Universidad del Valle

Análisis y Diseño de Algoritmos I

Facultad de Ingeniería

Escuela de Ingeniería de Sistemas y Computación

PROYECTO

Integrantes

Marroquín Almeida Alejandro - 1942529

Díaz García Alessandro - 1940983

Castillo Molina Juan Manuel - 1941563

Acuña Vanegas Brayan Stiven - 1940805

PROBLEMA 1 - REESTRUCTURACIÓN DE LA COMPAÑÍA

Descripción de la solución

Para la solución del problema se usa una estructura de datos de nombre Disjoint Set. Esta estructura contiene un conjunto de elementos que están divididos en varios conjuntos no intersectados. Mediante el uso del Disjoint Set se pueden realizar operaciones como la fusión de algunos de estos conjuntos, encontrar el miembro representativo de un conjunto, o realizar consultas sobre si un par de elementos pertenecen al mismo conjunto, lo que hace esta estructura la más útil para la representación del problema planteado.

Ya en el código se usa un método find, el cual se encarga de buscar el representante del conjunto al que pertenece x. Se tiene un condicional, en el cual se pregunta si el elemento x es el padre de sí mismo. De no serlo se vuelve a llamar recursivamente a find con el padre del elemento. Esto se repite hasta hallar al representante del conjunto y retornarlo.

Se tiene también el método union, el cual se encarga de unir un par de conjuntos en uno solo.

Por último se tiene el método leer_instrucciones, que se encarga de procesar la query correspondiente, estos son: Si el query 1, se usa la función Union para unir dos departamentos de un conjunto; si es 2 se usa Union múltiples veces para unir un rango de departamentos; finalmente el query 3 es una consulta para saber si los departamentos pertenecen a un mismo conjunto o no.

Complejidad de la solución

La complejidad de la solución dependerá de las entradas proporcionadas al programa, ya que realiza diferentes operaciones de acuerdo a esto y por tanto su complejidad puede ser variable.

En cuanto al método *find*, su complejidad viene a ser O(n) en el peor de los casos, que se da cuando se debe recorrer toda la lista de padres del conjunto al que pertenece para finalmente encontrar el representante del conjunto. En el mejor caso es O(1), que se da cuando el elemento es el mismo representante del conjunto.

El método *union* viene a tener una complejidad de O(n) igualmente, y esto sucede cuando se debe recorrer toda la lista de padres de ambos conjuntos a unir para finalmente hallar al representante de los mismos.

El método *leer_instrucciones* es simplemente O(1) ya que solo se limita a leer las instrucciones del archivo de texto y escoger el query a realizar que corresponda al mismo.

En resumen, el programa solución tiene una complejidad de O(n), que viene dada por los peores casos que pueden ocurrir con los métodos *find* y *union*. Pero esta complejidad puede ser variable, dependiendo de las operaciones que se quieran realizar y las entradas que se escojan al ejecutar el programa.

Pruebas

Entrada	Salida
10 6 1 1 2 1 2 3 1 3 4 3 1 4 2 8 10 3 2 9	true false
256 14 2 1 15 2 15 50 1 50 51 3 51 33 2 13 75 2 66 90 3 43 69 2 100 150 3 123 89 2 90 100 3 123 89 2 200 220 1 211 145 3 200 100	true true false true true
20 3 2 10 20 2 1 10 3 4 15	true
38 10 2 8 15 2 4 21	true false true

