Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Tarea Recuperativa
Hashing Perfecto
Informe de Implementación & Resultados

Integrante: Manuel Saavedra

Profesor: Benjamín Bustos

Auxiliares: Máximo Flores Valenzuela
Sergio Rojas H.

Ayudantes: Claudio Gaete M.
Martina Mora
Matías Rivera

Fecha de realización: 30 de Diciembre de 2024
Fecha de entrega: 31 de Diciembre de 2024
Santiago, Chile

Índice

1. Introducción

En este informe se compararan los tiempos de ejecución del hashing perfecto y sus tamaños resultantes al modificar los parámetros del algoritmo. Para lograr la implementación de hashing perfecto también se implemento una fábrica de funciones de hashing mediante generación de primos utilizando el algoritmo de Miller-Rabin.

Durante el primer experimento se utilizara un conjunto S de enteros positivos aleatorios tal que

$$|S| = n \in [10, 10^7] \tag{1}$$

Mientras que en los siguientes experimentos se utilizara $n = 10^6$ y se modificaran las variables k y c, manteniendo la desigualdad

$$\sum_{i=1}^{10^6} c * b_i^2 \leq k * 10^6 \tag{2}$$

Se espera demostrar que el algoritmo de hashing perfecto es $O(n)$ en tiempo de ejecución y como es afectado por variaciones en k y c para encontrar cuales serian las constantes optimas.

2. Desarrollo

2.1. Herramientas Utilizadas

El proyecto se desarrollo principalmente en C++23, utilizando CMake para armar y compilar el ejecutable final

También se escribió un script de Python 3 para generar los gráficos utilizados en el informe mediante la librería MatPlotLib

El informe final se escribió utilizando Typst.

2.2. C++

La definición de los experimentos y la función `main()` se encuentran en el archivo `src/cpp/main.cpp`, y la implementación del hashing perfecto se encuentra en `src/cpp/lib/PefectHash.cpp` y `src/cpp/lib/PefectHash.hpp`.

Para implementar el hashing perfecto y que se ejecutara en una cantidad de tiempo razonable se utilizo como base la implementación en pseudo-código mencionada en Wikipedia junto al algoritmo Las Vegas encontrado en el apunte del curso.

Como funciones auxiliares se implemento generación y verificación de números primos mediante el algoritmo Monte-Carlo de Miller-Rabin siguiendo el esquema planteado en el apunte del curso.

La función de hash utilizada tiene la forma

$$h_{l(x)} = k_l * x \bmod p_l \bmod (l = 0 ? n : c * b_i^2) \tag{3}$$

tal que h_0 es la función de hash distributiva a usar al inicio.

La construcción de la tabla de hash perfecto sigue de la siguiente forma:

1- Sea `ht` un objeto de tipo `lib::PerfectHash`.
2- Se llama a la función `ht.build(S, k, c, time_limit=1)`.
   - `S` $\rightarrow$ Vector de n enteros a utilizar en la tabla hash, de tipo `std::uint64_t` para evitar overflows.
   - `k` $\rightarrow$ Constante k a utilizar.
   - `c` $\rightarrow$ Constante c a utilizar.
   - `time_limit` $\rightarrow$ Tiempo limite, en minutos, que puede demorar la construcción de la tabla.
3- Se inicializan las variables internas de la tabla de hash, junto a asignar una capacidad de n al vector B_i y al vector `table` donde se almacena el l asociado al par (k_l, p_l) de B_i.
4- Repetir mientras no se exceda el tiempo limite:
   a- Generar el par (k_0, p_0) tal que k_0 es un número aleatorio positivo y $p_0 > n$.
   b- Iniciar un contador `total_size=0` para llevar registro del tamaño de la tabla.
   c- Iniciar los vectores auxiliares `K` y `B_tmp`.
   d- $\forall x \in S, B_{h_0(x)}.\text{pushback}(x)$, con B el vector auxiliar `B_tmp`.
   e- $\forall i \in [0, n-1]$,
      a- $b_i = |B_i|$
      b- total_size $\leftarrow$ total_size $+ c * b_i^2$
      c- $l \leftarrow 0$
      d- $l \leftarrow l + 1$
      e- $\forall x \in B_i$
         a- Si $h_l(x) \in K,$ volver a 4.e.d
         b- $K.\text{pushback}(h_l(x))$
      f- Almacenar B_i en B y l en `table`
   f- Si total_size $> k * n$ volver a 4.
5- Si se construyo la tabla dentro del tiempo limite, devolver 0. En caso contrario entregar 1.

