver ejecutar el algoritmo.

IX. CONCLUSIONES

Esta práctica nos ha servido para entender lo útiles que son los algoritmos probabilísticos.
Teniendo una buena base de conocimientos sobre un problema, reducirlo a muestras sobre las que poder aplicar unos cálculos probabilísticos, puede reducir el coste del problema significativamente sin empeorar ni alterar los resultados del programa.

Como hemos podido ver en el estudio de rendimiento, con apenas un 1% de los píxeles de la imagen podemos acertar 9 de cada 10 banderas, ahorrándonos un 99% del procesamiento.

Además, al ser el último proyecto de la asignatura, nos gustaría recalcar que hemos aprendido mucho gracias a todas las prácticas. Nos ha gustado mucho la metodología de no tener examen final y no centrarse en aprender la teoría si no más en poner en práctica los conceptos y entender el proceso de diseño de algoritmos.

También el poder practicar un rango de algoritmos tan amplio y distintos nos permite practicar mucho y aprender a desarrollar algoritmos mejores y más robustos, así como poder elegir una metodología adecuada para el problema en cuestión.

Por último, solo decir, que la documentación de cada capítulo a veces se nos ha hecho un poco larga, y creemos que hemos tenido que sacar más contenido del necesario o tal vez repetirnos un poco para poder llegar al mínimo, y que tal vez al complementar con el video podría bastar con menos páginas (o, en lugar de forzar el número de páginas, probar con el número de palabras) y estas quedarían más concisas y bien explicadas. Por lo demás ha sido un placer de asignatura y hemos aprendido mucho y repasados conceptos de años anteriores.

X. BIBLIOGRAFÍA

Nilsson, S. (s. f.). *Las Vegas vs. Monte Carlo algorithms*. Algorithms to Go. https://yourbasic.org/algorithms/las-vegas/

Probabilistic (Randomized) algorithms. (s. f.). http://www.cs.man.ac.uk/~david/courses/advalgorithms/probabilistic.pdf

What Is a Monte Carlo Simulation? (2021, 4 octubre). Investopedia. https://www.investopedia.com/terms/m/montecarlosimulation.asp

Brownlee, J. (2019, 25 septiembre). *A Gentle Introduction to Monte Carlo Sampling for Probability*. Machine Learning Mastery. https://machinelearningmastery.com/monte-carlo-sampling-for-probability/

Wikipedia contributors. (2022, 21 junio). *Monte Carlo method*. Wikipedia. https://en.wikipedia.org/wiki/Monte_Carlo_method

Colaboradores de Wikipedia. (2022, 4 mayo). Modelo de color HSV. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/Modelo_de_color_HSV

colaboradores de Wikipedia. (2022b, junio 1). RGB. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/RGB

Wikipedia contributors. (2022, 19 abril). Randomized algorithm. Wikipedia. https://en.wikipedia.org/wiki/Randomized_algorithm

XI. AUTORES

Jonathan Salisbury Vega, nació en Costa Rica en el 2000. A los 17 años se graduó de bachillerato en el instituto de educación secundaria IES Pau Casesnoves. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares.

Joan Sansó Pericás, nació en Manacor en 2001. A los 18 años se graduó de bachillerato en el instituto de educación secundaria IES Manacor. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares.

Joan Vilella Candia, nació en Elche en el 2000. A los 18 años se graduó de bachillerato en el instituto de educación secundaria IES Inca. Actualmente está cursando el Grado de Ingeniería Informática en la Escuela Politécnica Superior de la Universidad de las Islas Baleares.