Q 1

In Lesson 1, an algorithm's complexity is measured based on the input size instead of the input values.

Given: A number n and two primes p and q.

Question: Determine whether $n = p \times q$.

The input size for the algorithm that checks whether $n = p \times q$. The inputs, which are the integers n, p, and q can be represented in binary form, and the number of bits required to represent an integer x in binary is approximately $\log_2(x)$. Therefore, the size of each input is:

$$\text{Size of } n = \log_2(n), \quad \text{Size of } p = \log_2(p), \quad \text{Size of } q = \log_2(q)$$

The total input size S is the sum of the sizes of all inputs:

$$S = \log_2(n) + \log_2(p) + \log_2(q)$$

Given the relationship $n = p \times q$, we can express $\log_2(n)$ as:

$$\log_2(n) = \log_2(p \times q) = \log_2(p) + \log_2(q)$$

Substituting this back into the expression for S:

$$S = (\log_2(p)+\log_2(q))+\log_2(p)+\log_2(q) = 2\log_2(p)+2\log_2(q) = 2(\log_2(p)+\log_2(q))$$

In Big O notation, constant factors are disregarded. Therefore, the input size simplifies to:

$$S = O(\log_2(p) + \log_2(q)) = O(\log_2(n))$$

The input size for the algorithm is:

$$O(\log_2(n)) \approx O(\log n)$$

Q2

Prove that the Median of Medians selection algorithm operates in linear time, $T(n) = O(n)$, when using groups of size 7.

Let $T(n)$ denote the running time of the Median of Medians algorithm on an input of size n.

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{3n}{7}\right) + O(n)$$

The algorithm starts off by dividing the input array into groups of 7 elements each, sorting each group, and selecting the median from each group. This results in $\frac{n}{7}$ medians. The time complexity for grouping and finding the medians is $O(n)$

because there are $\frac{n}{7}$ groups and sorting each group of 7 elements takes constant time. Thus, the total time for this step is:

$$\frac{n}{7} \times O(1) = O(n)$$

Once the medians are identified, the algorithm makes a recursive call to find the median of the medians:

$$\text{Recursive Call} = T\left(\frac{n}{7}\right)$$

This pivot ensures that the array can be partitioned such that at least $\frac{3n}{7}$ elements are eliminated from further consideration. Partitioning the array around the pivot takes linear time:

$$\text{Partitioning Time} = O(n)$$

After partitioning, the algorithm goes into the recursion on at most $\frac{3n}{7}$ elements.

The following is derived: consider that each group of 7 elements has a median (the 4th element after sorting). There are $\frac{n}{7}$ such medians. Selecting the median of these medians as the pivot guarantees that at least half of the medians are less than or equal to the pivot. Consequently, at least $\frac{n}{14}$ groups have medians $\leq$ pivot. In each of these $\frac{n}{14}$ groups, there are at least 4 elements $\leq$ pivot (the median and three elements less than it). Therefore, the number of elements $\leq$ pivot is at least:

$$\frac{n}{14} \times 4 = \frac{4n}{14} = \frac{2n}{7}$$

Similarly, at least $\frac{2n}{7}$ elements are $\geq$ pivot. This means that a total of:

$$\frac{2n}{7} + \frac{2n}{7} = \frac{4n}{7}$$

elements are eliminated from consideration, leaving at most:

$$n - \frac{4n}{7} = \frac{3n}{7}$$

elements for the next recursive call. This derivation confirms that the recursive call operates on at most $\frac{3n}{7}$ elements.

$$\text{Recursive Call} = T\left(\frac{3n}{7}\right)$$

Now, we use mathematical induction to prove that $T(n) = O(n)$.

Base Case: For $n \leq 7$, the algorithm can directly find the median without further recursion. Thus:

$$T(n) = O(1) \leq d \times n$$

for a constant d ¿ 0.

Inductive Hypothesis: Assume that for all integers $m < n$, the running time satisfies:

$$T(m) \leq d \times m$$

where d is a positive constant.

Inductive Step: We aim to show that:

$$T(n) \leq d \times n$$

Starting with the recurrence relation:

$$T(n) = T\left(\frac{n}{7}\right) + T\left(\frac{3n}{7}\right) + c \times n$$

where c is a constant representing the linear-time operations.

