**Theory Component for Grading Student Participation in Group Chat**

A logo of a bird and a shield

Description automatically generated

University of Technology, Jamaica

School of Computing and Information Technology

Analysis of Algorithms (CIT3003)

Mr. David White, Mr. Arnett Campbell

Christoph Banton - 2100760, Chase Walfall - 2102044, Tashaw Turner - 2103380,

Enoch Omoregie - 2111440

**Table of Contents**

[Summary of the Issue 3](#_heading=h.gjdgxs)

[Problem Category 3](#_heading=h.30j0zll)

[Class Complexity 3](#_heading=h.1fob9te)

[Computational Means and Design Techniques 4](#_heading=h.3znysh7)

[Algorithmic Design and Description of Working Ability 6](#_heading=h.2et92p0)

[Category of the Algorithm 7](#_heading=h.tyjcwt)

[Proof of Correctness 8](#_heading=h.3dy6vkm)

[Analysis of Algorithms 12](#_heading=h.1t3h5sf)

[Pseudocode 12](#_heading=h.4d34og8)

[Overall Complexity 16](#_heading=h.2s8eyo1)

# **Summary of the Issue**

The issue faced by the teacher is that it is tedious for the tutor to teach and efficiently keep up with tracking the participation of each student. A demanding task such as teaching is already a tasking operation, so to go through the session and document and grade each person in terms of participation afterwards would be very tiresome and is susceptible to mistakes. This is important to note as being able to track student participation would be a valuable metric for the teacher to have. Using a way that provides ease of mind and efficiency complements the quality of the marks given, which not only helps the teacher to better spend his/her time and energy, but also better rewards students for their contribution to making a better learning environment for each other.

# **Problem Category**

The problem belongs to the String processing problem category, as the task involves analyzing chat logs, which are essentially strings of text, to determine and grade student participation, which requires various string processing techniques and algorithms. The problem was addressed using a Python-based algorithm.

# **Class Complexity**

The class complexity of this is class P (Polynomial time) as it is combating decision problems using simple operations such as iterating over entries, condition checking, performing similarity calculations which are algorithm solutions completed in polynomial time. These operations entail sifting through components multiple times and are quite straightforward in their implementation of such while limiting the number of nested loops used. Knowing this we can imply that due to the nature of the algorithm, the time complexity can be influenced by the size of the chat log, the number of known questions, and the length of messages. Therefore, the algorithm is moderately fast for small n, but not for large n, n being the number of operations to be called out which is affected by the input size.

# **Computational Means and Design Techniques**

**Python Language and Key Libraries:**

Python is the core programming language used, known for its ease of readability and wide range of libraries.

Libraries like Tkinter, re, chardet, nltk, and scikit-learn are utilized for various functionalities ranging from GUI creation, text encoding detection, natural language processing, to machine learning for text analysis.

**User Interface and Input Handling:**

The GUI, built using Tkinter, allows for interactive user input to collect crucial information like the host’s name for identifying specific entries in the chat log.

Event-driven programming in Tkinter makes the application responsive to user actions, like button clicks to load files and process data.

**Regular Expressions for Parsing Chat Logs:**

Regular expressions are employed to parse chat logs effectively, matching patterns to extract user messages and timestamps. This technique ensures precise and efficient data extraction from chat logs.

**Data Structures for Efficient Data Handling:**

Dictionaries are used to store and manage various datasets like known questions and answers, keyword weights, and user scores. This choice of data structure enables fast lookups and streamlined data processing.

**Optimized Text Processing:**

Text processing includes converting text to lowercase and stripping whitespaces, making comparisons case-insensitive and more consistent.

Private messages and disallowed phrases are skipped to optimize the processing and focus on relevant content.

**Natural Language Processing and Machine Learning:**

NLTK is used for tokenizing messages and checking for real words and context, adding depth to the analysis of user messages.

The scikit-learn library’s TfidfVectorizer function is used to convert text into a numerical vector representation, facilitating the analysis of text data.

Cosine similarity is computed between user responses and correct answers to quantitatively assess the similarity, aiding in the evaluation of user contributions in the chat.

**Modular and Structured Script Design:**

The script is structured into various functions, each dedicated to a specific task like validating chat entries, checking messages, and computing scores. This modular approach enhances code maintainability and readability.

To create a robust system for chat log analysis, the implementation combines traditional programming techniques with advanced computational methods. It combines Python's and its libraries' strengths, the efficiency of data structures like dictionaries, the precision of regular expressions, and the power of NLP and machine learning to provide a comprehensive tool for analyzing and scoring user interactions in a chat environment. This method combination demonstrates a thoughtful and effective approach to problem solving.

