CSci 230 — Data Structures and Algorithms
Project 3 — Web Data Mining, Inc.
Preliminary Due Date: Tuesday, Nov. 10th at 11:59:59 pm
Program Due: Tuesday, Nov. 17 at 11:59:59 pm

You have formed a new startup company called "Web Data Mining, Inc." Your main product is the "WebMiner" software that discovers common key-words in a collection of web documents. Using the discovered patterns makes it possible for you to automatically classify web documents under specific categories, much like the categories under Yahoo (www.yahoo.com). Let's look at how WebMiner works with the means of an example, discussed below.

WebPage Data

The input to the program consists of a set of web pages, where each web page is associated with a list of key-words that occur in that page. For example let's look at web pages of Computer Science (CS) and Electrical Engineering (EE) departments at various schools. Looking at the common key-words we want to automatically label each web page as belonging to the category "CS" or "EE". Consider the following database of web pages and their keywords:

```
RPI_1 -- computer_vision, computer_systems, semiconductors
RPI_2 -- computer_vision, computer_systems, programming_languages,
        computation_theory
MIT   -- computer_vision, computer_systems, programming_languages,
        computation_theory, semiconductors
NWU   -- computer_vision, computer_systems, programming_languages,
        semiconductors
UofR_1-- computer_vision, computer_systems, programming_languages,
        computation_theory
UofR_2-- computer_vision, computer_systems, semiconductors
```

For simplicity let's assign each web page a unique number as follows:

```
RPI_1  = 1
RPI_2  = 2
MIT    = 3
NWU    = 4
UofR_1 = 5
UofR_2 = 6
```

Let's also assign each key word a unique number as follows:

```
computer_vision         = 1
computer_systems        = 2
programming_languages   = 3
computation_theory      = 4
semiconductors          = 5
```

The new database now looks like:

```
WebPage Num_KeyWords    KeyWords
1       3               1 2 5
2       4               1 2 3 4
3       5               1 2 3 4 5
4       4               1 2 3 5
5       4               1 2 3 4
6       3               1 2 5
```

This example web database has 6 department web sites, and there are 5 different key words that appear in the pages.

Finding Common Key Words

Let's see how WebMiner finds common key words. The algorithm used by WebMiner takes multiple steps. In each step it counts the number of occurrences of a set of words. In step i it counts sets of size i. We say that a word set is **frequent** if appears in at least **min_per** percent of the web pages. The value of min_per is an input to the algorithm. For example, if min_per = 0.5 (50%), then any set of words that occurs in at least 3 (50% of 6) pages is considered frequent.

Lets go back to our example and look at the different steps, assuming min_per = 0.5

Iteration 1: Count Word Sets of Length 1

In the first iteration we count word sets of length 1. We first have to form a list of **candidate** word sets that we will count in the current iteration. For iteration 1, the candidate set, denoted $\mathcal{C}_1$, is the set of all the unique words under consideration, i.e., $\mathcal{C}_1 = \{1, 2, 3, 4, 5\}$.

The next step is to count the number of times each candidate occurs in the database. We get the following result:

Candidates:	1	2	3	4	5
Count :	6	6	4	3	4

For example, the above table says that word 1 appears in all 6 web pages. At this stage if any word set is not frequent (occurs less than 3 times) we drop it from consideration. The remaining word sets form the set of frequent word sets of length 1, denoted $\mathcal{F}_1$. For the above example $\mathcal{F}_1 = \{1, 2, 3, 4, 5\}$.

Iteration 2: Count Word Sets of Length 2

The starting point for step 2, is the set of frequent words $\mathcal{F}_1$ computed in Step 1. We form a new candidate set $\mathcal{C}_2$ by looking at all distinct pairs of members of $\mathcal{F}_1$. For our example,
we get $\mathcal{C}_2 = \{12, 13, 14, 15, 23, 24, 25, 34, 35, 45\}$ (note: 12 actually stands for the word set $\{1, 2\}$).