3. Resultados

3.1. Experimento 1

n	**tiempo** [s]	**tamaño**
10^1	0.000	24
10^2	0.005	214
10^3	0.046	2074
10^4	0.469	20136
10^5	5.181	199434
10^6	51.19	1998634
10^7	507.061	20002974

Tabla 1: Resultados Experimento 1

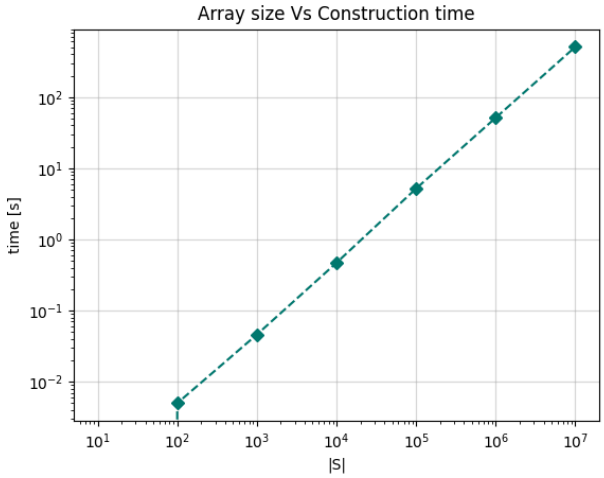

Figura 1: Gráfico del tiempo de construcción por tamaño de S

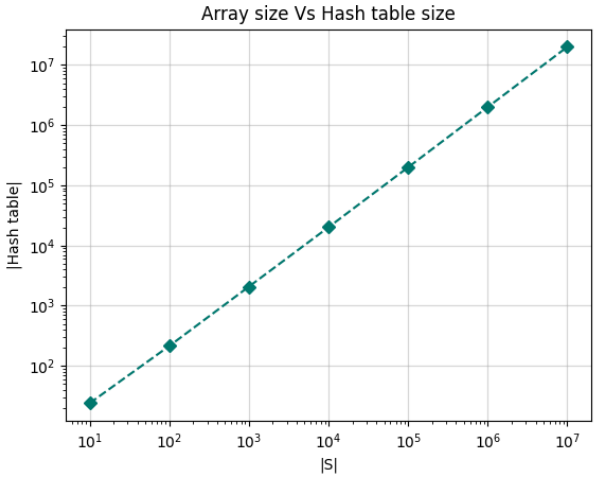

Figura 2: Gráfico del tamaño de tabla por tamaño de S

Tanto Figura 1 como Figura 2 utilizan escala logarítmica para poder visualizar la linealidad ya que n crece de forma potencial.

3.2. Experimento 2

k	**tiempo** [s]	**tamaño**
1	100.452	-
2	50.393	1999014
3	52.658	2000242
4	50.648	2001976
5	49.6	1999076
6	50.118	1997952
7	49.872	1997490
8	48.891	1998956

Tabla 2: Resultados Experimento 2

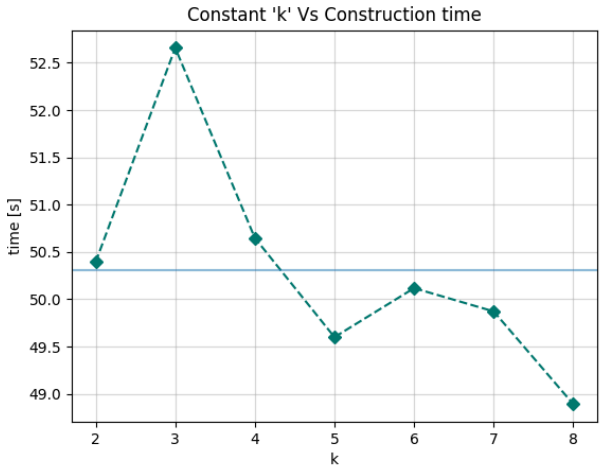

Figura 3: Gráfico del tiempo de construcción por k

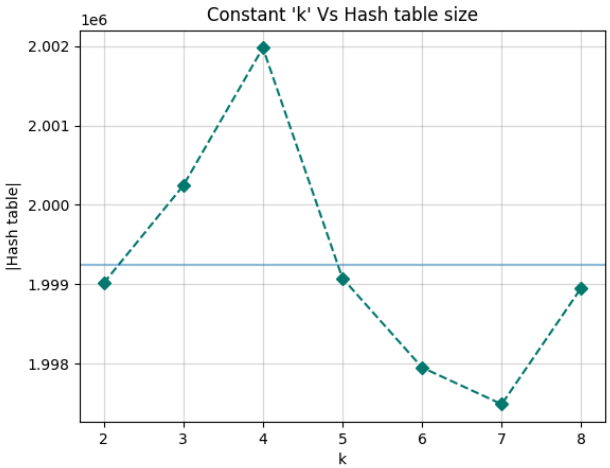

Figura 4: Gráfico del tamaño de tabla por k

3.3. Experimento 3

c	**tiempo** [s]	**tamaño**
1	50.324	2000958
2	26.625	3999896
3	75.219	-
4	67.649	-