Applying the inductive hypothesis:

$$T\left(\frac{n}{7}\right) \leq d \times \frac{n}{7}$$

$$T\left(\frac{3n}{7}\right) \leq d \times \frac{3n}{7}$$

Substituting these into the recurrence:

$$T(n) \leq d \times \frac{n}{7} + d \times \frac{3n}{7} + c \times n = \left(\frac{4d}{7} + c\right) \times n$$

To satisfy $T(n) \leq d \times n$, we require:

$$\frac{4d}{7} + c \leq d$$

Solving for d:

$$\frac{4d}{7} + c \leq d \implies c \leq d - \frac{4d}{7} = \frac{3d}{7}$$

$$c \leq \frac{3d}{7} \implies d \geq \frac{7c}{3}$$

Thus, choosing:

$$d = \frac{7c}{3}$$

satisfies the inequality:

$$\frac{4d}{7} + c = \frac{4 \times \frac{7c}{3}}{7} + c = \frac{28c}{21} + c = \frac{4c}{3} + c = \frac{7c}{3} = d$$

Therefore:

$$T(n) \leq d \times n$$

for all n, where $d = \frac{7c}{3}$. Hence, the Median of Medians algorithm with group size 7 operates in linear time:

$$T(n) = O(n)$$

Q3

You are given an $n \times n$ grid populated with n^2 bugs. Each bug occupies a unique grid cell, ensuring that no two bugs share the same position (i.e., each bug is at least 1 unit apart from any other bug). Your task is to design an efficient algorithm to identify the closest pair of bugs on the grid. The algorithm should operate within a time complexity of $O(n^2)$.

Taking the hint from the question, we can modify the standard Closest Pair algorithm. The modification involves leveraging the grid's structured layout to limit comparisons exclusively to a bug's immediate neighbors. By doing so, we eliminate redundant distance calculations inherent in the standard approach, where each pair of points might be compared multiple times. This type of comparison strategy allows each pair of bugs to be evaluated only once, thereby reducing the total number of necessary operations from $O(n^2 \log n)$ to $O(n^2)$.

Step 1: Grid Initialization

We start by representing the $n \times n$ grid as a 2D matrix. Each cell in the matrix corresponds to a grid position. We initialize all cells to 'None' to indicate the absence of bugs.

Initialize a grid G of size $(n+1) \times (n+1)$ with all entries set to 'None'.

Step 2: Mapping Bugs to the Grid

Next, we iterate through the list of n^2 bugs and place each bug in its corresponding cell within the grid based on its (x, y) coordinates.

For each bug b in the list of bugs:

- Let (x, y) be the coordinates of b. - If $1 \leq x \leq n$ and $1 \leq y \leq n$, then set $G[x][y] = b$. - Else, raise an error indicating invalid coordinates.

Step 3: Defining Selective Neighbor Directions

To avoid redundant comparisons inherent in the standard Closest Pair algorithm, we define a set of selective neighbor directions. For each bug, we will only compare it with bugs in the following four directions:

1. Right Neighbor: $(0, +1)$ 2. Bottom Neighbor: $(+1, 0)$ 3. Bottom-Right Diagonal Neighbor: $(+1, +1)$ 4. Bottom-Left Diagonal Neighbor: $(+1, -1)$

Step 4: Iterating Through the Grid and Comparing Neighbors

We initialize variables to keep track of the minimum squared distance found and the corresponding closest pair of bugs.

Set min_distance_sq $= \infty$. Set closest_pair $=$ (None, None).

We then traverse each cell in the grid. For each bug found, we compare it only with its selective neighbors as defined in Step 3. For x from 1 to n:

- For y from 1 to n:
  - If $G[x][y] \neq$ None:
    - For each direction (dx, dy) in the list of directions:
      - Let $nx = x + dx$ and $ny = y + dy$.
      - If $1 \leq nx \leq n$ and $1 \leq ny \leq n$:
      - If $G[nx][ny] \neq$ None:

· Calculate squared distance:

$$\text{distance\_sq} = (nx - x)^2 + (ny - y)^2$$

· If distance_sq < min_distance_sq:
· Set min_distance_sq = distance_sq.
· Set closest_pair = $(G[x][y], G[nx][ny])$.
· If min_distance_sq = 1, terminate the search early as no closer pair exists.

Step 5: Finalizing the Closest Pair

After completing the traversal and comparisons, we compute the actual Euclidean distance of the closest pair found by taking the square root of the minimum squared distance.

