Ben Holmes & Anthony House
ComS 311 - Project Assignment 1

Report

1. Pseudo codes for methods add, and search(Tuple t) methods from HashTable.

   Add(Tuple t)

   ```
   Instead of an array, we use another internal hash table. The internal hash table, uses a linked list for collision handling.
   Get and save hash for tuple key
   Get and save the internal hash index using the tuple key - hash
   If hashTable at index hash is null
       Create an arraylist at the position and using the position for inside hash, add to the front of the list. Increment the number of elements
   Else if the internal hash table and arraylist at the inside hash position is null, create a linked list
       Add to the first position and incrementing the number of elements
   Else
       Get the linked list by getting the hash table position, then the internal arraylist.
       Set a found var to false
       Iterate the arraylist
       Check if the tuples equal each other and increment occurrences if it's found. Set found to true and break from the loop.
       If it's not found, add it to the beginning of the arraylist
   ```

   Search(Tuple t)

   ```
   Get and save hash for tuple key
   Get and save the internal hash table using the tuple key - hash
   Set the return result to 0
   If the hash table at the hash position is null, return 0
   Else if the inside hashtable is null, return 0
   Else iterate through the internal hash table.
       If you find a matching tuple, add the occurences to the result
   Return the result
   ```

2. Derive and state asymptotic run-times of above methods. Express run-time as a function of n, and k, where n is the number of elements on the Hash Table and k is the length of the String.

Add(Tuple t)

```
int hash = hashFunc.hash(t.getKey());                                                       // constant
int insideHash = insideHashFunc.hash(t.getKey() - hash);                                    // constant
    if(hashTable.get(hash) == null)                                                         // constant
        hashTable.set(hash, new ArrayList<LinkedList<Tuple>>(Collections.nCopies(insideTableSize,  null)));  // constant
        hashTable.get(hash).set(insideHash, new LinkedList<Tuple>());                       // constant
        hashTable.get(hash).get(insideHash).addFirst(t);                                    // constant
        numOfElements++;                                                                    // constant
    else if(hashTable.get(hash).get(insideHash) == null)                                    // constant
        hashTable.get(hash).set(insideHash, new LinkedList<Tuple>());                       // constant
        hashTable.get(hash).get(insideHash).addFirst(t);                                    // constant
        numOfElements++;                                                                    // constant
    Else                                                                                    // constant
        LinkedList<Tuple> linkedTuples = hashTable.get(hash).get(insideHash);               // constant
        boolean found = false;                                                              // constant
        for(int i = 0; i < linkedTuples.size(); i++)                                        // sum 1 to n
                Tuple tuple = linkedTuples.get(i);                                          // constant
                if(t.equals(tuple))                                                         // constant
                        hashTable.get(hash).get(insideHash).get(i).increment();             // constant
                        found = true;                                                       // constant
                        Break;                                                              // constant
                if(!found)                                                                  // constant
                        hashTable.get(hash).get(insideHash).addFirst(t);                    // constant
```

This function has the following runtime. $\sum_{i=1}^{n}$ c.

$\sum_{i=1}^{n} c \;\rightarrow\; c\sum_{i=1}^{n} 1 \;\rightarrow\; cn$

Runtime total is O(n)

Search(Tuple t)

```
int hash = hashFunc.hash(t.getKey());                                                       // constant
int insideHash = insideHashFunc.hash(t.getKey() - hash);                                    // constant
int result = 0;                                                                             // constant
if(hashTable.get(hash) == null)                                                             // constant
        return 0;                                                                           // constant
else if(hashTable.get(hash).get(insideHash) == null)                                        // constant
        return 0;                                                                           // constant
Else                                                                                        // constant
        Iterator<Tuple> it = hashTable.get(hash).get(insideHash).iterator();                // constant
        while(it.hasNext())                                                                 // sum 1 to n
                Tuple tuple = (Tuple) it.next();                                            // constant
                if(tuple.equals(t))                                                         // constant
                        result += tuple.occurrences;                                        // constant
        return result;                                                                      // constant
```

This function has the following runtime. $\sum_{i=1}^{n}$ c.

