Raimundo Herrera Sufán - 14632152

Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos — 1' 2017

Tarea 3

Análisis Teórico

Tipo de tabla de hash

La tabla escogida corresponde a una tabla de *hash* con direccionamiento cerrado, utilizando el método del encadenamiento. Se realizó esta decisión porque el manejo de los elementos es sustancialmente más sencillo en este tipo de tablas, no hay que realizar ningún tipo de *probing* ni abstracciones correspondientes al arreglo utilizado en direccionamiento abierto. Además, la utilización de *buckets* hace más explícita y clara la forma de obtener los elementos, junto con, para el caso de mi programa, facilitar en términos de complejidad el *rehashing*.

Funciones de *hash* y complejidad

1. **Hash incremental**

   La función de *hash* incremental se compone de dos casos, el caso cuando no hay un estado previo, y cuando si lo hay (ahí se incrementa).

   El pseudo-código base para realizar la función de *hash* incremental en el caso donde no hay un estado previo, es el siguiente:

```
hash = 0;
for (int i = 0; i < puzzle -> height; ++i) {
    for (int j = 0; j < puzzle -> width; ++j) {
        color = puzzle -> matrix[i][j];
        hash ^= numbers[i][j][color];
    }
}
```

   Donde *numbers* es un arreglo tridimensional de números aleatorios generados con la librería *pcg*. En cada coordenada i, j, k, se almacena un número aleatorio correspondiente a un color en una celda específica, es decir, el k-ésimo elemento de la coordenada i, j corresponde a al k-ésimo color para esa coordenada.

   De este modo, las colisiones son posibles, pero reducidas. Dos colores en distintas coordenadas pueden tener el mismo valor, tanto por la aleatoridad de los mismos, como por el numero de *bits* de cada número, ya que los números generados corresponden a números aleatorios de 32 *bits*.

   La pseudo-uniformidad viene dada por la utilización de números aleatorios.

   La función consiste en realizar `XOR` sucesivos entre todos los números correspondientes a los colores de cada celda. Al recorrer cada puzzle se identifica el color de la celda, y en base a ese número entre 0 y 7, se determina el elemento correspondiente en la lista, `numbers[i][j][k]`. Así, para un puzzle, se tienen tantos números aleatorios como como coordenadas, es decir, $height \times width$. Por consiguiente, se realizan esa cantidad de operaciones `XOR` *bitwise* entre dichos números.

Complejidad

En términos de complejidad, la cantidad de pasos realizados para obtener el *hash* inicial, es proporcional a las dimensiones del puzzle. Es decir *width* y *height*. Como se realizan `XOR` sucesivos, hay que considerar el costo de realizar estos, pero se considera constante ya que siempre los elementos sobre los cuales se realiza la operación son del mismo tamaño. Del mismo modo, la generación de números aleatorios se realiza una única vez durante la inicialización del programa, y depende de los mismos términos. Así la complejidad corresponde a:

$$\mathcal{O}(height \times width)$$

Para el segundo caso, hay que considerar los incrementos. El algoritmo empleado para obtener un *hash* a partir de uno anterior corresponde a deshacer los `XOR` realizados por cada una de las coordenadas modificadas, y realizar los `XOR` correspondientes a los nuevos valores de las coordenadas. Esto se realiza facilmente ya que `XOR(XOR(A,B), B) = A`. De este modo, para las operaciones relativas a columnas, es decir, aquellas `U o D`, la complejidad es $\mathcal{O}(height)$, y para las relativas a filas, es decir, aquellas `R o L`, la complejidad es $\mathcal{O}(width)$.

2. **Hash perfecto**

La función perfecta empleada consiste en concatenar todos los números correspondientes a los elementos de cada coordenada de la matriz. Esto se realiza en base al siguiente pseudo-código:

```
hash = 0;
for (int i = 0; i < puzzle -> height; ++i) {
    for (int j = 0; j < puzzle -> width; ++j) {
        color = puzzle -> matrix[i][j];
        hash <<= 3;
        hash += color;
    }
}
```

Donde hay que considerar que las operaciones son realizadas con números de tipo `mpz_t` debido a su potencial gran tamaño, de modo que las operaciones sobre ellos se realizan de otra manera. Sin embargo, para términos de complejidad y la posterior demostración se considerará, por simplicidad, estas operaciones.

La intención de la función es obtener un número único correspondiente a la concatenación binaria de 3 *bits* por coordenada. Se utilizan *shift lefts* con parámetro 3, lo que, expresado en enteros, corresponde a multiplicar por 8 cada vez que se va a sumar un nuevo elemento.