After forming the candidate set we count their frequency in the database:

Candidates:	12	13	14	14
Count :	6	4	3	4

Candidates:	23	24	25
Count :	4	3	4

Candidates:	34	35	45
Count :	3	2	1

We now select those candidates that are frequent (count at least 3) to get $\mathcal{F}_2 = \{12, 13, 14, 15, 23, 24, 25, 34\}$.

Iteration 3: Count Word Sets of Length 3

The set $\mathcal{F}_2$ is the starting point for constructing new candidates of length 3. We look at all distinct pairs of members of $\mathcal{F}_2$, and form a new set of length 3 by taking a union of the pairs, as long as they share a common prefix, i.e., they have all elements in common except the last element. For example, if the two members are 12 and 13 then the common prefix is 1, and the resulting candidate is 123, but if the two pairs are 12 and 23 then we don't generate anything, since 12 and 23 have different prefixes. If we apply this procedure for all pairs we get the following candidate set, $\mathcal{C}_3 = \{123, 124, 125, 134, 135, 145, 234, 235, 245\}$

After generating $\mathcal{C}_3$, we apply a **pruning** step, i.e., for each candidate we check if all of its length 2 subsets are members of $\mathcal{F}_2$. If we don't find a subset in $\mathcal{F}_2$, we discard that candidate, since it cannot be frequent (note: one can easily prove that if a candidate is frequent, then all its subsets must be frequent). For example, consider the set 135. It has 3 subsets of length 2, namely 13, 15 and 35. We see that 35 doesn't belong to $\mathcal{F}_2$. Thus we remove 135 from $\mathcal{C}_3$. Similarly, we remove 145, 235, and 245. After pruning the new $\mathcal{C}_3 = \{123, 124, 125, 134, 234\}$.

We now count the occurrence of the candidates:

```
Candidates:     123     124     125     134     234
---------------------------------------------------
Count   :        4       3       4       3       3
```

We find that all of these are frequent. Thus $\mathcal{F}_3 = \{123, 124, 125, 134, 234\}$

Iteration 4: Count Word Sets of Length 4

We use the procedure described in the previous iteration to generate a new $\mathcal{C}_4$ from $\mathcal{F}_3$, to get $\mathcal{C}_4 = \{1234, 1235, 1245\}$, i.e., we consider all pairs from $\mathcal{F}_3$, and generate a set of length 4 by taking their union if they share a common prefix. For example, if the pairs are 123 and 124, then the common prefix is 12, and the result is 1234. If the pairs are 123 and 234, then there is no result since they don't share a common prefix.

Next we prune a candidate if any of its length 3 subsets is not in $\mathcal{F}_3$. For example 123, 125, 135, and 235 are the four length 3 subsets of 1235. We see that 135 doesn't appear in $\mathcal{F}_3$. Thus we discard the candidate 1235. We also discard 1245 for the same reason. The final candidate set is $\mathcal{C}_4 = \{1234\}$.

Now we count the candidates:

```
Candidates:     1234
--------------------
Count   :        3
```

Since this is frequent, we get $\mathcal{F}_4 = \{1234\}$.

Iteration 5: Count Word Sets of Length 5

Since $\mathcal{F}_4$ has only one member, we cannot generate any new candidates. At this point WebMiner stops.

Output

The final output of WebMiner consists of a list of all frequent word sets discovered in the process:
$\mathcal{F}_1 = \{1, 2, 3, 4, 5\}$
$\mathcal{F}_2 = \{12, 13, 14, 15, 23, 24, 25, 34\}$
$\mathcal{F}_3 = \{123, 124, 125, 134, 234\}$
$\mathcal{F}_4 = \{1234\}$

Data Analysis

If you look carefully, you'll find that all the frequent sets are subsets of either 1234 or 125. Using this information, we find all the pages that contain 1234. This is the page set $\{2, 3, 5\}$. Also the set of pages that contain 125 is $\{1, 3, 4, 5\}$. If we convert the identifiers back to the page names and key words we find:

```
word set ---- web pages
-------------------------------------------------
1234 ---- 2,3,5
computer_vision, computer_systems, programming_languages,
computation_theory --- RPI_2, MIT, UofR_1

125 --- 1,3,4,5
computer_vision, computer_systems, semiconductors
     --- RPI_1, MIT, NWU, UofR_2
```

We can now classify the web pages RPI_2, MIT, UofR_1 as belonging to CS departments, while the pages RPI_1, MIT, NWU, UofR_2 belong to EE departments. Note that MIT belongs to both CS and EE.