$\sum_{i=1}^{n} c \;\rightarrow\; c\sum_{i=1}^{n} 1 \;\rightarrow\; cn$

Runtime total is O(n)

3. For each of the classes BruteForceSimilarity, HashStringSimilarity, and HashCodeSimilarity

- Describe the data structures used.

  Brute Force
  - ArrayList<String> (To store S, T and U)
  - ArrayList<Integer> (To Store the count of the strings in S, and T)

  Hash String
  - HashTable (To store S, T and U)
  - ArrayList<String> (To store distinct strings in s1, s2, and both)

  Hash Code
  - HashTable (To store sets S, T and U)
  - ArrayList<Integer> (To store distinct codes in s1, s2, and both)

  Hash Table
  - ArrayList<> To store the hashes
  - LinkedList<Tuple> To manage collisions

- Pseudo code for all three methods

  Brute Force

  Length of S1

  Set the initial return result to 0.00

  If length of s1 is less than the shingle length, return 0.00

  For each distinct string in s1, find how many times it occurs in the multi-set and square that. Add this to the return result

  Return the square root of the result

  Length of S2

  Set the initial return result to 0.00

  If length of s2 is less than the shingle length, return 0.00

  For each distinct string in s2, find how many times it occurs in the multi-set and square that. Add this to the return result

  Return the square root of the result

  Similarity

  Set the initial return result to 0.00

  Set the top summation to 0

  For each distinct string from S1 and S2, find how many times it occurs in each respective multi-set. Multiply the occurrences found in S1 and S2 and add it to the top summation

  *Set a variable denominator to the length of S1 * the length of S2*

  If the denominator is 0, return 0.0

  return the top summation / denominator

  Hash String

Length of S1

```
Set the initial result to 0.0
If length of S1 is less than the shingle length, return 0.0
For each distinct string in s1, create a tuple using the hash code and string. Use that tuple to find the number of occurrences in the hash table and raise it to the power of 2
Return the square root of the result
```

Length of S2

```
Set the initial result to 0.0
If length of S2 is less than the shingle length, return 0.0
For each distinct string in s2, create a tuple using the hash code and string. Use that tuple to find the number of occurrences in the hash table and raise it to the power of 2
Return the square root of the result
```

Similarity

```
Set the initial result to 0.0
Set the top summation to 0.0
For each distinct string from s1 and s2, create a tuple using the hash code and string. In the S and T hashtable, find the number of occurrences for the tuple and multiple those numbers returned together - adding it to the top summation
Set the denominator to the length of s1 times the length of s2
If the denominator is 0, return 0.0
Return the top summation / denominator
```

Hash Code

Length of S1

```
Set the initial result to 0.0
If length of S1 is less than the shingle length, return 0.0
For each integer hash in the s1 distinct hashes
    Get the array at the integer hash
    If the array size is larger than 0, get the occurrences and raise it to the power 2
    Add that to the result
Return the square root of the result
```

Length of S2

```
Set the initial result to 0.0
If length of S2 is less than the shingle length, return 0.0
For each integer hash in the s2 distinct hashes
    Get the array at the integer hash
    If the array size is larger than 0, get the occurrences and raise it to the power 2
    Add that to the result
Return the square root of the result
```

Similarity

```
Set the initial result to 0.0
Set the top summation to 0
For each integer hash in the s1 and s2 distinct hashes
        Create an array with all of the tuples from S with the integer hash
        Create an array with all of the tuples from T with the integer hash
        If both arrays are larger than 0, get the first item in both arrays
        Multiply the tuple occurrences together and add it to the top summation
Set the denominator to the length of s1 times the length of s2
If the denominator is 0, return 0.0
Return the top summation / denominator
```

4. Derive and state asymptotic run-time of all three methods. Express run-times as functions of n, m and k. Where n and m are lengths of the strings and k is the shingle length parameter.