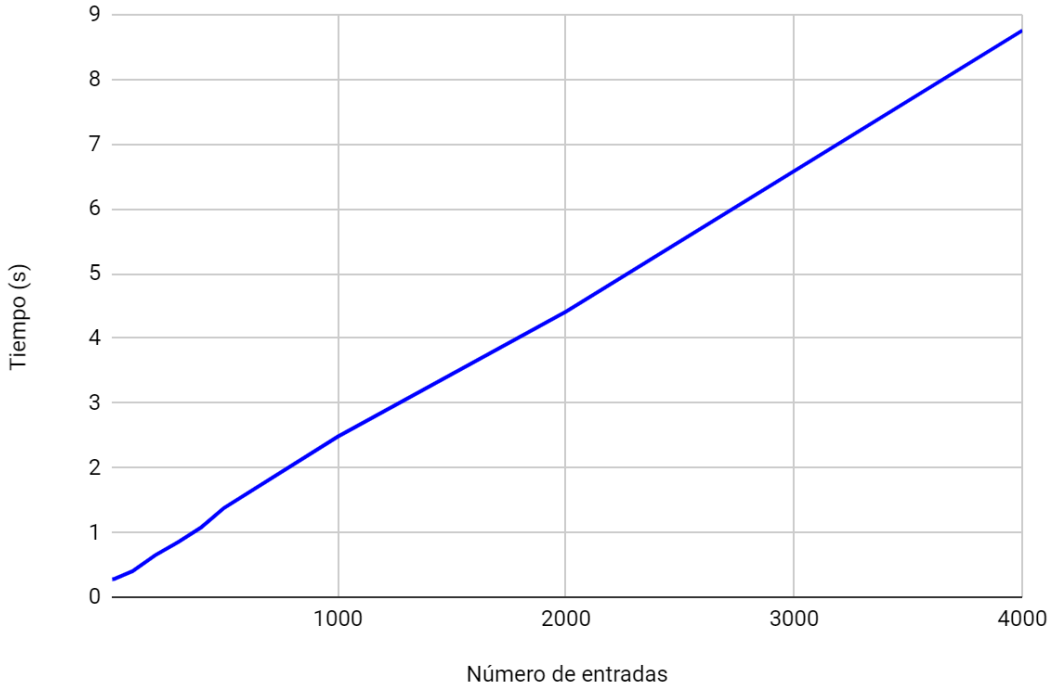
3 6 19 2 25 30 3 9 27 1 18 27 3 9 27 2 30 34 2 33 38 3 31 37	true
10000 15 2 1000 2000 2 3000 4000 2 5000 6000 3 2000 3000 3 4000 5000 1 2000 3000 3 2000 3000 1 4000 5000 3 4000 5000 2 6001 6999 1 6001 5999 3 6000 6001 2 9000 10000 1 6666 9999 3 6543 9876	false false true true true true
1234 8 2 150 333 2 50 101 3 45 200 1 65 222 3 88 312 2 987 1234 1 256 1000 3 278 1156	false true true
74511 19 2 123 333 2 456 678 3 321 654 1 254 444 3 321 654 2 10000 20000 2 45000 70000	false false false true true false true

3 13451 65342 2 20000 45000 3 13451 65342 2 1000 2345 2 4567 7865 1 1111 6666 3 2234 5653 2 7000 7500 2 8234 8999 3 7000 8000 2 7500 8234 3 7000 8000	
150 7 1 10 11 1 11 12 3 10 12 2 15 80 3 10 70 1 11 65 3 10 70	true false true
46000 12 2 1000 5000 2 7000 10000 2 15000 20000 3 1111 7777 3 9876 18456 1 2345 9654 1 8543 16983 3 4876 19876 1 4999 16321 3 4876 19876 3 1111 7777 3 9876 18456	false false true true true true
25000 13 2 24000 25000 2 22000 23000 2 20000 21000 3 23000 24000 3 21000 23000 2 23000 24000 2 21001 23000	false false true true

1 21000 21001 3 20000 25000 2 1 10000 2 10001 15000 1 5000 12500 3 3 14569	

Además de las pruebas que se adjuntan en este documento, se realizaron 10 pruebas con entradas mucho más grandes haciendo uso solamente del query2 e incluso con valores de N mayores a 10^5, donde se observa que este query tiene una complejidad lineal. Se puede observar que el tiempo de respuesta del programa es proporcional a la cantidad de entradas que se le proporcionan al programa. Por tanto su complejidad es de O(n).

