Universidad de Concepción
Ingeniería Civil Informática

DEPARTAMENTO
INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA UNIVERSIDAD DE CONCEPCIÓN

Proyecto 2 :
Estructuras de Datos Compactas

Integrantes: Leonardo Aravena C.,
Patricio Inostroza A.

Asignatura: Estructura de Datos y Algoritmos Avanzados.
Ayudante: Alexander Irribarra C.
Profesor: Dr. Diego Seco N.
Fecha: 26/11/2021

Introducción

El uso de los dispositivos tecnológicos ha crecido en gran manera durante los últimos años, lo que trae como consecuencia un aumento gigantesco en la cantidad de información que se genera y se procesa. Aunque también se ha visto un aumento en la capacidad y reducción de costos en los sistemas de almacenamiento, en ciertos escenarios existe la necesidad de comprimir la información de manera que ocupe menos espacio, pero que no haya pérdidas en la información.

Para lograr que la información ocupe menos espacio manteniendo la integridad de la información se pueden usar dos técnicas: Compresión y Estructuras de Datos Compactas.

La Compresión consiste en reducir el espacio de la información mediante codificación. Según la naturaleza de los datos y las técnicas de compresión se puede reducir bastante el espacio que utilizan los datos originales. Sin embargo, para poder realizar operaciones en los datos, ya sea consultas, cálculos o modificaciones, se necesita descomprimir estos datos para llevarlos a su forma original y ahí realizar las consultas.

Las Estructuras de Datos Compactas permiten reducir el espacio que utilizan los datos y poder realizar operaciones sobre estos datos sin necesidad de volver a su forma original. Con las estructuras de datos compactas apropiadas para los datos que se quieren comprimir es posible realizar consultas en órdenes de tiempo constante o muy cercano a constante, lo que ofrece una ventaja considerable respecto a la realización de consultas sobre datos comprimidos.

En el presente trabajo se reporta la utilización de Estructuras de Datos Compactas para almacenar datos de series de tiempos de rasters, representados como matrices bidimensionales que representan una propiedad de una región del espacio en un instante de tiempo. Se trabajó con tres datasets, donde las dimensiones de las matrices son de 8x8, 128x128 y 512x512, donde por cada dataset hay 120 matrices.

Desarrollo

Para la implementación de las estructuras de datos compactas se utilizó la biblioteca SDSL: *Succinct Data Structure Library* en conjunto con el lenguaje de programación C++.

Para facilitar la lectura de los archivos de texto que contienen los datos, se concatenaron en un solo archivo de texto (usando la herramienta *cat* de Linux) para cada dataset.

Se implementó una clase llamada *Compacta* que contiene los atributos y los métodos necesarios para cumplir con los requerimientos solicitados en el enunciado del proyecto.

Al momento de ejecutar el programa se debe ingresar la cantidad de elementos por fila de las matrices del dataset y la cantidad de matrices de cada dataset, para luego ingresar por entrada estándar los datos, los cuales pueden ser agregados fácilmente utilizando el operador "*<*" para usar un archivo de texto como entrada estándar.

1.- Cálculo de H_0

Para calcular la entropía de orden cero H_0 se tomó como referencia la fórmula:

$$H_0(S) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

Donde *S* es la secuencia de símbolos sobre un alfabeto Σ, *c* representa a los distintos símbolos que aparecen en *S*, n_c es la cantidad de ocurrencias de *c* y *n* la cantidad total de símbolos. *log* se refiere al logaritmo en base 2.

Para obtener el valor de H_0 del dataset se utiliza el método *calculateEntropy()* que no tiene parámetros y devuelve un valor del tipo *double* con el valor de H_0. En este método se recorren los datos ya almacenados en el *int_vector<>* llamado ***iv*** (más detalles en el punto 2) y se van almacenando en una estructura del tipo *map* donde la clave es el dato y el valor es la cantidad de apariciones de ese valor dentro del dataset. Luego de iterar por todos los datos del *int_vector<> iv* se itera por el *map* para calcular H_0 según la fórmula descrita anteriormente, retornando el valor.

A continuación, se observan los resultados de entropía para los datasets.