Data Structures for WebMiner

We will now describe the data structures that you must use for implementing the various steps of the WebMiner algorithm.

A word-set of length k has a vector of ints of length k for storing the different elements, and it has a count field for storing the frequency. The word-set is sorted. Note that you will never have to explicitly sort the word-set. The nature of the candidate generation procedure should guarantee that the new word-sets in each step are always sorted.

The set of frequent word-sets of length k, $\mathcal{F}_k$, should be stored as a sorted vector or linked-list of word-sets. For efficiency, store new elements using push_back(), and then sort the vector before printing.

The set of candidate word-sets of length k, $\mathcal{C}_k$ must be stored in a **HashTree** data structure, described below.

Candidate HashTree

The candidates are stored in a HashTree to facilitate fast counting. An internal node of the hash tree at depth d contains a hash table whose cells point to nodes at depth $d+1$. The size of the hash table in each internal node is denoted as $\mathcal{H}$. All the word-sets are stored in the leaves in a sorted linked-list.

To insert a word-set in $\mathcal{C}_k$, we start at the root and hash on successive items to follow the appropriate child. At depth d we hash on the d-th item in the word-set. We do this until we reach a leaf. The new word-set is then inserted in the linked list of that leaf. If the number of word-sets in that leaf exceeds a *threshold* value, that node is converted into an internal node, i.e., we create a new hash table for that node, and then rehash (on the d-th item) the list of word-sets so that they go to appropriate children. There is one exception where we do not convert a leaf node to an internal node even if the threshold is exceeded. This is the case if the depth of the hash tree would exceed the length of the word-sets (which is the same as the iteration number), i.e., we insure that the maximum depth of the tree in iteration k is k.

The hash table size of the hash tree $\mathcal{C}_k$ for each internal node should be chosen as follows:

$$\mathcal{H} = \left\lceil \frac{\mathcal{F}_{k-1} \cdot (\mathcal{F}_{k-1} - 1)}{2T} \right\rceil^{1/k}$$

where $\mathcal{H}$ (actually $\lceil \mathcal{H} \rceil$) is the hash table size, T is the leaf threshold, and $\mathcal{F}_{k-1}$ denotes the size of the set of frequent word-sets of length $k-1$ (from the previous iteration). Usually threshold T should be small. You can use the value $T = 2$.

Let's look at the hash tree that would be generated in iteration 3 in our example above. First there are 8 word-sets in $\mathcal{F}_2$. Therefore the hash table size is given as

$$H = \left\lceil [(8 \cdot 7)/(2 \cdot 2)]^{1/3} \right\rceil = \left\lceil 14^{1/3} \right\rceil = 3$$

We will use the hash function X mod 3 for each hash table in an internal node. The figure below shows how the tree grows as the candidates are added to it.

The tree starts out as the root being a leaf which will store word-sets. We insert 123, and then 124, as shown.

When we try to insert 125, we see that the threshold T=2 for the leaf capacity is exceeded, at this point we have to turn the root into an internal node with a hash table of size 3. We then have to rehash 123, 124, and 125 based on their first item (1). In this case all three have the same first item, and they end up in the same leaf list. Since the capacity of the leaf is exceeded, we again have to rehash, turning the leaf node into an internal node. But this time we hash based on the second element (2). Once again all word-sets end up in the same leaf. For the third time, we rehash based on the third element of the word-sets. For 123, the third element is 3, and it goes into cell 0; similarly, 124 and 125 end up in leaf pointed to by cell 1 and 2, respectively.

We add the next wordset, 134 at the right spot by first hashing on 1 and then on 3. Finally for 234, when we hash on 2 we hit a leaf and we put 234 there.