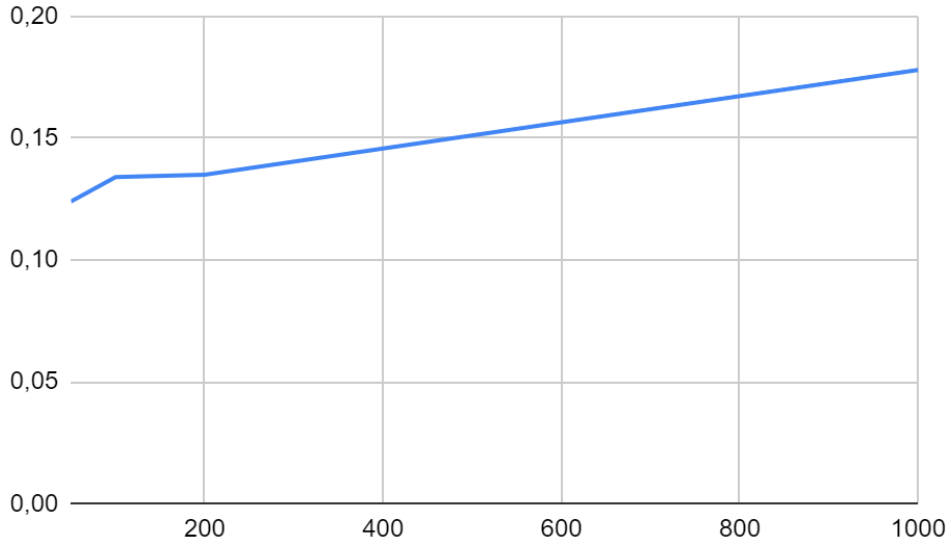
Prueba	**Entradas**	**Tiempo (s)**
1	10	0,279
2	20	0,285
3	100	0,405
4	200	0,653
5	300	0,856
6	400	1,08
7	500	1,38
8	1000	2,484
9	2000	4,414
10	4000	8,753



Usando Query2 complejidad O(n)

También se hicieron pruebas exclusivamente con el Query1, con tamaños de entrada muy grandes, lo cual nos dió un crecimiento lineal:

Prueba	Entradas	Tiempo (s)
1	25	0,166
2	50	0,124
3	100	0,134
4	200	0,135
5	1000	0,178



Usando Query1 complejidad O(n)

Podemos concluir que la complejidad obtenida en el análisis teórico concuerda con las pruebas realizadas con el programa. El programa tiene una complejidad de O(n).

PUNTO 2 - AUTOCOMPLETADO 2.0

Descripción de la solución

Para la solución de este problema se usa un Prefix Tree (una estructura de datos de tipo árbol), la cual permite la operación base como la inserción, búsqueda o eliminación de datos con un costo O(n).

Para la implementación de la estructura fue necesaria la creación de una clase auxiliar (Nodo), la cual conserva los datos base de cada sección del árbol, tal como letra (su identificador), words (la cantidad de palabras que se desprenden de este punto), sons (un diccionario con las conexiones que tiene con sus ramificaciones) y un identificador que indica si dicho nodo es el final de una palabra o no.

En la clase principal (Trie), se especifican las funciones que arman el árbol (insertar, buscar prefijos, buscar palabras y la eliminación de estas), todas estas operaciones con un costo lineal O(n).

Ahora bien, para el desarrollo del proyecto, fue necesario la adición de una función llamada buscar buscar sugerencia, ubicada en la clase Trie y que estaría encargada de devolver la respuesta al problema. Y la clase auxiliar de la clase Nodo que elegiría la mejor ramificación.

Complejidad de la solución

La complejidad estará ligada a la cantidad de instrucciones que se realicen. Como se sabe, la inserción de palabras a la estructura tiene un costo O(n). Una vez establecidas, se procede a la búsqueda de las sugerencias, las cuales conllevan 2 bucles, el primero, un for, que busca el prefijo de mayor longitud coincidente en ambos.

