Emanuel Estrada Larios
Prof. Jairo Salazar
Estructura de Datos
9 de octubre del 2017

Ruleta 4: Linear Probe Hashing

Descripción de trabajo

Para el desarrollo de este programa, se utilizó el lenguaje Java para crear un programa que utilizara la estructura de linear probe hashing en combinación con análisis y comparación de imágenes.

Para comenzar, se realizo una extensa investigación del método para entender mejor el funcionamiento y la implementación de la estructura y de nodos, utilizándose como referencia los libros de Cormen y Sedgewick[1], los cuales describen la arquitectura de una tabla hash y como lidiar con colisiones con linear probing en detalle e incluye ejemplos de código. En cuanto al algoritmo para analizar las imágenes, se utilizó el articulo del Dr. Neal Krawetz en The Hacker Factor Blog como principal fuente de información[4].

Más adelante, se comenzó con el desarrollo del programa como tal, comenzando primero con el mejoramiento de la estructura con la que ya se contaba por la ruleta anterior, el diseño del algoritmo y una vez terminado este, la implementación de una interfase gráfica. Se finalizó con una etapa de pruebas y debugging, así como con la realización de este reporte.

Evidencia de funcionamiento

Durante la etapa de pruebas, se probó brevemente el funcionamiento de la estructura una vez más, ya que esta funcionaba al ser la misma de la ruleta anterior.

Para este caso, las pruebas se enfocaron principalmente en probar el funcionamiento del algoritmo de análisis de imágenes y como podía implementarse este con un hash lineal. Se realizan pruebas tempranas en la terminal, en las que se observaban bit por bit de una versión reducida de la imagen a 8x8 pixeles y escala de grises, para tener 64 bits

de color. Con estos datos, se buscó un patron que pudiese ser útil para su implementación con el linear probe hash.

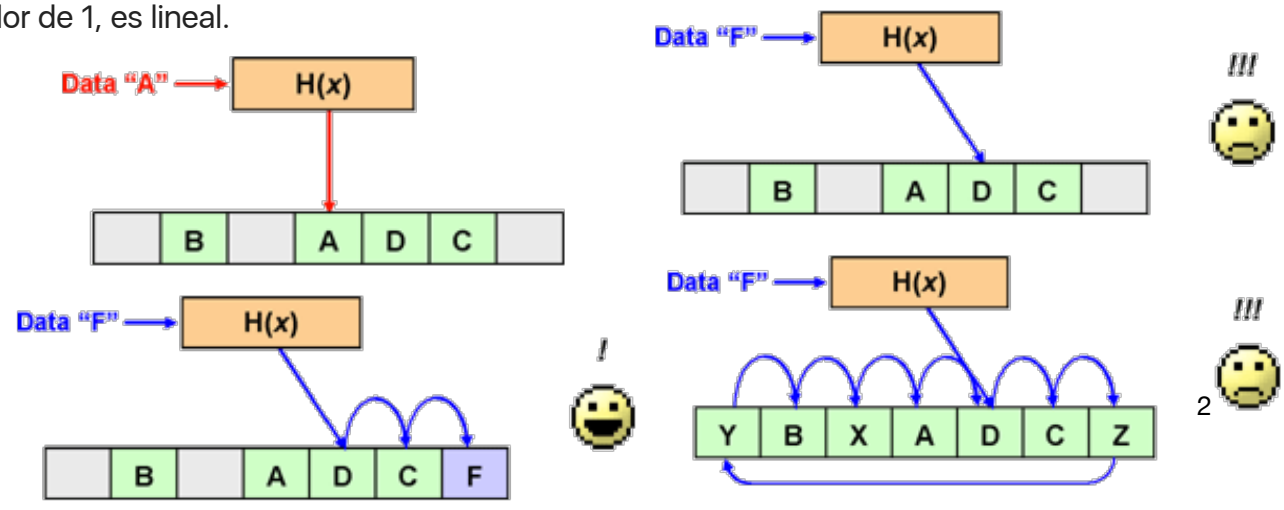
Más adelante, se probó el funcionamiento con la interface gráfica y con un tamaño de 100 para la tabla de hash. En la ruleta anterior se comprobó que el tamaño del hash puede ser mucho mayor y funcionar correctamente, pero se decidió utilizar este tamaño por dos razones: simplificar el resultado de las llaves para facilitar su interpretación y optimización de memoria.

