COL 761

Assignment 1

Dream Miners

Algorithm

Input

The data is read by the program one number at a time using '`fstream`' in C++. Each line is stored in a vector called '`items`' which is used to make or update a branch of the FP Tree by calling the '`makeFPBranch`' function defined in `FPTree` data structure. After a branch is made, the vector is cleared and the next set of items are stored in the vector, which is used to make the next branch and so on till the end of the file is reached.

`node` Data Structure

The node data structure contains the name, the frequency, the depth, the parent node, and a children vector of a node. It also contains an index for its position in its parent's children vector. A pointer to next item of the same name is also present.

`FPTree` Class

The `FPTree` class is a set of `node` data structures connected to each other either as parent and child or as items of same name.

When the `FPTree` variable is declared it makes the root of the tree. A function '`makeFPBranch`' is defined to add or update branches of the FP Tree. It uses depth first search to find a node and increase its frequency by one. When the node is found, it is also added to the linked list of items of same node.

If the node is not found, it creates a new node by calling '`makeFPnode`' function. This function takes the item name, the parent node and its index for parent's children vector as input and initializes the node. It also sets the item frequency to 1. This node is either added to the linked list of same items, if already present or used to create a new linked list.

A variable '`numTrans`' is defined to count the number of times '`makeFPBranch`' is called. This is the number of transactions that are in the file. This variable is also used to define support for frequent items.

The pointers to the head of all linked lists of same items are stored in a vector of nodes named '`head`'.

The correctness the FP Tree can be checked by printing all branches of the tree by using '`printFPTree`' function. It also uses a depth first approach to print the branches. It calls on itself recursively to print the next branch.

A ‘deleteTree’ function is also defined which calls on itself recursively till all its child nodes are deleted.

Other Functions

‘merge’ and ‘mergeSort’ functions are used to sort the ‘head’ vector by the node names. The ‘merge’ function divides the vector into two equal sized vectors. These vectors are then joined such that the new vector is arranged in descending order. The ‘mergeSort’ function calls itself recursively to merge and sort two equal halves of the vector.

The ‘makePath’ function uses a leaf node as an input to return a vector of all items in that branch except the leaf node. This function is called by ‘makeCondTree’ to create conditional FP Tree of the leaf node.

‘makeCondTree’ call ‘makePath’ function to find the items in a branch and make a FP Tree by calling ‘makeFPBranch’ function and sets all the items frequencies to leaf node’s frequency.

The ‘getTotalFreq’ function takes a node as an input and adds the frequency of all items of the same name in the tree.

The ‘mineFrequentItems’ function takes root node of a FP Tree, the support and a vector to store frequent items as input. It also uses a depth first approach to find a node which is a frequent item and adds it to the frequent item vector. If the node is not a frequent item, the function calls itself to find frequent items in its children.

The ‘recursiveMining’ function takes a leaf node, the support and vector to store the frequent item set as input. It then creates a conditional FP Tree of the leaf node using ‘makeCondTree’ function and mines frequent items from it by calling ‘mineFrequentItems’ function. If there are no frequent items in the conditional FP Tree. The function returns if there are no frequent items. Else, it sorts the ‘head’ vector of the tree by calling ‘mergeSort’ function. If there are frequent items in the tree, ‘recursiveMining’ function is called again to mine frequent items from that items conditional FP Tree.

Output

‘fstream’ is used to write a text file which is name ‘freqSets.txt’. The first line of this file is the name of the last item set. The rest of the file rows of names of frequent items.

Mapping

The mapping program reads the ‘freqSets.txt’ using ‘getline’ function. Then ‘stringstream’ is used to covert string to integers and then this row of numbers is stored as a row vector in a 2D vector of integers.

Each row of the 2D vector is used as key to map to an integer using `map` data structure using a loop. The value of a key is increased by one for each key vector. This value starts from the lowest order item name plus one. Since there are no items with names bigger than the lowest order item.

Output

This program then writes a text file named '`mapping.txt`' which contains rows of integers. Each row is a key-value pair. But the first integer of a row is value, and the following are items in key vector.