Brute Force

Length of S1

```
public static int getS1Count(String val)
        if(!s1DistinctStrings.contains(val)) return 0;                //O(n)
        int index = s1DistinctStrings.indexOf(val);                           //O(n)
                return S.get(index);                                  //O(1)
```

Total = 2O(n) or O(n)

```
public float lengthOfS1()
        float result = 0.0f;                                                  //O(1)
        if(s1.length() < length) return 0.0f;                                 //O(1)
        for(String string : s1DistinctStrings)                                //O(n)
                result += power(getS1Count(string), 2);                       //O(2n) for getS1Count()
        result = (float) Math.sqrt(result);                                   //O(1)
        return  result;                                                       //O(1)
```

Total O(n^2)

Length of S2

```
public static int getS2Count(String val)
        if(!s2DistinctStrings.contains(val)) return 0;                          //O(m)
        int index = s2DistinctStrings.indexOf(val);                             //O(m)
        return T.get(index);                                                    //O(1)
                                                                                Total 2O(m)

public float lengthOfS2()
        float result = 0.0f;
        if(s2.length() < length) return 0.0f;                                   //O(1)
        for(String string : s2DistinctStrings) {                                //O(m)
                result += power(getS2Count(string), 2);                 //O(m)
        result = (float) Math.sqrt(result);                                     //O(1)
        return result;                                                          // O(1)
```

Total O(m^2)

Similarity

```
public float similarity()
        float result = 0.0f;                                                    //O(1)
        long topSummation = 0;                                                  //O(1)
        for(String string : distinctStrings)                                    //O(n+m)
                topSummation += (getS1Count(string) * getS2Count(string));      //O(2n + 2m)
        float denominator = (this.lengthOfS1() * this.lengthOfS2());            //O(n^2+m^2)
        if(denominator == 0) return 0.0f;                                       //O(1)
        result = topSummation/denominator;                                      //O(1)
        return result;                                                          //O(1)
```

Total O(n^2+m^2) or O(n^2)

Hash String

Length of S1

```
float result = 0.0f;                                                // constant
if(s1.length() < length)                                            // constant
        return 0.0f;                                                // constant
for(String string : s1DistinctStrings)                              // sum 1 to n
        Tuple tuple = new Tuple(computeHash(string), string);       // constant
        result += power(S.search(tuple), 2);                        // log(2) = O(1)
result = (float) Math.sqrt(result);                                 // constant
return result;                                                      // constant
```

This function has the following runtime. $\sum_{i=1}^{n} c$.

$\sum_{i=1}^{n} c \rightarrow c\sum_{i=1}^{n} 1 \rightarrow cn$

Runtime total is O(n)

Length of S2

```
float result = 0.0f;                                                // constant
if(s2.length() < length)                                            // constant
        return 0.0f;                                                // constant
for(String string : s2DistinctStrings)                              // sum 1 to m
        Tuple tuple = new Tuple(computeHash(string), string);       // constant
        result += power(T.search(tuple), 2);                        // log (2) = O(1)
result = (float) Math.sqrt(result);                                 // constant
return result;                                                      // constant
```

This function has the following runtime. $\sum_{i=1}^{n} c$.

$\sum_{i=1}^{n} c \rightarrow c\sum_{i=1}^{n} 1 \rightarrow cn$

Runtime total is O(n)

Similarity

```
float result = 0.0f;                                                // constant
long topSummation = 0;                                              // constant
for(String string : distinctStrings)                                // sum 1 to n + m
        Tuple tuple = new Tuple(computeHash(string), string);       // constant
        topSummation += (S.search(tuple) * T.search(tuple));        // constant
float denominator = (this.lengthOfS1() * this.lengthOfS2());        // n (above)
if(denominator == 0)                                                // constant
        return 0.0f;                                                // constant
result = topSummation/denominator;                                  // constant
return result;                                                      // constant
```

This function has the following runtime. $\sum_{i=1}^{n+m} c + \sum_{i=1}^{n} c$

$\sum_{i=1}^{n+m} c + \sum_{i=1}^{n} c \rightarrow c(\sum_{i=1}^{n+m} 1 + \sum_{i=1}^{n} 1) \rightarrow c(n+m+ \sum_{i=1}^{n} 1) \rightarrow c(n+m + n) \rightarrow c(2n+m) \rightarrow 2nc+mc$

Runtime total is O(n)

Hash Code

Length of S1

```
float result = 0.0f;                                            // constant
if(s1.length() < length)                                        // constant
    return 0.0f;                                                // constant
for(Integer hash : s1DistinctHashes)                            // sum 1 to n
    ArrayList<Tuple> list = S.search(hash);                     // constant
    if(list.size() > 0)                                         // constant
        for(Tuple tuple : list)                           // O(1) | < 2 items
            result += power(tuple.occurrences, 2);              // log (2) = O(1)
return (float) Math.sqrt(result);                         // constant
```