Si desea comprobar el funcionamiento del programa, puede obtener y compilar el código completo en GitHub[2].

Marco teórico

Las tablas de hash son una estructura que ayuda a realizar operadores de diccionario de forma rápida y eficiente, porque utilizando una función se determina un índice único para guardar cada llave en un arreglo en la mayoría de los casos, y en caso de que se produzca un índice repetido, se puede resolver con diversos métodos.

Una tabla hash es en realidad un arreglo de llaves que se ordena mediante una función (comúnmente conocida como función hash), que utilizando la llave de cada elemento y una cierta operación, determina un índice para dicho elemento, y cuando un índice se produce más de una vez, lo cual se conoce como colisión, se puede resolver de varias maneras, en este caso se utiliza una técnica de direccionamiento abierto, llamado linear probing. Esta técnica define su función hash de modo que el índice sea el resultado de la llave modulo tamaño del arreglo, y en caso de que el índice ya esté lleno, se va incrementando uno al valor de la llave, y se repite la operación cuantas veces sea necesaria. A esto se le llama número de probes, y por el hecho de que se le incrementa un valor de 1, es lineal.



Los diagramas anteriores ilustran el funcionamiento de la estructura. En la primera, se busca agregar el elemento A, por lo que se llama a hash que regresa un índice, y se agrega. En el segundo, se busca agregar el elemento B, pero al llamar hash, este regresa un índice que ya está ocupado por D, por lo que se le suma 1 y se recalca el hash hasta que regrese un índice desocupado. Si todos los índices ya están ocupados, entonces la función de agregar falla y regresa un error[3].

En cuanto al algoritmo de análisis de imágenes, este es muy sencillo, y en este programa, se utilizó simplemente como un generador de llaves que son después ingresadas en la función hash. Primero, se toma la imagen y se reduce a un tamaño de 8x8 y se convierte a escala de grises. Esto produce una imagen de exactamente 64 bits con únicamente 64 colores. El valor de estos 64 colores se promedia y apartar de este promedio, se cuentan cuántos bits se encuentran sobre el promedio y cuántos debajo del promedio. Si se comparan estos dos números con los de otra imagen, se puede observar que si su diferencia es 0, las dos imágenes son la misma definitivamente; si su diferencia es de 1-2, la imagen puede ser la misma que la otra pero con alguna breve modificación; si su diferencia es de 3-5 las imágenes son diferentes pero visualmente parecidas; y si la diferencia es de más de 5 entonces son definitivamente imágenes diferentes. Utilizando estos dos números, se concatenan y generan una llave de cuatro dígitos, que al entrar en la función de linear hash, el resultado también presenta la característica de las diferencias, las imágenes similares quedarán en índices similares. Este algoritmo no solo es simple y rápido, sino que funciona el 90% de las veces.

Código

```
1. class Key<T>{
2.   public int k;
3.   public T data;
4.
5.   public Key(int k, T data){
6.     this.k = k;
7.     this.data = data;
8.   }
9. }
```

Comentarios del código por líneas:

1-9: Clase genérica Key.

2-3: Declaración de los atributos k (identificador de llave) y data (dato satelital).