- If closest_pair $\neq$ (None, None):
  - Set min_distance = $\sqrt{\text{min\_distance\_sq}}$.
  - Return closest_pair and min_distance.
- Else, return None (No pair found).

Time Complexity Analysis

The algorithm operates in two main parts: grid initialization and the neighbor comparison.

Part 1: Creating the grid takes $O(n^2)$ time since there are n^2 cells. Mapping n^2 bugs to the grid also takes $O(n^2)$ time. Total Time for Phase 1: $O(n^2) + O(n^2) = O(n^2)$.

Part 2: Each of the n^2 bugs performs a constant number of comparisons (four in this case). Therefore, the total number of operations in this phase is $4n^2$. Total Time for Phase 2: $O(n^2)$.

Overall Time Complexity: $O(n^2) + O(n^2) = O(n^2)$.

1 Introduction

Determining the similarity between two C programs requires a comprehensive approach. This would involve more than textual or syntactical comparison. This paper introduces an algorithm designed to assess program similarity across three key dimensions: *structural similarity*, *functional similarity*, and *content similarity*. These dimensions capture different characteristics of the programs:

- **Structural similarity**: Evaluates the organization and arrangement of code constructs such as loops, conditionals, and function calls.
- **Functional similarity**: Assesses the behavior and output of the programs when executed with the same inputs.
- **Content similarity**: Analyzes the distribution of tokens such as keywords, operators, and identifiers.

2 Integration and Ensemble Approach

The evaluation of program similarity requires a balanced integration of multiple dimensions. Each metric captures distinct characteristics of the programs, which are necessary for comprehensive analysis. Using an ensemble approach allows us to leverage the strengths of each metric while mitigating the weaknesses inherent in any one aspect.

The overall similarity score S_{total} is calculated as a weighted sum of the three similarity scores:

$$S_{\text{total}} = w_1 \cdot S_{\text{structural}} + w_2 \cdot S_{\text{functional}} + w_3 \cdot S_{\text{content}}$$

where $w_1 + w_2 + w_3 = 1$. The choice of weights w_1, w_2, and w_3 reflects the relative importance of each metric depending on the context of the evaluation.

2.1 Min-Max Normalization

To ensure that all similarity metrics are comparable and scaled consistently, we can apply min-max normalization to each score. This normalization process maps each similarity score to the range [0, 1], ensuring uniform scaling across metrics.

2.2 Weight Selection and the Importance of the Ensemble

The ensemble approach allows the algorithm to be adaptable, enabling it to prioritize different aspects of similarity based on the specific application. For example:

- In plagiarism detection, structural similarity may take precedence because the syntactic organization of code tends to persist even when variables

and comments are altered. Here, we may assign a higher weight w_1 to structural similarity to better capture such relationships.

- For software verification or correctness evaluation, functional similarity is more critical. We may assign a higher weight w_2 to emphasize behavior and outputs, as functionally identical programs may differ in structure or content without altering their behavior.

- Content similarity becomes useful in cases where coding style or token usage matters, such as assessing code complexity or consistency with coding standards. In such contexts, w_3 can be increased to emphasize token distribution and information content.

The ability to adjust weights gives this algorithm flexibility, ensuring that it is applicable across a wide range of tasks such as plagiarism detection, software verification, and style analysis.

3 Methodology

The algorithm assesses the similarity between two C programs by independently computing the structural, functional, and content similarity, and then combining these scores into a single metric.

3.1 Structural Similarity: AST and Tree-Edit Distance

The structural similarity metric is particularly well-suited for comparing the organization and control flow of programs. By generating an Abstract Syntax Tree (AST) for each program, we create a hierarchical representation of its syntactic structure. This structure abstracts away surface-level details such as variable names, making it easier to focus on the logical organization of the code.

This is ideal as AST-based approaches are highly effective in scenarios where the layout and control flow of the program are essential, such as plagiarism detection or code optimization analysis. Even if two programs have different variable names or comments, their ASTs will reflect the underlying control structures—making ASTs resilient against trivial obfuscations.

Tree-Edit Distance (TED): TED is a strong method for quantifying the structural similarity between two programs. It captures how much one program must be transformed to resemble the other by calculating the minimum number of insertions, deletions, or substitutions required to modify one tree into the other. This approach is ideal for structured data like ASTs, where hierarchical relationships need to be preserved during comparison. A lower TED indicates a higher structural similarity.