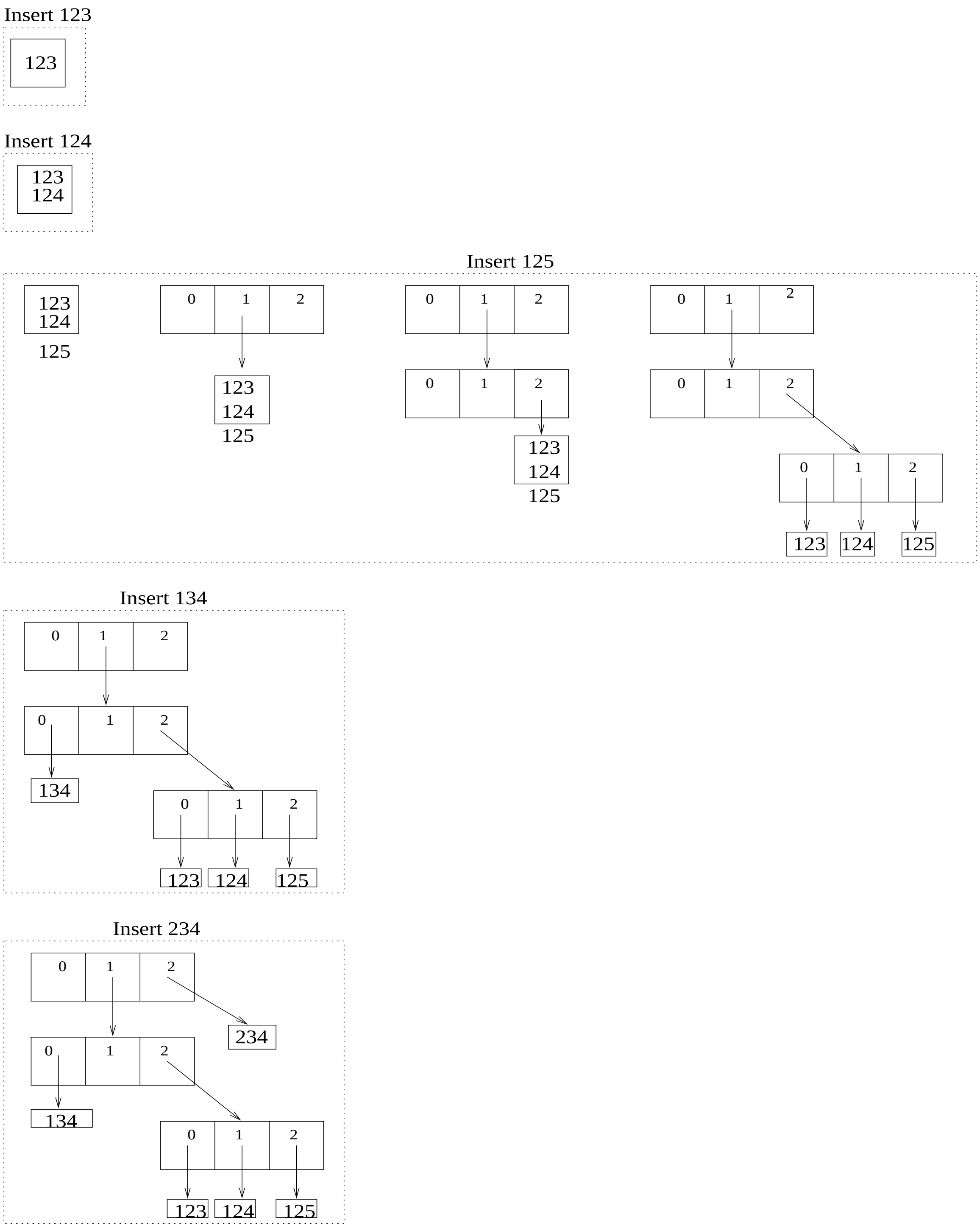

Figure 1: Hash Tree Building

WebMiner Algorithm Pseudo-Code

We will now give the pseudo-code for the different steps of the algorithm. The main WebMiner algorithm is given below:

```
MINCOUNT = ceil (min_per*NumWebPages);

F1 = getF1();  //special case for first step

for (k=2; F_{k-1} != NULL ; k++){

     /*Candidate generation */
     C_k = new HashTree;
     for each word-set in F_{k-1}, say w1{
         for each word-set in F_{k-1} after w1 (i.e. > w1), say w2{
             if (w1 and w2 share common prefix of length k-2){
                 cand = union (w1, w2);
                 if (!prune(cand, F_{k-1})) C_k.Insert(cand);
             }
         }
     }
     delete F_{k-1};

     /*Candidate Counting*/
     for each input record, say R{
         for each k length subset of R, say S{
               S' = C_k.find(S);
               if (S' != NULL) S'.count++;
         }
     }

     /*Frequent set generation*/}
     for each set, say S, in C_k
         if (S.count >= MINCOUNT) F_k.push_back(S);

    sort(F_k)
    print F_k;
    delete C_k;
}
```

Program synopsis, input and output format

The program will be run with the command-lines

```
WebMiner input_datafile min_per
```

The input file will have the following format:

```
NumWebPages
MaxKeyWords
WebPage Num_KeyWords  KeyWords
.....
```

For our running example, the input file will look like:

```
6
5
1 3 1 2 5
2 4 1 2 3 4
3 5 1 2 3 4 5
4 4 1 2 3 5
5 4 1 2 3 4
6 3 1 2 5
```

With min_per = 0.5, your output should look like (it has the word set followed by the count):

```
Length 1 (5)
1 - 6
2 - 6
3 - 4
4 - 3
5 - 4

Length 2 (8)
1 2 - 6
1 3 - 4
1 4 - 3
1 5 - 4
2 3 - 4
2 4 - 3
2 5 - 4
3 4 - 3

Length 3 (5)
1 2 3 - 4
1 2 4 - 3
1 2 5 - 4
1 3 4 - 3
2 3 4 - 3

Length 4 (1)
1 2 3 4 - 3

Total = 19
```

Notes, hints, and assumptions

- We have provided the main algorithm, a routine that generates subsets, and other class prototypes for HashTree in the directory www.rpi.edu/dept/cs/cs230/public/projects/project3.
- You will need to "rewind" the input file after you have counted the frequency of word-sets in an iteration. Do this using fd.clear(); and fd.seekg(0); where fd of type ifstream.