5-8: Constructor de la clase, donde se recibe como parámetro el identificador de llave y el dato y se guardan en sus respectivos atributos.

```
1. class LinearProbeHash {
2.
3.   private int m, i, numberOfKeys;
4.   private Key[] hashTable;
5.   private Key deleted;
6.
7.   public LinearProbeHash(int tableSize){
8.     this.m = tableSize;
9.     this.hashTable = new Key[this.m];
10.    this.deleted = new Key(0,"");
11.    insertElement(deleted);
12.  }
13.
14.  private int hash(Key key){
15.    return (auxHash(key) + this.i) % this.m;
16.  }
17.  private int auxHash(Key key){
18.    return key.k;
19.  }
20.
21.  public int insertElement(Key key){
22.    this.i = 0;
23.    int j = 0;
24.
25.    while(this.i < this.m){
26.      j = hash(key);
27.      if(hashTable[j] == null || hashTable[j] == this.deleted){
28.        hashTable[j] = key;
29.        return j;
30.      }
31.      else{
32.        this.i++;
33.      }
34.    }
35.    return 0;
36.  }
```

```
37. public int[] searchElement(Key key){
38.    this.i = 0;
39.    int j = 0;
40.
41.    while(this.i < this.m){
42.      j = hash(key);
43.      if(hashTable[j] != null && hashTable[j] != this.deleted){
44.        if (hashTable[j].k == key.k){
45.          int[] fullMatch = {j};
46.          return fullMatch;
47.        }
48.        else
49.          this.i++;
50.        }
51.      else
52.        break;
53.    }
54.      j = hash(key);
55.      int[] similarImg = new int[5];
56.
57.      if(hashTable[j] != null && hashTable[j] != this.deleted)
58.        similarImg[0] = j;
59.      if(hashTable[(j-1) % this.m] != null)
60.        similarImg[1] = (j-1) % this.m;
61.      if(hashTable[(j+1) % this.m] != null)
62.        similarImg[2] = (j+1) % this.m;
63.      if(hashTable[(j-2) % this.m] != null)
64.        similarImg[3] = (j-2) % this.m;
65.      if(hashTable[(j+2) % this.m] != null)
66.        similarImg[4] = (j+2) % this.m;
67.
68.      return similarImg;
69.  }
70.
71.  public void deleteElement(Key key){
72.    int j = searchElement(key);
73.    hashTable[j] = this.deleted;
74.  }
75.
76.}
```

Comentarios del código por líneas:

1-76: Fragmento de la clase LinearProbeHash.

3-5: Declaración de atributos, entre ellos el arreglo de llaves y la llave "deleted".

7-12: Constructor de la clase, donde también se agrega la llave "deleted" al índice cero

que servirá como índice regresado al producirse un error.
14-19: Funciones hash y hash auxiliar, que calculan el índice del elemento utilizando el valor de su llave y los principios de linear probing.
21-36: Método insertElement, que recibe como parámetro la llave a almacenar y regresa el índice donde se almacene el dato.
25-34: While loop, que corre mientras el número de probes sea menor al tamaño de la tabla y mientras no se alcance un índice nulo o eliminado.
26-32: Se calcula el índice de la llave llamando a hash( ), y si este índice está disponible, se guarda la llave en el mismo, de lo contrario, se incrementa el número de probes para re-calcular el índice.
38-69: Método searchElement, que recibe como parámetro la llave a buscar y regresa un arreglo con el índice del elemento exacto o con los elementos más similares en caso de no encontrar el elemento exacto.
41-53: While loop, que corre mientras el número de probes sea menor al tamaño de la tabla y mientras no se alcance un índice nulo o eliminado.
43-49: Se calcula el índice de la llave llamando a hash( ), y si el identificador de llave almacenado en este índice coincide con el del parámetro, entonces regresa el número de índice, de lo contrario, se incrementa el número de probes para re-calcular el índice.
71-74: Método deleteElement, que recibe como parámetro una llave, calcula su índice y le asigna la llave dummy "deleted".

*Se omitieron algunos métodos que son irrelevantes a la funcionalidad principal de la estructura, como el método printTable, entre otros.

```
1.   private int imgKeyGen (File img) {
2.     BufferedImage originalImg = null;
3.     int pxAvobe = 0, pxBelow = 0;
4.
5.     try {
6.       originalImg = ImageIO.read(img);
7.       Image thumbnail = originalImg.getScaledInstance(8, 8,
         Image.SCALE_SMOOTH);
8.
9.       BufferedImage bufferedThumbnail = new
         BufferedImage(thumbnail.getWidth(null), thumbnail.getHeight(null),
         BufferedImage.TYPE_BYTE_GRAY);
10.
11.      bufferedThumbnail.getGraphics().drawImage(thumbnail, 0, 0, null);
```