El segundo es un ciclo while que recorre el árbol a medida que llama a la función auxiliar del nodo (elegir_siguiente), la cual analiza las opciones que cada nodo hijo ofrece, empezando por la cantidad de hijos que esté, a su vez, posea (esto con un coste lineal O(k), donde k representa la cantidad de hijos que tiene. En caso de generarse un empate, se tiene en cuenta el orden lexicográfico que tengan. En casos donde las palabras ingresadas distan bastante unas de las otras, la cantidad de nodos hijos suele disminuir a medida que se avanza en el Trie. Con ello, el buscar la sugerencia representa un gasto O(n) + O(n).

Por otro lado, como se mencionó en el punto anterior, la complejidad dada por la lectura de las instrucciones es despreciable.

Como resumen, las acciones que operan el problema tienen un costo O(n), lo que conlleva a un costo general O(n).

Pruebas

Entrada	Salida
8 1 pera 1 perro 1 parte 1 caldo 1 cama 2 perla 2 camionero 2 calzado	perro cama caldo
10 1 salsa 1 sal 1 sarten 1 vino 1 sol 1 viento 2 saltar 2 ventisca 2 cesta 2 balon	sal viento sal sal
15 1 salmon 1 carne 1 dudas 1 puerta 1 cifra 1 pollo 1 lujos 1 vida 1 carmen 1 vista 1 pantalon 2 zapato 2 lejos	carmen lujos carne carmen

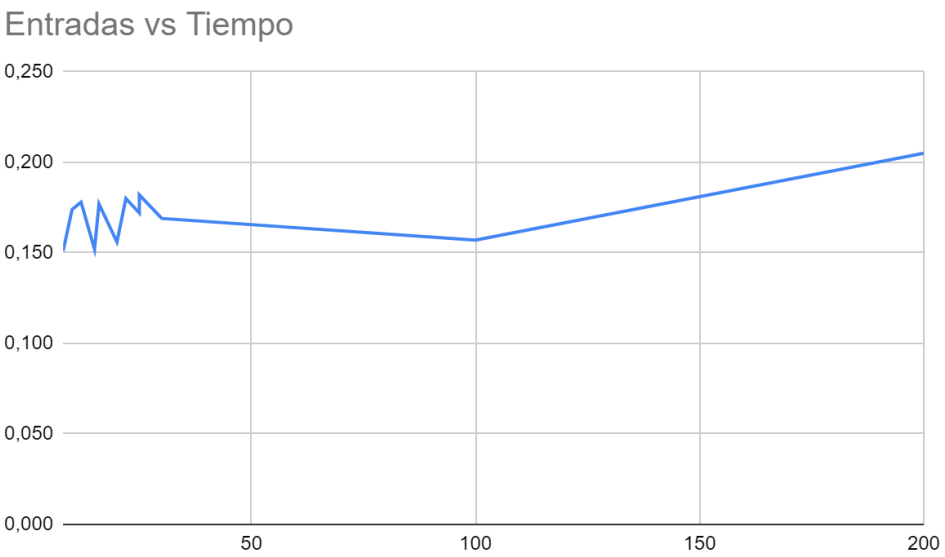
2 carnicero 2 avion	
25 1 escape 1 salida 1 entonces 1 camara 1 paso 1 intento 1 fantasma 1 mucho 1 merece 1 ver 1 otro 1 cielo 1 gusta 2 gris 2 pelo 2 gasto 2 panda 2 estufa 2 letrero	gusta paso gusta paso escape camara
20 1 animal 1 ganso 1 gala 1 sopa 1 soltar 1 decimal 1 decimos 1 diente 1 camino 1 espada 1 saltar 1 soltar 2 espuma 2 hoja 2 soldar 2 soldado	espada soltar soltar soltar decimal camino espada soltar