- Use the leaf threshold value $T = 2$. If a leaf has more than T items then it will be converted into an internal node. The only exception is if the tree is already at the maximum depth. In this case, the leaf may hold more than T word-sets.

Submission dates and guidelines

The final submission deadline is Nov. 17 at 11:59:59 pm. Projects submitted within 24 hours after this deadline will have 10 points taken from each member's grade (maximum of 100). Projects submitted within 24-48 hours after this deadline will have 20 points taken from each member's grade. Projects submitted later than this will NOT be accepted for credit.

- Each group member should submit (electronically) a brief, confidential evaluation the contributions of every individual in the group.
- Each class or function should include a comment indicating the primary author and who made what revisions.

To facilitate group interaction, a preliminary submission is required. By Tuesday, Nov 10th, at 11:59:59 pm, each group must submit a brief outline of the design of the classes for the project. This outline should indicate the class declarations, member function declaration (but not implementation) and member variables. Each group must indicate which member is responsible for which part of the project, and how the different components will be integrated.

Each submission will be briefly critiqued and returned (electronically). Students not submitting an outline or submitting an obviously thoughtless one by the indicated date will have a 10 point penalty taken from the project grade. During final implementation of the project, students should feel free to revise the design (in part based on the feedback).

You may find the following task breakdown useful in assigning work to individual team members (but feel free to modify it to suit your individual team):

1. `Hash Tree` class:
   1. Insert()
   2. Rehash()
   3. Find()
2. `Candidate Counting`:
   1. Implement pseudo-code
   2. candidate_count()
   3. find and increment candidate count
3. `Candidate Generation`:
   1. Implement pseudo-code
   2. union();
   3. prune();
4. `Frequent Set Generation`:
   1. Implement pseudo-code
   2. getF1();

   `Hash Tree Iterator` class

Grading criteria

Please refer to project 2 for grading criteria, etc.