The structural similarity score, normalized using the size of the larger AST, provides a fair measure that works well even for programs of varying sizes and complexities.

$$S_{\text{structural}} = 1 - \frac{D_{\text{AST}}}{\max(\text{Size}(\text{AST}_A), \text{Size}(\text{AST}_B))}$$

3.2 Functional Similarity: Behavioral Equivalence Through Test Cases

Functional similarity focuses on how programs behave when executed. The method uses test cases to evaluate program outputs and determines whether two programs are functionally equivalent.

Even when two programs differ significantly in their structural organization, they may still produce identical results. This situation frequently arises when comparing optimized versus non-optimized code, or recursive versus iterative solutions. Functional similarity is thus crucial in contexts like software verification, where behavior is the ultimate measure of correctness.

Test Case Coverage: The test cases used for functional comparison need to cover both typical and edge-case scenarios to ensure the programs' behavior is rigorously evaluated. By comparing outputs across a diverse set of inputs, this methodology ensures that functional equivalence is based on a thorough examination of the programs' behavior.

$$S_{\text{functional}} = \frac{\sum_{i=1}^{N} \text{Match}_i}{N}$$

3.3 Content Similarity: Token Distribution and Entropy

Content similarity provides a quantitative measure of how information is distributed within the programs by focusing on token frequency and distribution. This methodology is particularly useful for assessing code complexity and stylistic consistency. By tokenizing the programs into their smallest syntactic components (keywords, operators, identifiers), we capture the coding style and the complexity of the code. This approach is ideal for detecting similarities in coding patterns, structure, and usage of language constructs.

Shannon Entropy: Shannon entropy is an ideal metric for this analysis because it measures the uncertainty or randomness in the token distribution. Programs with similar token distributions will have comparable entropy values, reflecting similar patterns of language use.

$$S_{\text{content}} = 1 - \frac{|H_A - H_B|}{\max(H_A, H_B)}$$

4 Time Complexity Analysis

The overall time complexity is determined by the combination of structural, functional, and content similarity.

Where:

- **Structural Similarity (AST + TED Calculation)**: Generating ASTs has a time complexity of $O(n)$, where n is the size of the program. The Tree-Edit Distance (TED) calculation has a complexity of $O(n^3)$, making it the most computationally expensive component of the algorithm.
- **Functional Similarity (Test Case Execution)**: Running test cases on both programs has a time complexity of $O(N \cdot T(n))$, where N is the number of test cases and $T(n)$ is the time complexity of the program being tested.
- **Content Similarity (Tokenization + Entropy Calculation)**: Tokenization has a linear time complexity of $O(m)$, where m is the number of tokens. Entropy calculation has a complexity of $O(m \log m)$.

Thus, the overall time complexity is dominated by the AST comparison for structural similarity and the test case execution for functional similarity, resulting in a complexity of $O(\max(n^3, N \cdot T(n)))$, where n is the size of the code and N is the number of test cases. Therefore, this may not be the most elegant or best algorithm designed to accomplish this task, but it is thorough. The next steps would involve refining it in making it more efficient.

References

[1] R. Koschke, *"Survey of Automated Techniques for Program Reverse Engineering"*, IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 671-694, 2010.

[2] C. E. Shannon, *"A Mathematical Theory of Communication"*, The Bell System Technical Journal, vol. 27, no. 3, pp. 379–423, 1948.

[3] K. Zhang and D. Shasha, *"Simple Fast Algorithms for the Editing Distance between Trees and Related Problems"*, SIAM Journal on Computing, vol. 18, no. 6, pp. 1245-1262, 1989.

[4] S. Schleimer, D. S. Wilkerson, and A. Aiken, *"Winnowing: Local Algorithms for Document Fingerprinting"*, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76-85, 2003.

[5] M. Joy and M. Luck, *"Plagiarism in Programming Assignments"*, IEEE Transactions on Education, vol. 42, no. 2, pp. 129-133, 1999.

[6] T. Ball and S. K. Rajamani, *"Automatically Validating Temporal Safety Properties of Interfaces"*, Proceedings of the 8th International SPIN Workshop on Model Checking of Software, pp. 103-122, 2001.

[7] B. S. Baker, *"On Finding Duplication and Near-Duplication in Large Software Systems"*, Proceedings of the Second Working Conference on Reverse Engineering, pp. 86-95, 1995.