2 desconectar 2 caminar 2 espalda 2 letras	
30 1 intento 1 pimienta 1 medalla 1 pared 1 monitor 1 torre 1 cama 1 calzado 1 sabanas 1 armario 1 piso 1 esponja 1 talco 1 botella 1 hoja 1 vaso 2 destornillador 2 cabeza 2 cuaderno 2 pisar 2 pista 2 precio 2 madera 2 escoba	pimienta calzado calzado piso piso pimienta medalla esponja pimienta calzado pimienta calzado botella esponja
25 1 vida 1 saltar 1 casa 1 cazador 1 escudo 1 salud 1 bienestar 1 espejo 1 lodo 1 nodo	lejos cajon vida vida cajon saltar casa vida vida lodo

1 cajon 1 lejos 1 cerca 1 asador 1 viento 2 lupa 2 pendula 2 vino 2 vinilo 2 tractor 2 saltamontes 2 caserio 2 visa 2 vista 2 loto	
12 1 escoba 1 palo 1 lento 1 ceja 1 litro 1 pista 1 poner 1 escuchar 2 palco 2 lentejas 2 cesta 2 escudo 2 lindo	palo lento ceja escuchar litro
22 1 lejos 1 cerca 1 espejismo 1 vaso 1 plato 1 caos 1 celular 1 pelo 1 estrecho 1 camino	termo celular caos camino pelo

1 sabana 1 gorra 1 sombrero 1 termo 1 hojas 1 llavero 1 mesa 2 teclado 2 cesto 2 caotico 2 circo 2 pelaje	
16 1 pera 1 sastre 1 camara 1 rata 1 caja 1 cable 1 tornillo 1 ventilador 1 bolso 1 roca 2 castillo 2 cielo 2 salir 2 canal 2 vender 2 tornado	cable cable sastre cable ventilador tornillo

Prueba	Entradas	Tiempo (s)
1	8	0,151
2	10	0,174
8	12	0,178
3	15	0,152
10	16	0,177
5	20	0,156
9	22	0,180
4	25	0,172
7	25	0,182
6	30	0,169
11	100	0,157
12	200	0,205



Análisis de tiempos solución 2

Al observar la gráfica de los tiempos, se puede confirmar el análisis previo realizado, de que la complejidad de la solución es lineal O(n). Las pruebas se realizaron con entradas que oscilan desde 10 hasta 200, donde se puede ver un incremento en el tiempo de respuesta del programa. Se concluye por tanto que entre más entradas tenga la solución, más es el tiempo que se tarda en dar respuesta.

Conclusiones

Una vez realizadas varias pruebas con las soluciones planteadas, se observa la eficiencia de las estructuras de datos en la búsqueda y realización de diversas operaciones con conjuntos de datos o árboles.

El Disjoint Set, usado en el primer problema es una herramienta muy útil para la realización de búsquedas y uniones de conjuntos disjuntos, que trabaja de una manera sumamente eficiente, con un costo lineal en el peor de los casos, lo que lo convierte en una estructura de búsqueda muy rápida.

En cuanto a la estructura que se usa en el segundo problema, el prefix tree (o trie), que almacena cadenas de caracteres, muestra su eficiencia en el proceso de búsqueda al realizar búsquedas e inserciones de datos en un tiempo muy corto, lo que lo hace muy útil en tareas como la realizada, que era autocompletar palabras.

Si no fuera por algunas de estas estructuras, programas que tienen entradas similares a las que se plantean en nuestras soluciones, pero implementados de una forma ineficiente, pueden tardar hasta horas en dar una respuesta, o, en el peor caso, imposibilitar dar respuesta alguna.

Aspectos a mejorar

Las soluciones planteadas en los problemas intentaron ser lo más óptimas posibles para una rápida ejecución y obtener la respuesta deseada de forma más eficiente. Quizá se podría mejorar el tiempo de respuesta añadiendo estructuras algo más complejas, pero que en su ejecución terminen simplificando el proceso realizado por el programa. También, con algunas modificaciones, las soluciones puedan resolver problemas con entradas mucho más grandes que las puestas de ejemplo; sin embargo, esto ya queda para una implementación futura.