De este modo, como se suma un número correspondiente al color de la celda, cuyo valor oscila entre 0 y 7, al realizar 3 *shift lefts*, lo que queda son ceros a la derecha del número dispuestos para agregar el valor del color, y que funcione como una concatenación.

Intuitivamente, la función corresponde a enumerar todos los posibles estados, obteniendo un número que no solo considera la suma de cada valor asociado a un color, sino que también el orden en el que se agrega al *hash*.

Complejidad

En términos de complejidad, la cantidad de pasos realizados para obtener cada *hash*, es proporcional a las dimensiones del puzzle. Es decir *width* y *height*. Como se realizan operaciones de *shift left* y suma, se consideraran como operaciones de tamaño constante. Por ende, al realizar una para cada celda del puzzle, el análisis es análogo al *hasheo* inicial anterior, sin embargo, en términos de memoria, estos cálculos pueden volverse un poco más complejos de acuerdo al manejo que haga de esta la librería `GMP`. No obstante, sin perjucio de lo anterior, complejidad corresponde a:

$$\mathcal{O}(height \times width)$$

Donde, las constantes asociadas son un poco mayores a las del *hash* incremental inicial, ya que hay más operaciones por iteración, además de la consideración de memoria. En este caso, no se considera una función incremental, ya que se debe calcular el *hash* entero para cada estado del puzzle. Así, las operaciones de hasheo **siempre** consideran la complejidad $\mathcal{O}(height \times width)$.

Recorrido función de *hash* perfecta

El recorrido de la función perfecta corresponde a todos los enteros entre 0 y $8^{(height\ x\ width)}$, no inclusive. Esto porque se consideran valores binarios entre el cero y el número compuesto por puros unos, en binario. De este modo, por ejemplo para un tablero de $2x2$, los valores van entre `000000000000` y `111111111111`, lo que corresponde a todos los enteros entre 0 y 4096, no inclusive. Esto porque cada color concatenado corresponde a un entero entre 0 y 7, lo que es lo mismo en binario, que los valores entre `000` y `111`. Así el recorrido corresponde a un subconjunto de los enteros, caracterizado por el conjunto:

$$R = \{0, 1, ..., 8^{(width\ x\ height)} - 1\}$$

Inyectividad función de *hash* perfecta

La función, en términos de números enteros que define la función perfecta es la siguiente:

$$h(matrix) = \sum_{i=0}^{height} \sum_{j=0}^{width} 8^{height \times (height-1-i)+(width-1-j)} \times matrix[i][j]$$

Donde, $matrix$ corresponde a la matriz de colores respectiva de cada puzle. Por lo tanto, para demostrar inyectividad, se tiene que cumplir que:

$$\forall\ X, Y \in D,\ h(X) = h(Y) \rightarrow X = Y$$

que considera D como el dominio de todos los puzles posibles.

Para demostrar lo pedido, se puede, equivalentemente, probar que $\forall\ X, Y \in D,\ X \neq Y \rightarrow h(X) \neq h(Y)$. Si se toman dos pares de matrices distintas X e Y, hay que demostrar que $h(X)$ y $h(Y)$ son distintas. Sin pérdida de generalidad, si se considera el caso en que dos matrices difieren solo por un componente, (y que esta diferencia es mínima, o sea de un color en la escala), existe un par particular de índices (i, j) tal que

$$X[i][j] \neq Y[i][j], \text{ y } X[k][l] = Y[k][l]\ \forall (k, l) \neq (i, j)$$

De modo que se tiene que lo que aportan a la sumatoria dichas coordenadas para ambas matrices, es

$$V_x = 8^{height \times (height-1-i)+(width-1-j)} \times X[i][j]$$

$$V_y = 8^{height \times (height-1-i)+(width-1-j)} \times Y[i][j]$$

Y esto es lo único en lo que puede variar el valor de $h(X)$ respecto de $h(Y)$. Asumamos, por contradicción que V_x y V_y son iguales (por notación se utilizará $height = h$ y $width = w$):

$$8^{h \times (h-1-i)+(w-1-j)} \times X[i][j] = 8^{h \times (h-1-i)+(w-1-j)} \times Y[i][j]$$

.

Aplicando logaritmo en base 8, se llega a la siguiente igualdad.

$$h(h-1-i)+(w-1-j)+log_8(X[i][j])=(h-1-i)+(w-1-j)+log_8(Y[i][j])$$

Y despejando se tiene

$$log_8(X[i][j])=log_8(Y[i][j])$$

Volviendo a aplicar una exponenciación a 8, se llega a que:

$$X[i][j]=Y[i][j]$$

Y como en los elementos i,j era en lo único que diferían las matrices, si estos son iguales, se contradice la hipótesis, con lo que se tiene que $h(X)$ e $h(Y)$ tienen que ser distintos.

■