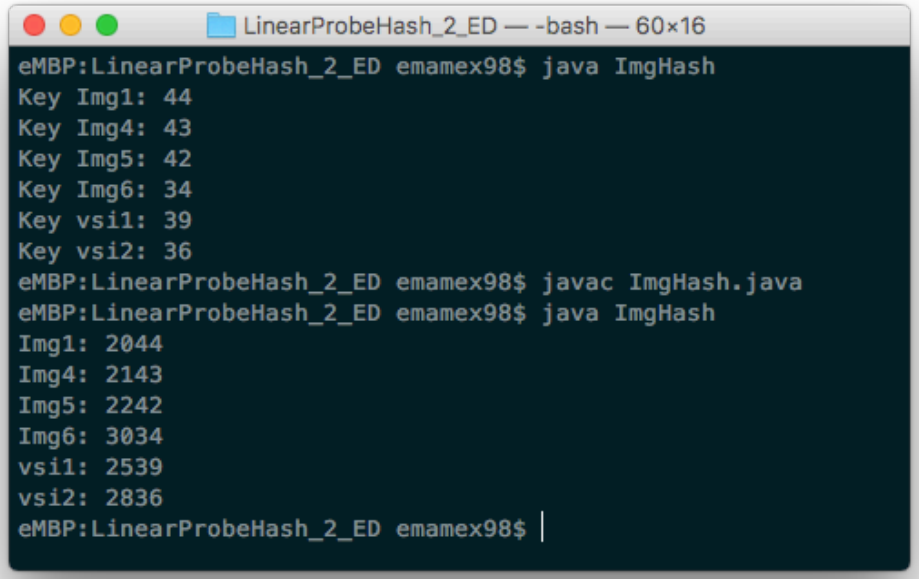
```
12.     int[] pixels = bufferedThumbnail.getRGB(0, 0, 8, 8, null, 0, 8);
13.     int colourAverage = 0;
14.
15.      for (int i=0; i<pixels.length; i++) {
16.        colourAverage += pixels[i];
17.      }
18.
19.      colourAverage = colourAverage / 64;
20.
21.      for (int i=0; i<pixels.length; i++) {
22.        if(pixels[i] <= colourAverage)
23.          pxBelow++;
24.        else
25.          pxAvobe++;
26.      }
27.
28.    }
29.
30.    catch (IOException e) {  System.out.println("Error"); }
31.    return Integer.parseInt("" + pxBelow + pxAvobe);
32. }
```

Comentarios del código por líneas:

1-32: Fragmento de clase ImageFinder; método imgKeyGen.
2: Declaración de variable tipo BufferedImage en donde se almacena la imagen a evaluar.
5-28: Try en donde se procesa y analiza la imagen para generar la llave.
7: El tamaño de la imagen se reduce a 8x8, sin mantener la escala de aspecto.
9: La imagen reducida es ahora convertida a escala de grises para reducir los colores a 64.
12: En un arreglo se guarda el valor del color de cada pixel.
15-19: Se calcula el valor promedio del color de cada pixel.
21-26: For loop recorre el arreglo y cuenta cuantos pixeles están sobre y debajo del promedio.
30: En caso de que la lectura del archivo regrese una excepción, el catch la maneja.
31: La función regresa la concatenación de los números de pixeles sobre y debajo del promedio, la cual será utilizada como llave.

