

Structural Similarity Detection for Scratch Programs Using Block Relationship Reconstruction

Yingjiang Wang

College of Information Engineering, Suqian University, Suqian, Jiangsu, China

Abstract: Direct text comparison of Scratch programs is not feasible because they are shown in graphical blocks. This paper studies block relationship reconstruction and opcode sequence matching in an existing program comparison system. Read sprite and block records in the system, organise scripts according to their connections, convert opcodes to text, and calculate structural similarity with difference feedback. The implementation is shown by source code analysis, project modification and small-scale structure tests. Out of 32 cases, 30 returned a similarity score, and the two cases with malformed connections exceeded the time limit set by the test harness. As shown in the results, the method can distinguish among the selected opcode substitutions, block insertions and deletions, ordering changes, etc., and filters out some differences in identifiers and editor coordinates. However, some parameter values, Stage codes and branches have not been fully included in the comparison. Therefore, we can compare the structure with the reference template, but we do not know whether it is effective.

Keywords: Scratch; Block-Based Programming; Structural Similarity; Opcode; Static Analysis

1. Introduction

Scratch uses connected blocks [1] to express program logic and can create interactive stories, animations and games [2]. Teachers need to determine whether students have used the required blocks and control structures in the programming exercises according to the reference solutions. These are the tasks of instruction selection and organisation of sequences, conditions and loops. Not all arrangements of the editor are associated with different logics, and visually similar scripts can have distinct instructions. Therefore, detection needs to reconstruct block relationships from the

underlying data and convert the results of visual editing into a form suitable for computational comparison.

The current tools support all-around analysis of Scratch programs. Hairball provides static analysis [3], Dr. Scratch studies features of computational thinking [4], and LitterBox finds bug patterns and code smells [5]. Whisker and subsequent studies have explored automated execution, test generation, and model-based behavioural checking [6-8]. Code clone research also classifies program representations and their conditions of application [9]. The above studies have shown that finding possible problems, observing the runtime behaviour and comparing structures with a reference are all independent evaluation tasks. This study focuses on the last task and is limited to the implementation of the current project; it does not introduce a general-purpose sequence matching tool as a new algorithm. The system will build a workflow for teachers to review the reconstruction of block relationships from references, opcode text matching and API feedback. Assessment of computational thinking also includes practices and perspectives [10], and these cannot be reduced to a single structural score.

What structural information is retained in the current representation? What non-structural changes are not shown in the output? What changes have been omitted? To solve the problems mentioned above, we first analyze the data flow among the loading, reconstruction and comparison modules, and then observe similarity scores and difference reports after controlled modifications.

2. Method and Implementation

Figure 1 shows the general process. Parse and represent the reference program and the program to be compared in the same way before comparison. The workflow comprises four stages: project parsing, block relationship reconstruction, text matching, and result reporting. First, extract sprite and block records

from the project data, and use references among blocks to reconstruct the instruction order and some nesting relationships. Organised opcodes are then converted to text and used as the basis for comparing the two programs. Next, the system calculates character-level similarity and line-level difference. Compare the similarity score with a predefined threshold to determine

whether the program structures are similar; otherwise, find the location and content of changes at the line level when the score falls below 1. Finally, the API returns the score, a threshold-based decision, and any available difference information to inform teachers of further investigation of the parameter values and programme behaviour.

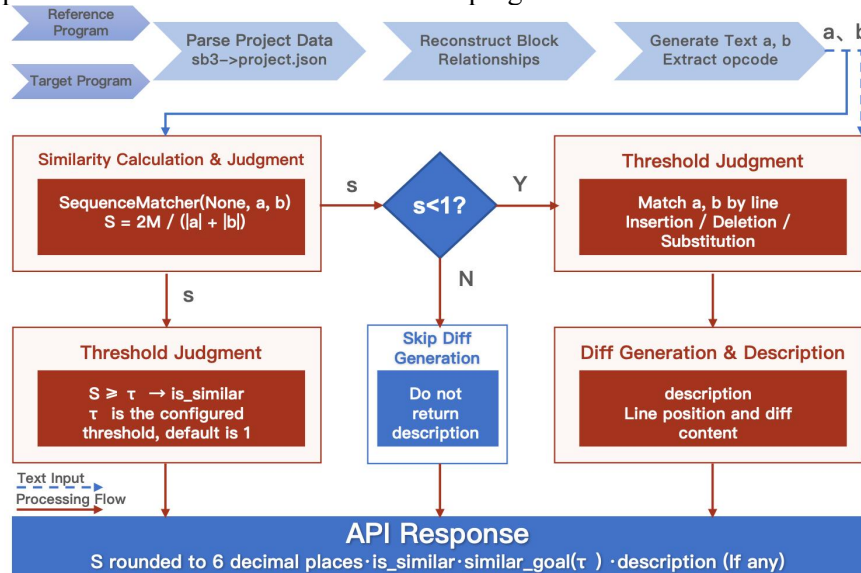


Figure 1. Data Flow and Conditional Branch Comparison in Scratch Programs

2.1 Project Parsing and Structural Reconstruction

The system reads project.json from an SB3 archive to get sprite and block data for the targets. The loading module can load local files, binary content and network addresses, and all of them are finally used for project parsing. Blocks are connected by identifiers: opcode specifies the instruction type, next indicates the following block, parent stores the parent reference, topLevel marks a top-level entry, and inputs stores input relationships. These fields correspond to those in the official serialization implementation [11]. Identifiers are used for lookup and are not included in the comparison content of the final text. Detection does not need to recognize screenshots of blocks or use their coordinates in the editor.