This function has the following runtime. $\sum_{i=1}^{n} c + 1 + 1$ or just $\sum_{i=1}^{n} c$.

$$\sum_{i=1}^{n} c \rightarrow c\sum_{i=1}^{n} 1 \rightarrow cn$$

Runtime total is O(n)

Length of S2

```
float result = 0.0f;                                            // constant
if(s2.length() < length)                                        // constant
    return 0.0f;                                                // constant
for(Integer hash : s2DistinctHashes)                            // sum 1 to m
    ArrayList<Tuple> list = S.search(hash);                     // constant
    if(list.size() > 0)                                         // constant
        for(Tuple tuple : list)                           // O(1) | < 2 items
            result += power(tuple.occurrences, 2);              // log (2) = O(1)
return (float) Math.sqrt(result);                         // constant
```

This function has the following runtime. $\sum_{i=1}^{m} c + 1 + 1$ or just $\sum_{i=1}^{m} c$.

$$\sum_{i=1}^{m} c \rightarrow c\sum_{i=1}^{m} 1 \rightarrow cm$$

Runtime total is O(n)

Similarity

```
float result = 0.0f;                                            // constant
float topSummation = 0.0f;                                      // constant
for(Integer hash : distinctHashes)                              // sum 1 to n + m
    ArrayList<Tuple> list1 = S.search(hash);                    // constant
    ArrayList<Tuple> list2 = T.search(hash);                    // constant
    if(list1.size() > 0 && list2.size()>0)                      // constant
        int s1Occurrences = 0;                                  // constant
        int s2Occurrences = 0;                                  // constant
        for(Tuple tuple : list1)                                // O(1) | < 2 items
            s1Occurrences += tuple.occurrences;                 // constant
        for(Tuple tuple : list2)                                // O(1) | < 2 items
            s2Occurrences += tuple.occurrences;                 // constant
        topSummation += (s1Occurrences * s2Occurrences)         // constant
float denominator = (this.lengthOfS1() * this.lengthOfS2());    // constant
if(denominator == 0) return 0.0f;                               // constant
result = topSummation/denominator;                              // constant
return result;                                                  // constant
```

This function has the following runtime. $\sum_{i=1}^{n+m}$ c + 1 + 1 + 1 + 1 or just $\sum_{i=1}^{n+m}$ c.

$\sum_{i=1}^{n+m} c \rightarrow c \sum_{i=1}^{n+m} 1 \rightarrow c(n+m) \rightarrow nc+mc$

Runtime total is O(n)

5. You are provided two test files. Convert the files into strings and run the method similarity from all three classes. Use 8 as shingle length. Report the similarities returned and the run-times.

   - Brute Force
     - 0.41189092
     - Total time (seconds) = 126.39
   - Hash String
     - 0.41189092
     - Total time (seconds) = 0.241
   - Hash Code
     - 0.4119063
     - Total time (seconds) = 0.364

6. Compare all three run times. Which one is smallest? Which one is largest? Does one run time equal (or very close to) another run time? Explain why one run-time equals (or very close) /smaller/larger than the other run times.

   BruteForce > HashCode > Hash String
   BruteForce = O(n^2)
   HashCode = O(n)
   HashString = O(n)

   The Hash Code function should run with the least time, but we didn't use Roll-Over hashing due to the overflow problem in Java. Instead we just computed the hash code over each substring. If we would have been allowed to store a Long value instead of an int, we could have managed the hashes consistently. The Brute Force function takes the longest. The Brute Force is the slowest because we are not using a hash table. We must search through the array list each time we add something new to make sure it isn't there. The Hash String was quicker than Brute Force because we are hashing specific elements into specific spots. This changed it to O (1). Same with Hash String.

7. Do all three methods return the same value for similarity? If not, explain the reason.

   Brute Force and Hash String both run with the same similarity. Hash code does not run with the same similarity however. The Hash Code has a different similarity compared to the other two because sometimes, two hash codes get hashed into the same spot, even though they don’t represent the same string. Therefore, there’s no distinction between them, and that causes the values returned by lengthS1() and lengthS2() to differ, because they will be counting two different objects as the same occurrence.