Casos de prueba

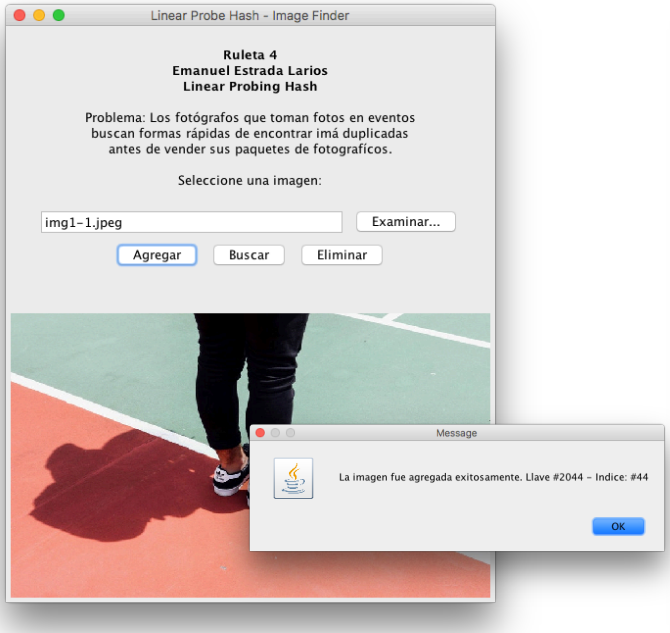
Como se mencionó anteriormente, se realizaron diversas pruebas para demostrar el funcionamiento del programa. En las primeras etapas del proyecto, se realizaron pruebas en la terminal en dónde se analizaban los índices y llaves de imágenes similares para

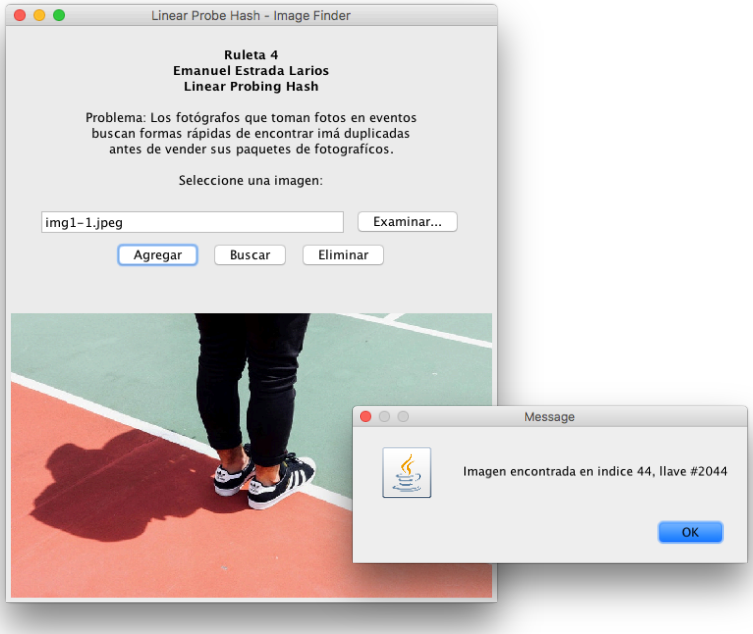


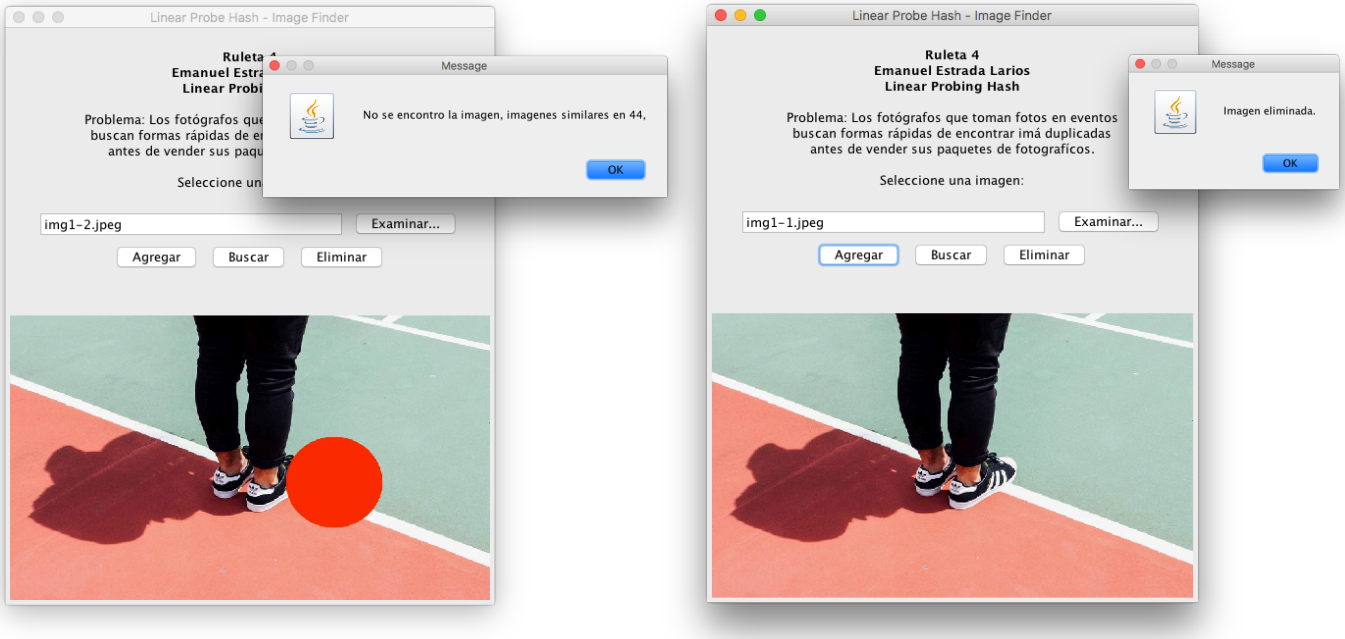
establecer una regla de similitud. Como se puede observar en la imagen anterior, los índices de imágenes similares son muy cercanos.