Structural Reconstruction first organises conditions and control bodies, then recovers script order from the top-level entries. The implementation scans the block collection, processes input references such as CONDITION and SUBSTACK, gathers the corresponding blocks into lists, and attempts to embed inner control bodies in the outer structures. Then it starts from topLevel blocks and follows the next

links to build script lists. For example, the opcodes in a sequential script are organised by successor links, and loop bodies are added through substack connections. Some nesting relationships are still preserved. Branch handling is based on the specific fields and list operations in the existing code; a full abstract syntax tree (AST) or equivalent processing is not performed for the second branch, SUBSTACK2.

This Design changes the comparison object from an archive and its assets to text representing instruction organisation. First, reconstruct the connection order prior to sequence matching and avoid direct comparison of the raw JSON. If the two programs have the same set of opcodes, different connection orders will result in different reconstructed text. Repeatedly scan the block collection for successors and carry out several list lookups and modifications in the nesting reconstruction. Therefore, parsing cannot be viewed as a single pass that visits each block only once. The actual cost will be higher if there are many blocks and poor organisation.

The project implementation orders non-Stage sprites by name, and each sprite object outputs its block list. The textual representation of each block retains only its opcode, and the text for different sprites is joined with newline

characters. List brackets, commas, and spaces are also included in matching as part of the string representation. Identifiers and editor coordinates are not output directly, eliminating some storage-related differences. Sprite names also affect comparison order indirectly through sorting. The Stage is excluded from the final sprite sequence used for comparison, and parameter and field values are not fully retained. The resulting representation therefore captures only part of the program structure rather than all of its semantic information.

Before invoking the comparator, the API also removes brackets, commas, and newline characters from the `str_blocks` attribute. However, the comparator reads text regenerated from the sprite objects. Character removal from this attribute therefore does not alter the scoring input. Distinguishing intermediate attributes from the actual call path helps avoid incorrectly describing operations that do not participate in the computation as algorithmic steps.

2.2 Similarity and Difference Feedback

Let a and b be the texts generated by the reference program and the program under comparison, respectively. `SequenceMatcher` from the Python standard library is used for implementation, characters are the comparison units, and the default automatic heuristic is enabled [12]. Use the tool to find the corresponding block in the sequence and calculate a ratio. It has not been adapted into a tree edit distance algorithm in this study, and it is not considered equivalent to a longest common subsequence algorithm. The corresponding similar items are:

$$S(a, b) = \frac{2M}{|a| + |b|} \quad (1)$$

Here, M is the total number of characters in the matching blocks, and $|a|$ and $|b|$ are the lengths of the two strings. A metric to assess the correspondence of the extracted representations, rather than the proportion of matching Scratch blocks. An opcode is composed of several characters, and different instruction names can use these characters. Therefore, replacing one instruction may not result in a further reduction of the score per block. The delimiters, text lengths and comparison directions of the results will also change. The default heuristic also includes frequently occurring features in the long sequence. Therefore, according to both the representation and the behaviour of the matcher,

the scores should be interpreted rather than considered direct indicators of functional equivalence.

Divide the text into lines and use matching tags to generate insertion, deletion and replacement information in the system. Difference reports contain line numbers and the corresponding content of the reference or compared text. The locations are references to the generated text, not coordinates in the Scratch editor, so they do not directly correspond to a specific visual block. The character-level score is a general difference; the line-level description provides details of the review. Therefore, the two outputs have different feedback functions.

2.3 API Structure and Decision Logic

The `/scratch/compare` endpoint accepts a reference program, a program under comparison, and a similarity threshold ranging from 0 to 1, with a default of 1. The system compares the unrounded score against the threshold and then rounds the returned score to six decimal places. This prevents display precision from affecting the decision. In the response, `is_similar` contains the threshold-based decision, `similar_number` contains the score, and `similar_goal` contains the specified threshold. Whenever the original score is below 1, the response may include description, even if the score meets a lower threshold.

3. Experiments and Analysis

The experiments directly invoked the comparison algorithm in the original project modules, without introducing an alternative detector. The project directory contains `sb3` files for two distinct projects verified by hashing. The two test projects contain 32 and 33 block records, respectively, including Stage records. A total of 32 cases were executed using mutations of the original projects and minimal structural inputs. Of these, 30 returned scores, while two cases with malformed connections exceeded the two-second limit imposed by the test harness. This limit was an experimental safeguard, not a timeout mechanism already implemented in the original service.