Dataset	H_0
8x8	1.1223
128x128	1.89803
512x512	4.5655

Tabla 1: *Entropía de orden 0 de cada dataset.*

2.- Almacenamiento en *int_vector<>*

La estructura *int_vector<w>* es una clase para almacenar enteros positivos donde el parámetro *w* especifica la cantidad de bits usados para codificar los enteros, donde *w* puede tomar los valores: 1, 8, 16, 32 o 64.

En este caso los valores de los datos en los tres datasets no supera el valor de 40, el cual puede ser representado por 6 bits, el parámetro *w* se fija en 8. Con esta representación se obtiene una reducción del espacio utilizado si se compara con la representación de los enteros *int* que es de 32 bits.

Al momento de leer los datos por entrada estándar se van almacenando en un *int_vector<8>* para luego ser utilizado como parámetro del constructor de la clase *Compacta* donde se almacena en el atributo *int_vector<8>* ***iv***.

La siguiente tabla muestra en detalle la comparación de tamaños, del *int_vector<8>* ***iv*** y el tamaño del archivo de texto que contiene todas las matrices de los datasets originales.

Dataset	Tamaño int_vector<8>	Tamaño archivo txt
8x8	7.69 kB	24 kB
128x128	1.88 MB	5.7 MB
512x512	30 MB	90 MB

Tabla 2: *Diferencias de tamaño entre int_vector<> y el archivo con las matrices de los datasets.*

Es bastante notable la reducción de tamaño entre la estructura compacta *int_vector<>* y el tamaño que ocupan las matrices almacenadas en el archivo de texto, siendo aproximadamente de ⅓ del tamaño original.

3.- Codificación con bitmap

En la codificación de los datos se utiliza bitmap que va a almacenar un 1 por cada ocurrencia de valor que sea distinta a la que lo sigue, y un 0 por secuencias que sean repetidas dentro de la secuencia almacenada en *int_vector<8>* ***iv***. Por cada 1 ingresado en el bitmap, se va a almacenar el entero de la secuencia en un vector de enteros, que luego será traspasado a un *int_vector<8>* ***v*** (Se crea un vector de enteros para conocer el tamaño de cuantos elementos almacena y realizar la construcción del *int_vector<8>* ***v***, posterior a eso el vector es liberado).

Para el bitmap se utiliza la clase *bit_vector* para crear la estructura ***signo*** donde se pueden almacenar bits.

El tamaño utilizado por *int_vector<8>* ***v*** , *bit_vector* ***signo*** y el tamaño total de ambas estructuras para cada dataset se puede observar la siguiente tabla.

Dataset	Tamaño bit_vector signo	Tamaño int_vector<8> v	Tamaño total (ambas estructuras)
8x8	968 B	408 B	1376 B
128x128	0.23 MB	0.18 MB	0.41 MB
512x512	3.75 MB	6.20 MB	9.95 MB

Tabla 3: *Tamaño de estructuras bit_vector, int_vector<> y la suma de ambas.*

Con estas dos estructuras se puede reconstruir la información original sin pérdidas. Cabe notar que el tamaño que suman ambas estructuras es cercano al 10% del tamaño del archivo de texto que contiene el dataset original.

4.- Representación con *rrr_vector* y *sd_vector*

Dado el bit vector creado a partir del bitmap, se debe poder realizar la operación rank utilizando una de las estructuras *rrr_vector* o *sd_vector*, dependiendo de cuál utiliza menor espacio.

La clase *rrr_vector* es un vector que almacena bits, y se puede crear a partir de un bit vector ingresándolo por el constructor. Esta estructura permite además de almacenar, poder realizar la operación rank en tiempo constante. Se hace uso de la clase *rank_1_type* para permitir su soporte de rank para el bit 1, existiendo su análogo *rank_0_type* para bit 0.

La clase *sd_vector* es similar a *rrr_vector* en métodos y construcción a partir de un bit vector, y permite las mismas operaciones de rank 0 y rank 1.