Tabla 3: Resultados Experimento 3

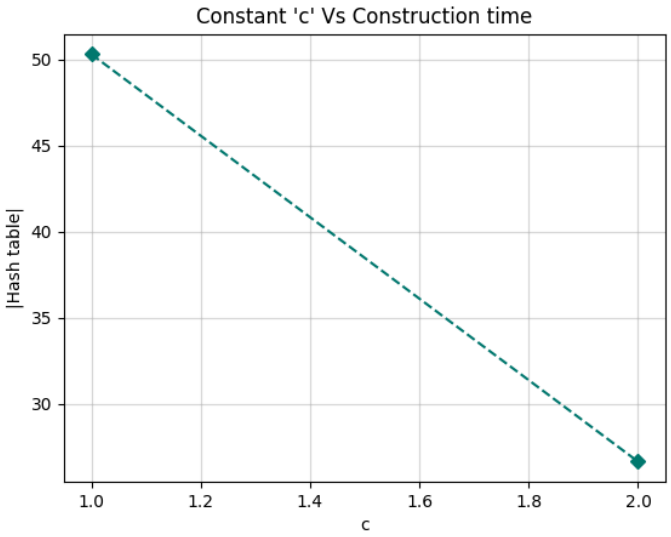

Figura 5: Gráfico del tiempo de construcción por k

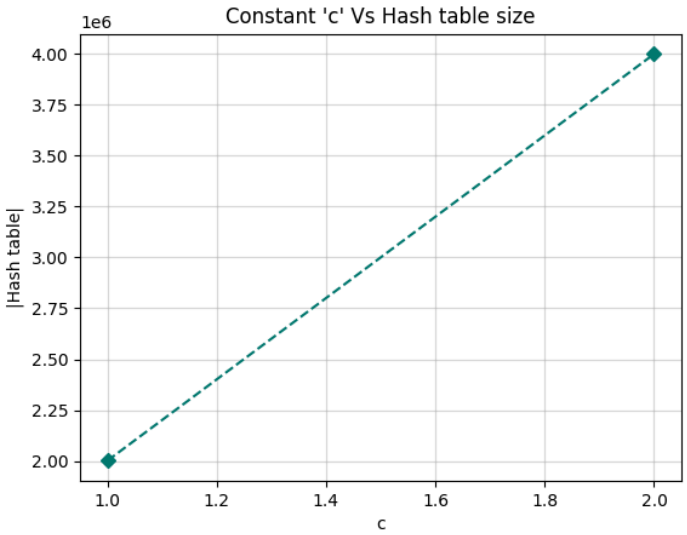

Figura 6: Gráfico del tamaño de tabla por k

3.4. Experimento 4

k	c	**tiempo** [s]	**tamaño**
1	1	100.319	-
2	1	103.32	-
2	2	86.578	-
3	1	53.677	2001388
3	2	86.546	-
3	3	77.612	-
4	1	51.17	2001006
4	2	81.144	-
4	3	76.879	-
4	4	61.513	-
5	1	50.095	1997362
5	2	27.117	4001724
5	3	75.616	-
5	4	73.491	-
5	5	63.562	-
6	1	48.444	1997608
6	2	26.875	4002836
6	3	18.392	5986242
6	4	73.173	-
6	5	60.616	-
6	6	62.441	-
7	1	49.15	2000716
7	2	26.245	3999856
7	3	18.241	5999160
7	4	72.256	-
7	5	72.299	-
7	6	65.209	-
7	7	65.033	-
8	1	50.166	1998302
8	2	25.68	4001580
8	3	17.713	6002088
8	4	15.155	7996056
8	5	61.635	-
8	6	63.915	-
8	7	67.059	-
8	8	67.689	-