Data Compression

Introduction:

The primary objective is to illustrate how a dataset can be compressed using a compact representation and later decompressed to retrieve the original data. The provided code demonstrates these processes along with calculating element counts and the compression ratio.

Code Explanation:

1. Libraries: The code begins with including essential C++ libraries- 'iostream', 'vector', 'unordered_map', and 'set'. These libraries are used for input/output operations, working with collections, and managing associations between keys and values.
2. Compression and Decompression Functions: The core functionality of the code is encapsulated in two functions – 'compressdataset' and 'decompressDataset'. These functions emulate the corresponding code functions. They take the dataset and a mapping dictionary as input and produce the compressed and decompressed versions of dataset, respectively.
3. Main function: The 'main' function is the entry point of the program. It starts by defining a sample dataset and a mapping dictionary. Then, it employs the 'compressdataset' function to compress the dataset and the 'decompressDataset' function to restore the original dataset. The element counts and compression ratio are calculated and displayed.
4. Iterating through data: The algorithm iterates through the input dataset itemsets by itemsets. It keeps count of the total number of items in dataset.
5. Input and Output: The input to the 'compressDataset' function is a list of numbers. The output is a compressed version of the input, where frequent items are replaced with single character. Decompressed dataset generated by replacing character in compressed dataset with replaced frequent items.

Key Value Generation and Mapping

1. Header Includes:
   - '<iostream>': Standard input/output stream objects.
   - '<fstream>': File stream objects for file input objects.

- ‘<string>’: String manipulation function.
- ‘<set>’: Container for storing sets of unique integers.
- ‘<vector>’: Dynamic array container for storing itemsets.
- ‘<sstream>’: String stream functionality for extracting values from strings.
- ‘<unordered_map>’: Container for storing mappings beteen itemsets and keys.

2. Custom Hash function (‘SetHash’):
This part defines a custom hash function for ‘std::set<int>’ using bitwise operations. The purpose of this hash function is to ensure that sets of integers have appropriate hash values when used in an unordered map.
3. Main function:
The main function encapsulates the program logic. In main function it takes input file as path of file then open it and read items line by line and if file is not opening it give error ‘Error opening file’. Vector named ‘frequent_itemsets’ that will stores sets of integers(itemsets). Each element of this vector will represent an itemset. String variable ‘line’ will be used to store each line read from the input file. ‘while’ loop used to read each line from the input file. It continues to loop until it reaches the end of the file. Inside loop, a set named ‘itemset’ is declared to temporarily store the integers from each line as an itemset. Inside this loop, the string stream ‘while (iss >> item) {‘extracts integers from the line. The extraction continues if integers can be extracted. Each integer extracted from the line is inserted into the ‘itemset’ set. This ensures that duplicates are automatically removed. Once the entire line has been processed, the ‘itemset’ set is added to the ‘frequent_itemsets’ vector. This effectively stores the current itemset in the vector.
4. Output Read Itemsets:
After reading and processing the itemsets , the code outputs each itemset to the console. This is useful for verifying that the itemsets have been correctly read and processed.
5. Initialize Unordered Map:
An unordered map named ‘itemset_to_key’ is initialized to store the mapping between itemsets and their corresponding keys. The custom ‘SetHash’ hash function ensures that the sets are hashed properly.
6. Assign Key Values:
The program iterates through the ‘frequent_itemsets’ vector and assigns a unique key to each itemset based on the order of discovery. The key is stored in the ‘itemset_to_key’ unordered map.
7. Utilize the Mapping:
Finally, the code uses the mapping stored in the ‘itemset-to_key’ unordered map. It iterates through the map and prints the keys and corresponding itemsets. This demonstrates how the mapping can be used to associate keys with itemsets.

Resources:

1. www.geeksforgeeks for syntax and data structures
2. ChatGPT for getting insights into the code.
3. www.stackoverflow.com for debugging

4. We created a FP Tree program which worked fine but we couldn't join sets for frequent items sets of size 3 or more. We used https://github.com/ankesh007/Data-Mining-Assignments/blob/master/Assignment_1/src_FPTree/FP_Tree.cpp to mine frequent item sets.