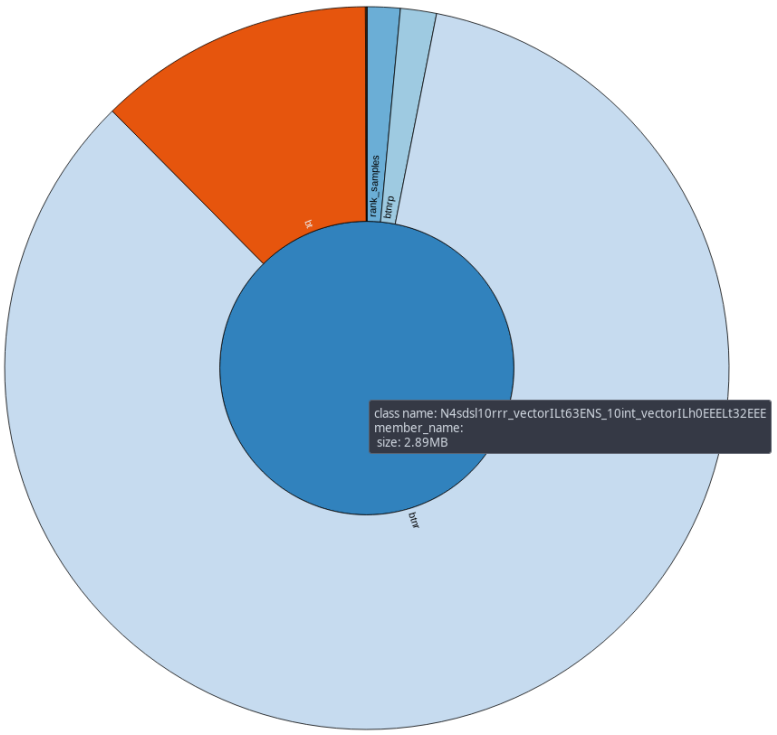
Para poder obtener el tamaño, se crearon ambas estructuras utilizando los distintos datasets y la función *size_in_bytes* para consultar el tamaño en bytes, la siguiente tabla posee los valores respectivos:

Dataset	Tamaño rrr_vector	Tamaño sd_vector
8x8	355 B	590 B
128x128	116 KB	150 KB
512x512	3.03 MB	3.93 MB

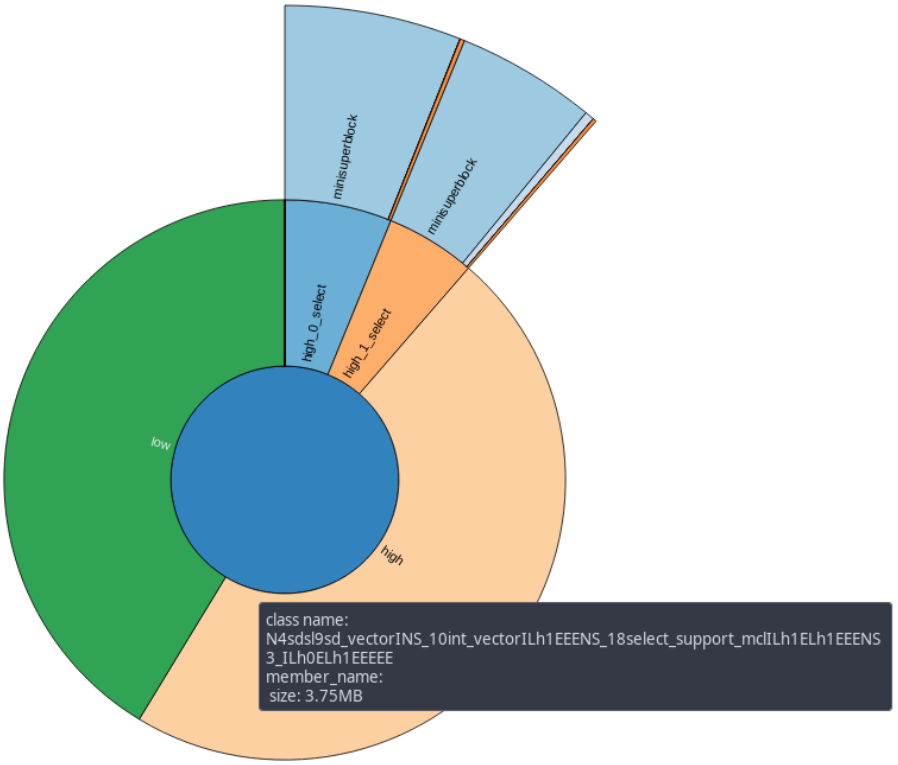
Tabla 4: *Tamaño de estructuras rrr_vector y sd_vector.*

Se puede apreciar cómo *sd_vector* utiliza más espacio que *rrr_vector*.