Tabla 4: Resultados Experimento 4

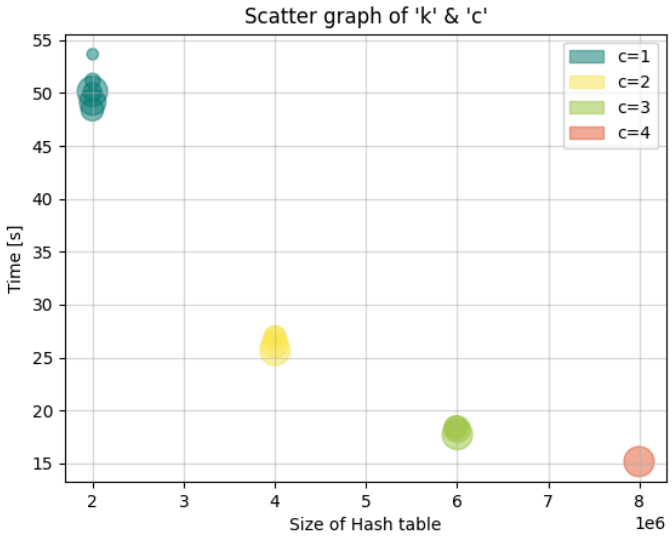

Figura 7: Nube de puntos para experimento 4

En Figura 7 el area de cada punto representa el k, con la mayor area indicando un mayor k.

4. Análisis

4.1. Experimento 1

Se logra apreciar la linealidad del hash perfecto tanto en tiempo de construcción como el tamaño de la tabla.

También se aprecia que el tamaño que toma la tabla de hash tiende a ~$2n$ y que el tiempo para $n = 10^6$ es consistentemente inferior a 60s, por lo que se usara un minuto como el limite de tiempo para los siguientes experimentos ya que demorarse más de un minuto es más ineficiente que el algoritmo "de referencia".

4.2. Experimento 2

Analizando los resultados del experimento 2, incluso en diferentes ejecuciones, se aprecia que el tiempo de construcción tiende a decrecer pero se puede considerar dentro del valor esperado por la naturaleza probabilística del algoritmo.

Tampoco se logra apreciar que aumentar el valor de k tenga un efecto en el tamaño de la tabla final.

4.3. Experimento 3

Es evidente que aumentar c causa que el tiempo sea sustancialmente menor, pero aumenta de forma directa el tamaño de la tabla lo cual afecta la probabilidad de que el algoritmo encuentre una función de hash distributiva que cumpla con los requerimientos.

4.4. Experimento 4

Se aprecia que aumentar c disminuye el tiempo de ejecución pero aumenta el tamaño final de la tabla y observando la forma de la curva hay una relación inversa de forma exponencial entre el tiempo de ejecución y tamaño de la tabla al variar k y c. También se observa que $c \geq\sim \frac{1}{2}k$ causa que el algoritmo no logre terminar dentro del limite de tiempo.

5. Conclusión

Los experimentos 3 y 4 muestran que mantener $c < \frac{1}{2}k$ garantiza que se complete la construcción en un tiempo razonable, lo cual es consistente con la ecuación

$$\begin{aligned} \sum_{i=0}^{n} c * b_i^2 &\leq k * n \\ &\Rightarrow \\ c * \sum_{i=0}^{n} b_i^2 &\leq k * n \\ &\Rightarrow \\ c &\leq \frac{k * n}{\sum_{i=0}^{n} b_i^2} \end{aligned} \tag{4}$$

El valor esperado de $\sum_{i=0}^{n} b_i^2$ es $< 2n$ por lo que es necesario que $c < \frac{k}{2}$ para poder cumplir con las estipulaciones del algoritmo de hashing perfecto.

Por otra parte, el experimento 2 muestra que aumentar k no necesariamente implica aumentar el tamaño final de la tabla pero sí aumenta la chance de encontrar una función distributiva adecuada.

En consecuencia, seria mejor variar c y utilizar un k fijo.
El k fijo le da "holgura" al espacio total sin la necesidad de usarlo en su totalidad, mientras el c variable permite amortizar el espacio de las celdas con colisiones con el espacio de aquellas sin colisiones.
Los experimentos muestran que el espacio total tiende a ~$2n$ por lo que un valor de $k \sim 4$ permitiría suficiente holgura ante un c variable, mientras que $k = 2c + 1$ para un c fijo seria el valor de k ideal.

Al contrario, un c fijo y k variable sería más difícil de optimizar ya que celdas sin colisiones usarían mucho más espacio del que realmente necesitan.

En conclusión utilizar un k fijo, c variable y estructuras de datos eficientes como bit arrays sería la forma ideal de implementar hashing perfecto, y dependiendo de si el espacio total es una consideración o no, utilizar un k tal que $c \in [1, 2, 3]$ permite cortar el tiempo de ejecución a la mitad o un cuarto, pero con $c \geq 4$ la disminución en tiempo no vale el aumento en espacio requerido.