# **Algorithmic Design and Description of Working Ability**

From a top-level view the algorithm questions and to analyze a chat log, grade participants based on their responses to known questions, and calculate scores. From a more in-depth perspective, first the parse\_chat\_log function reads through a chat log file line by line. It uses a regular expression pattern to match entries in the format ‘HH:MM:SS From User: Message.’. The script checks for empty lines, private messages, and disallowed phrases and excludes them from the parsed data. The data is then stored in a list of dictionaries representing the different chat entries. The grade\_chat\_log function takes the parsed chat log entries, host name, known question and answers, keywords and weights, disallowed phrases, and trigger phrases as input. It then iterates through the entries checking if a trigger phrase is present in any of them. If it finds the host has sent a message with a trigger phrase it will mark it as a question and use it as a reference in dealing with participant responses. For each participant’s responses, it checks for disallowed phrases, calculates a score based on how close their answer is using keyword matching and cosine similarity. It then update the user’s score in a dictionary. The cosine\_similarity\_score function then uses the scikit-learn library to calculate the cosine similarity between two text strings. It first converts the text to TF-IDF (Term Frequency-Inverse Document Frequency) vectors and then computes the cosine similarity between these vectors.

# **Analysis of Algorithms**

## **Pseudocode**

FUNCTION grade\_chat\_log

Input: content (list of chat entries), host\_name (string), known\_questions (list),

keywords (list), disallowed\_phrases (list), triggers (list)

Output: scores (dictionary)

Initialize scores as an empty dictionary

Set trigger\_present to False

Set question to an empty string

For each entry in content:

Get user from entry

Get message from entry and convert it to lowercase

If user is host\_name:

If trigger\_present is True:

Check if any trigger is in the message

If trigger is present:

Set trigger\_present to False

Continue to the next iteration

Check if any trigger is in the message

Set trigger\_present based on the above check

If trigger\_present is True:

Set question to the current message

Continue to the next iteration

If trigger\_present is True:

Check if message contains any disallowed phrases

If yes, continue to the next iteration

Calculate message\_score by matching keywords in message with known\_questions, keywords, and the current question

If user is not in scores:

Initialize user's score in scores to 1

Add message\_score to user's score in scores

Return scores

FUNCTION match\_keywords(message, known\_questions, keywords, graded\_question)

Initialize score to 0.0

FOR EACH question-answer pair in known\_questions

IF question is part of the graded\_question

Calculate cosine similarity score between message and answer

Double the similarity score

Add scores for each keyword present in the message

IF score is less than or equal to 0

IF message has context AND contains real words

Set score to 1

RETURN score

RETURN score

FUNCTION check\_disallowed\_phrases(message, disallowed\_phrases)

Convert message to lowercase

FOR EACH phrase in disallowed\_phrases

IF phrase is in message

IF message exactly matches the phrase

RETURN True

RETURN False

FUNCTION has\_context(message)

Tokenize the message

IF number of tokens is greater than 3

Perform part-of-speech tagging on tokens

Collect unique parts of speech tags

IF number of unique POS tags is greater than 3

RETURN True

RETURN False

FUNCTION is\_real\_word(message)

Tokenize the message

FOR EACH token in tokens

IF token is in the predefined word list

RETURN True

RETURN False

FUNCTION cosine\_similarity\_score(user\_answer, correct\_answer)

Initialize a TF-IDF Vectorizer

Transform user\_answer and correct\_answer into TF-IDF matrix

Calculate cosine similarity between the two vectors

RETURN the cosine similarity score

# **Category of the Algorithm**

This algorithm belongs to the Divide and Conquer category. We can see this in how the algorithm works, as the chat log is first parsed through and separated into entries using regular expression matching combined with checking and elimination for private messages and disallowed phrases. Each entry is then dealt with on that level in which the answer is vectorized and then ran into a cosine similarity function in which we compare the vectorized answer given by the student with the actual answer vectorized, using a mathematical function. With all the entries then calculated on their own, the values are then brought back together and matched to the respective user to represent the participation score gathered from the entire chat log.

# **Proof of Correctness**

1. **For The grade\_chat\_log Function:**

**Loop Invariant:**

The loop invariant for the grade\_chat\_log function is that at the start of each iteration (for any k where 1 ≤ k ≤ N, with N being the total number of entries in content), the variable trigger\_present accurately represents whether a trigger has been detected in any of the entries processed up to that point in the loop.

**Initialization:**

Before the loop starts (at k = 0), trigger\_present is set to False. This establishes that no triggers are active before processing any chat log entries, which is the base case for our loop invariant.

**Maintenance:**

If the loop invariant holds before an iteration at k (where 1 ≤ k < N), then trigger\_present will be updated based on the content of the current message. If a trigger is found, trigger\_present remains True, and scores are updated accordingly. The invariant is maintained because if trigger\_present was True at step k, it will either remain True or be set to False at step k + 1, depending on whether the end trigger is detected.

**Termination:**

When the loop terminates (at k = N), the hypothesis (loop invariant) that trigger\_present reflects the presence of a trigger in the chat log entries is used to update the scores dictionary. This final state of scores indicates the correctness of the algorithm, as it represents the accumulated scores based on the rules defined within the loop, including the handling of triggers.

* Loop Invariant at Start (k = 1):
  + For all k in the range 1 to N, trigger\_present is False.
  + This is the state before the first entry is processed.
* Maintenance (for each step k to k + 1):
  + Given that the invariant holds at the beginning of the iteration k, after processing the k-th entry, trigger\_present will correctly reflect the state of triggers.
  + If trigger\_present was True and no end trigger is found in the k-th entry, it remains True.
  + If an end trigger is found, trigger\_present is set to False.
* Termination (at k = N):
  + After the final iteration, the loop invariant ensures that trigger\_present will accurately reflect the state of the last entry.
  + The scores dictionary is updated according to the triggers processed, which is the desired outcome of the algorithm.