A continuación se observan las representaciones del espacio utilizado por los campos internos de las estructuras *rrr_vector<>* y *sd_vector<>* para el dataset 512x512.



Representación de tamaño para estructura rrr_vector<> con dataset 512x512.



Representación de tamaño para estructura sd_vector<> con dataset 512x512.

Para obtener el tiempo de consultas, se realizaron 100 consultas con valores aleatorios que van desde 1 a N-1, luego se obtuvo un valor promedio que se observa en la siguiente tabla:

Dataset	Tiempo Rank rrr_vector (µs)	Tiempo Rank sd_vector (µs)
8x8	1.08	0.45
128x128	1.09	0.73
512x512	1.26	0.74

Tabla 5: *Tiempo de consulta rank.*

Se nota claramente cómo en ambos casos la consulta de rank se realiza en muy poco tiempo, donde el tamaño del dataset tiene casi nula influencia. La operación rank de *sd_vector<>* es algo más rápida que la de *rrr_vector*, pero a esta escala de valores no se puede considerar que esa diferencia sea significativa.

5.- Matrices de diferencias en *k2-tree*

La matriz de diferencias es una matriz binaria cuyos valores se obtienen a partir de dos matrices de datos consecutivas al comparar sus elementos en la misma posición *[x][y]*. Si son iguales en la matriz binaria se almacena un 0, si tienen elementos distintos se almacena un 1. Luego se requiere almacenar cada una de estas matrices en un *k2-tree.*

k2-tree es una estructura compacta en forma de árbol para grafos que toma ventaja de áreas grandes que tengan valores vacíos. La implementación de *k2-tree* de SDSL permite recibir en su constructor una matriz de adyacencia en la forma de *vector<vector<int>>*, que en este caso es la matriz binaria de diferencias. Entre sus métodos públicos destacan: *adj(i,j)* que permite determinar si los nodos *i*, *j* son adyacentes y *neigh(i)* que, dado el nodo *i*, retorna un vector con todos sus vecinos.

Para obtener las matrices binarias de diferencias se recorre el *int_vector<8>* ***iv*** mediante dos iteradores para comparar los elementos de una misma posición en matrices consecutivas. Si los elementos son iguales se inserta un 0 en la matriz binaria (implementada como *vector<vector<int>>*), si son distintos se inserta un 1. Por cada matriz binaria se crea un *k2_tree<4>* y su tamaño en bytes se suma a la variable entera *size_k2* que será reportada luego de comparar todas las matrices del dataset. Opcionalmente, puede guardarse cada *k2_tree* en un vector para su posterior uso.

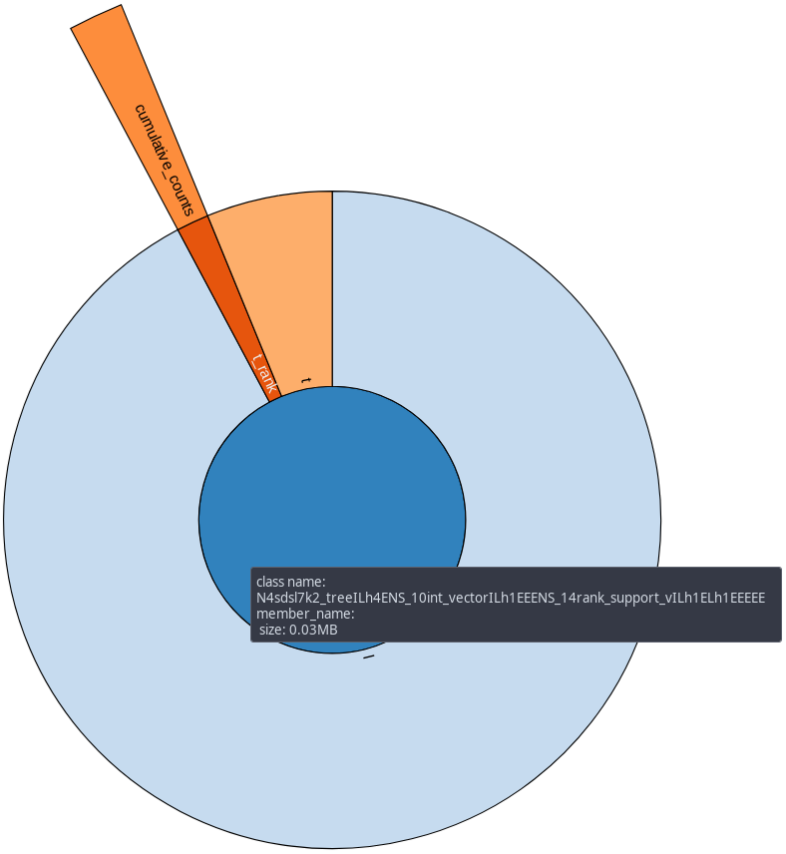
El tamaño de cada *k2_tree* se obtiene usando el método *size_in_bytes(o)* de SDSL, que permite obtener el tamaño en bytes de un objeto compacto *o*.