En las capturas de pantalla posteriores, se puede observar pruebas realizadas en la GUI. En la primera, se carga una imagen de un directorio y se agrega a la hash table. La imagen genera la llave 2044 y por lo tanto se almacena en el índice 44.

En la segunda captura de pantalla, se busca la misma imagen en el banco y es encontrada exitosamente.







En las capturas posteriores ilustran en la izquierda se busca una imagen similar a la que se agregó recientemente, y la búsqueda al no encontrar la imagen en el banco regresa imágenes similares exitosamente. En la derecha se elimina la imagen con llave 2044 de la tabla hash.

Si desea realizar sus propias pruebas utilizando el código que se encuentra disponible en GitHub[2] se proponen los siguientes casos:

1. Probar el programa utilizando la interface gráfica para probar las funciones "Agregar", "Buscar" y "Eliminar". Sin embargo, como ya se menciono antes, en la interface se limita el tamaño de la tabla a 100, pero esto puede ser modificado en el código fuente.
2. Realizar pruebas con imágenes visualmente similares o modificaciones de una misma imagen como se ilustró en las pruebas anteriores.
3. Durante las pruebas, no se excedió el límite de 100 de elementos en la lista, por lo que se recomienda trabajar con tamaños similares para garantizar funcionamiento exitoso.

