Trie

(1) Project Experience

Our partnership is good. We worked well. Next time I'm going to start it early because the biggest barrier for our team is procrastination.

My favorite part of the project is the implementation of worklist. It's fun to achieve the same function with different implementations. The least favorite part of the project is the write-up.

I really enjoy the project as it is my first time to work on a CSE homework in a group. This is very different from CSE 143 and I prefer the group work.

The project is better to be explained more in class.

(2) WorkLists

Having several data structures allow us to satisfy different requirement. Array is constant time but it is hard to resize; while linkedlist it's better for managing queues. We can then choose different data structures based on the problem to increase efficiency.

Many applications using the same interface can be converted to a better implementation of the same interface easily. Because our worklist implements the generic queue & stack interfaces, they can be instantly applied to all applications that use these interfaces.

If there are different interface, it is redundant that we have to specify which implementation to use when declaring a variable. Also it is difficult to change it to other implementation.

(3)

TrieMap is used exclusively to store character strings. A valid key-value mapping can only be established if we can iterate through the key to generate a unique "path" in the trie. If we want to map between generic objects, for example, we can't reliably do that using a trie. Even if we are using strings as keys, the keys might be too big in size to be efficient (e.g. if the key is a thousand bytes long, then we have to make a thousand nodes to store one value). Instead, it's much easier to use a general hash algorithm to more efficiently store them.

(4)

Pseudocode using TrieSet/TrieMap

map = new TrieMap<IP address, (boolean) blackListed>

insert < xx.xx.xx.xx , true> to put a single address into black list.

insert < xx.xx.xx , true> to put a group of addresses into black list.

to check whether an address is black listed, trace down till we hit a node that is blacklisted (e.g. to check for 11.22.33.44, first check if 11 is blacklisted, then 11.22 and so on. The address is cleared if we never hit a node that is blacklisted.)

(5)

Explanation:

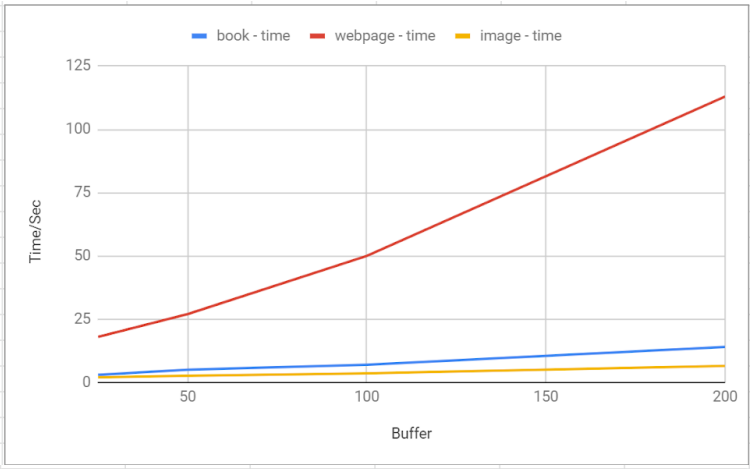
(IPv4) IP addresses always consist of four numbers separated by dots, so using a trie map, we can reach any possible address by tracing through 4 nodes, with the final node being a single address node and the first three nodes being intermediate nodes.

The benefit of that is that we can easily put all addresses under a common address group into the black list with a single edit (e.g. blackList(12.34.56) would blacklist any address starting with 12.34.56 by marking the intermediate node for 12.34.56 as blacklisted).

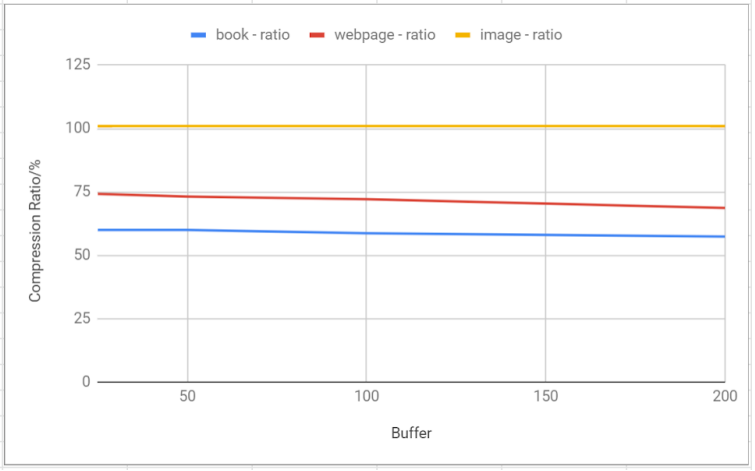
HashSet/HashMap on the other hand, can't treat multiple keys with the same prefix as a group, therefore can't provide the above functionality.

(6)

buffer	book - time	webpage - time	image - time
25	3		
50	5		
100	7		
200	14		
25		18	
50		27	
100		50	
200		113	
25			2
50			2.6
100			3.6
200			6.5



buffer	book - ratio	webpage - ratio	image - ratio
25	60		
50	60		
100	58.7		
200	57.4		
25		74.2	
50		73.2	
100		72.1	
200		68.7	
25			101
50			101
100			101
200			101



We chose three files to run the test: a book file wizard_of_oz.txt (228 KB), a NYT article article.html (1,795 KB), and an image husky.jpg (94 KB). We ran the Zip program with four different buffer sizes for each file: 25, 50, 100 and 200, while recording the time taken and the compression ratio achieved.

In general, we found out that the runtime increases with buffer size, and a higher compression ratio is achieved using a bigger buffer size. We also observed significant memory use under a high buffer size. Our explanation is that with a bigger buffer, a bigger underlying Huffman tree is constructed with each batch of data processed. Therefore we are able to achieve greater compression rate because of better utilization of each Huffman node. However, the downside is that a longer time is taken to navigate within the bigger tree to find the correct spot to insert data.

Our explanation fits our findings specific to each file type. We know that a human-readable text file has lots of repeating words which can be greatly compressed, a jpg file with completely non-repetitive RGB numbers cannot be compressed by much, while the HTML file with a mix of natural language and tags sits somewhere in between.