El tiempo de construcción de *k2_tree* se almacena en una variable que se suma por cada creación de *k2_tree*.

Dataset	Tiempo Construcción	Tamaño
8x8	1.87 ms	5.98 KB
128x128	0.32 s	144 KB
512x512	5.54 s	4.06 MB

Tabla 6: *Tiempo de construcción y tamaño de k2-tree por cada dataset.*

En este caso se nota claramente como el tiempo de construcción y el tamaño dependen de las dimensiones de las matrices desde las cuales se construye.



Representación de tamaño estructura de un solo k2-tree para una matriz de diferencias del dataset 512x512

6.- Representación de las matrices de manera compacta

Otra forma de representar las diferencias entre matrices es almacenando la diferencia que existe entre los valores de una misma posición entre matrices consecutivas. Para lograr esto se utiliza el *int_vector<8>* ***iv*** que contiene todas las matrices del dataset y con dos iteradores se van comparando los elementos entre matrices. Si son iguales se avanza a la siguiente posición, si son distintos se guarda la diferencia en *vector<int>* ***diff*** y se avanza a la siguiente posición. Luego de recorrer todas las matrices se obtiene la cantidad de elementos que contiene el *vector<int>* para almacenar las diferencias de manera compacta en *int_vector<4>* ***vdif***. Se eligen 4 bits porque se conoce la naturaleza de los datos con los que se trabaja y las diferencias se pueden representar con 4 bits.
Puesto que la estructura *int_vector<>* no permite números enteros negativos se utiliza de manera adicional *bit_vector* ***signo*** para representar el signo de la diferencia.
Luego se recorre el vector *diff*, si el elemento es negativo se copia el valor absoluto en *vdif* y en *signo* se inserta un 1. Si el elemento es positivo se copia en *vdif* y en *signo* se inserta un 0. Cuando se terminan de recorrer todos los elementos de *diff* se libera su memoria.

Dataset	Tamaño int_vector<4> vdif	Tamaño bit_vector signo	Tamaño Total
8x8	872 B	224 B	1096 B
128x128	264.85 KB	66.22 KB	331.07 KB
512x512	11.15 MB	2.79 MB	13.94 MB

Tabla 7: *Tamaños de int_vector<> de diferencias, bit_vector de signo y su tamaño en conjunto*

Conclusiones

De acuerdo a los experimentos realizados y los resultados que obtuvimos, se pudo aplicar correctamente las estructuras de datos compactas de la biblioteca SDSL, que permiten una reducción de tamaño significativa, en nuestro caso, de hasta un 10% en relación al tamaño del archivo original, lo que implica que es de gran utilidad para tipos de datos enteros que son similares y aparecen con alta frecuencia.

La documentación y los ejemplos de la biblioteca nos permitió conocer el uso de las estructuras, con el fin de aprovechar las ventajas que ofrecen en compresión de datos y en proyectos futuros considerar la implementación de las estructuras utilizadas en este proyecto o inclusive las no empleadas.