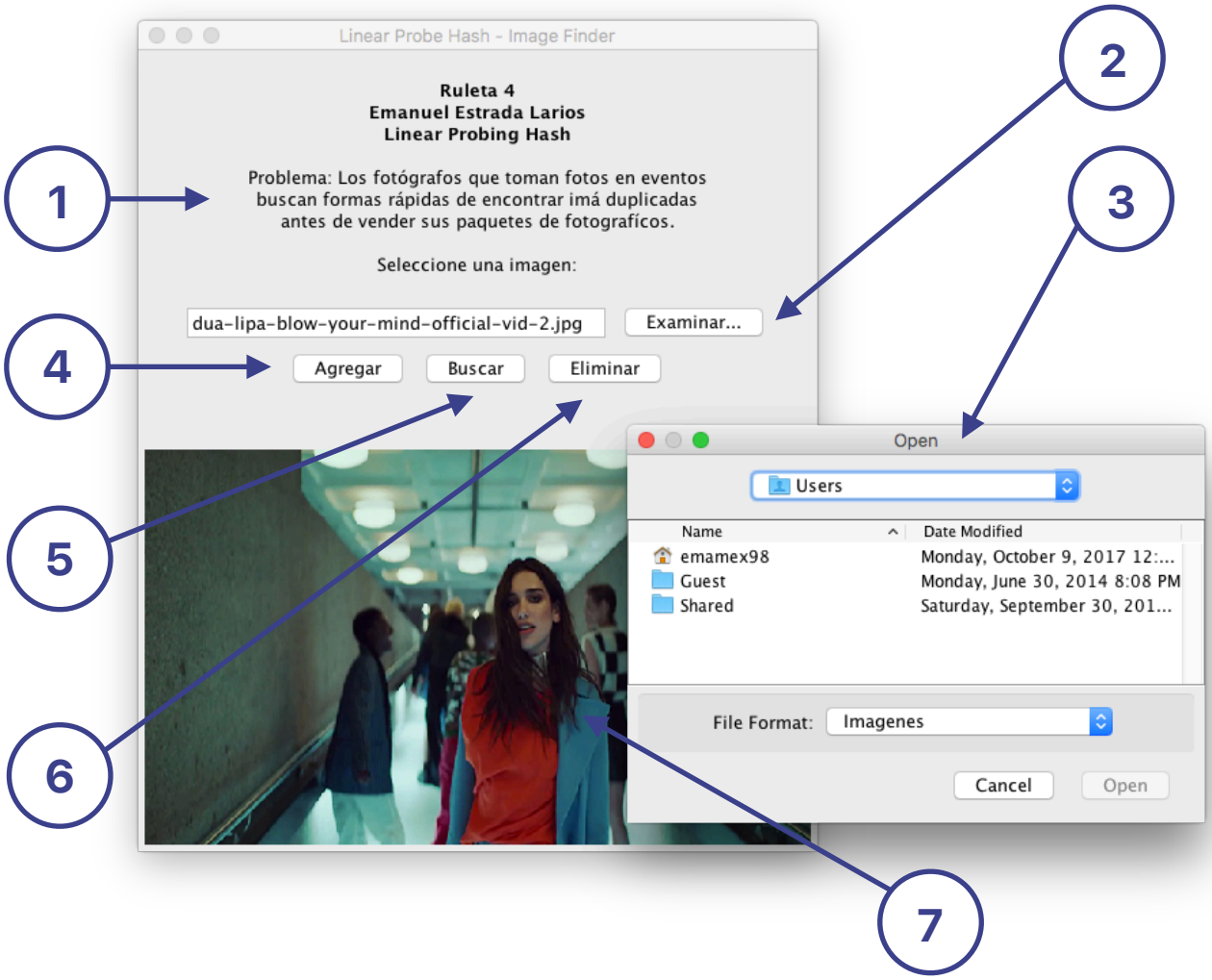
Argumentos

Este programa resuelve el problema propuesto de manera óptima ya que el tiempo de ejecución de un linear probe hash es mínimo ya que las operaciones no recorren por lo general el arreglo completo, sino que acceden directamente a los índices asignados por la

función hash. La complejidad de tiempo de este algoritmo es $O(1r)$, donde $r = \alpha / (1 - \alpha)$, considerando que α representa el factor de carga, que se obtiene de $\alpha = n/m$ [4].

Del mismo modo, el análisis de la imagen es óptimo porque se reduce a solamente 64 colores en escala de grises, que es mucho más eficiente que leer una imagen con tres capas de color de tamaño n^2 pixeles.

Descripción de la GUI



1. Presentación del problema, aplicación de la vida real y descripción del programa.
2. Botón para cargar un archivo, acompañado de un campo de texto en el que se muestra el nombre del archivo seleccionado.
3. Explorador de archivos que se activa al presionar el botón de "Examinar".

4. Botón para agregar la imagen a la tabla hash.
5. Botón para buscar la imagen en la tabla hash.
6. Botón para eliminar la imagen de la tabla hash.
7. Area de display, en dónde se muestra la imagen con la que se está trabajando.

Referencias y Recursos:

[1] "Hash Tables" in Introduction to Algorythms by Cormen, T.
[2] GitHub repository by Emanuel Estrada: https://github.com/emamex98/LinearProbeHash_2_ED
[3] "Linear Probing Hash Tables" by RMIT University: http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html
[4] "Looks like it" by Neal Krawetz: http://www.hackerfactor.com/blog/index.php?/archives/432-Looks-Like-It.html