1. **Match\_keywords Function:**

**Loop Invariant:**

At the start of each iteration for any k (where 1 ≤ k ≤ M, with M being the total number of keywords), the score reflects the sum of keyword matches found in the message up to that point.

**Initialization:**

Before the loop starts (at k = 0), score is set to 0.0. This indicates that no keywords have been matched before the loop begins.

**Maintenance:**

If the loop invariant is true before iteration k, then after checking the k-th keyword, score is updated to include the match score for that keyword if present. The loop invariant is maintained because the score continues to accurately represent the total match score for all keywords checked up to iteration k.

**Termination:**

When the loop terminates (at k = M), the final value of score reflects the total match score for all keywords in the message, which implies the algorithm's correctness in determining the relevance of the message to the keywords.

1. **Check\_disallowed\_phrases Function:**

**Loop Invariant:**

At the start of each iteration for any i (where 1 ≤ i ≤ D, with D being the total number of disallowed phrases), the Boolean return value indicates whether any of the checked disallowed phrases are present in the message up to that point.

**Initialization:**

Before the loop starts (at i = 0), the return value is set to False, assuming no disallowed phrases have been found.

**Maintenance:**

If the loop invariant is true before iteration i, and if the i-th disallowed phrase is not found in the message, the return value remains False. If the i-th phrase is found, the return value is set to True, and the loop may terminate early. The loop invariant is maintained because the return value correctly indicates the presence of any disallowed phrase found up to iteration i.

**Termination:**

When the loop terminates (at i = D), the return value indicates whether any disallowed phrase is present in the message, which shows the correctness of the function in filtering out messages with disallowed content.

1. **Cosine\_similarity\_score Function:**

Since cosine\_similarity\_score method does not involve an iterative process over a loop but rather a single computation step, a loop invariant analysis does not apply in the traditional sense. The correctness of this function would be proven by demonstrating that the mathematical operations correctly calculate the cosine similarity according to the definition of the cosine similarity measure between two vectors. This would involve validating the linear algebra operations and ensuring that the function returns the correct scalar value representing the cosine of the angle between the two vectors in a multidimensional space.

1. **Grade\_chat\_log Function:**

Looping Through Chat Entries:

The outer loop runs once for each entry in "content". If `n`is the number of entries, this is O(n). Making it a linear operation

String Operations:

Lowercasing and extracting strings are (m) operations, where 'm' is the length of the string.

Trigger Presence Check:

This involves a linear search through triggers for each message, and its complexity depends on the number of triggers "t" and the message length 'm'. This can be approximated as O(t x m).

Disallowed Phrases Check:

Similar to trigger check, this is (d x m), where `d` is the

number of disallowed phrases.

Score Calculation:

This is the most complex part, as it involves several nested operations. The complexity here depends on the implementation of `match\_keywords" and it’s called functions.

1. **Match\_keywords Function:**

Loop Through Known Questions

This is O(q), where `q" is the number of known questions.

String Operations and Cosine Similarity Calculation

The cosine similarity calculation

This involves vectorization of the text and computation of the similarity score. This is typically **O(p),** where **“p”** is the number of unique words in the text. However, the exact complexity can vary based on the implementation of "TfidfVectorizer" and "cosine\_similarity".

**4. Check disallowed\_phrases" Function:**

Loop Through Disallowed Phrases:

This is O(d x m).

5. **Has\_context and is\_real\_word Functions**:

Tokenization and POS Tagging:

These operations are generally linear with respect to the

number of tokens, but POS tagging can sometimes be more complex Assume a complexity of

O(m) for simplicity.

6. **Cosine similarity score Function**:

-TF-IDF Vectorization and Cosine Similarity Calculation:

This can be complex, potentially O(p).

## **Overall Complexity**

Considering these aspects, the overall complexity of "grade\_chat\_log" is dominated by the complexity of "match\_keywords", particularly the cosine similarity calculation. The presence of nested loops and multiple string operations also contributes to the complexity.

If n is the number of chat entries, the average length of messages, the number of triggers,

the number of disallowed phrases, and p the number of unique words, a rough estimate of the

complexity would be O(n × (txm+d × m+q × p + m)).

**1. Components Analysis:**

• The terms t× m, d × m, and m are linear with respect to each variable. When multiplied by **n**, they become quadratic components, like **n × t× m and n x d x m.**

• The term **qx p** is linear since it's the product of two variables. When multiplied by **n**, it

becomes a quadratic term n × q × p.

**2. Overall Complexity:**

• With the inclusion of n × q × p, the overall complexity remains quadratic. This is because all components, when multiplied by n, result in quadratic terms.

In basic terms, the expression O(n × (t × m+d×m+q×p+m)) can be considered quadratic in nature. Each term in the expression, when scaled with the number of entries n, results in a quadratic relationship.