The controlled cases modified identifiers, coordinates, opcodes, connections, parameters, or branches to distinguish, as far as possible, the effects of preserved information from those of omitted information. Forward and reverse scores, generated text, and difference content were recorded to trace observed behavior back to the

processing pipeline.

We first examined output stability under non-structural changes. Each test project underwent self-comparison, consistent identifier renaming, editor-coordinate changes, and reversal of sprite storage order. All eight cases yielded a score of 1, indicating that these changes left the final representation unchanged. Identifier renaming updated the corresponding references simultaneously, avoiding the misclassification of broken connections as simple renaming. Comparing the reference solution with the other project yielded 0.319109; reversing the comparison direction yielded 0.211503.

We next examined whether structural edits changed the output. For a linear script containing four blocks, replacing a turn opcode, deleting a block, inserting a wait block, and changing connection order yielded scores of 0.951724, 0.859375, 0.912500, and 0.753425, respectively. Modifying an opcode within a single-level SUBSTACK yielded 0.887324. All five changes were distinguishable at a threshold of 1, showing that these structural edits, which were retained in the representation, affected the output. However, the differences between scores do not rank the edits by difficulty or error severity, and these five cases do not establish that changes to all nested structures can be detected.

Tests of information omission showed the primary deficiencies in the score. Changing the movement step or loop iteration from 10 to 100, modifying a key-event field, or changing only a second-branch opcode still resulted in a score of 1. Project-level mutations of Stage opcodes also left the score unchanged. The above observations are consistent with the missing retention of parameters, fields, Stage code and second branches in the text. Reversing the block dictionary order in the program under comparison reduced the score to 0.543103, and the same operation on the reference project still resulted in 1. As a result, the current reconstruction does not uniformly address all modifications in storage order.

Runtime measurement performed on macOS with the arm64 architecture and Python 3.12.14. Linear inputs with 10, 50, 100, and 200 blocks per project were each tested after five warm-up runs and then subjected to thirty timed runs. Excluding archive reading and network transfer, the median total time for parsing and comparing both projects was 0.290, 0.841, 2.073 and 5.694 ms. The median parsing time for the 200-block

input was 5.047ms and the median comparison time was 0.639ms; therefore, parsing accounted for the majority of the runtime under this condition. Based on the above results, this method has a low overhead for small linear projects and is suitable for concurrent service handling of relatively complex nested projects.

Individually test the six APIs using the Flask test client in Python 3.12.7. Self-comparisons are comparisons at different thresholds, invalid thresholds, missing parameters and a health check. The comparison of the two projects did not meet the default threshold, but it met a threshold of 0.3. Both responses mentioned the difference and the API separated the threshold decision from the difference generation. Invalid threshold and missing required parameters returned HTTP status code 400. In tests of malformed structures, both a dangling successor reference and a cyclic reference resulted in timeouts. Therefore, the original program cannot guarantee the prompt rejection of invalid references because there is no valid connection.

4. Discussion and Conclusions

The main achievement of the current project is the integration of a representation for block-based programs with a general-purpose matching tool. Underlying references are used to recover some of the instruction organisation, and connection information distributed in project records can be included in text comparison. The API returns the score and difference. Based on the above empirical results, it can be used to organise block-and-order screening tests and provide teachers with a structural reference.

Distinguish between structural similarity and behavioural correctness in practice. For exercises that need a specific type of loop or a fixed sequence of steps, structural feedback can be provided to check the implementation method. Creative works that support multiple algorithms and forms of interaction may deviate from the reference structure but are not considered errors. At the same time, a score of 1 may still be due to incorrect parameters or different runtime behaviour, and lowering the threshold will not recover the lost information. The direction of comparison should be the same, and according to the requirements of the exercise, parameters, stage codes and runtime behaviour need to be verified.

The sample size and range of tasks in the present study are relatively small. Tests of some

different projects and simple structural cases can demonstrate specific behaviour, but they cannot cover the assessment performance of all grades, exercise types, and programming styles. Teachers' annotations of the actual classroom activities can be used for further verification to explore how thresholds relate to people's judgments, and tests for invalid connections and complex control structures need to be added.

In short, the current method has built an end-to-end workflow from scratch for project data to structural scores and difference feedback, but the scope of comparison is limited by the extracted information. Clearly inform the user of the scope; it is more in line with the evidence than to interpret high scores as proof of functional correctness, and provide a basis for using the method as a teaching support tool.

References

- [1] MALONEY J, RESNICK M, RUSK N, et al. The Scratch programming language and environment. *ACM Transactions on Computing Education*, 2010, 10(4): Article 16.
- [2] RESNICK M, MALONEY J, MONROY-HERNÁNDEZ A, et al. Scratch: Programming for all. *Communications of the ACM*, 2009, 52(11): 60–67.
- [3] BOE B, HILL C, LEN M, et al. Hairball: Lint-inspired static analysis of Scratch projects. In: *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*. New York: ACM, 2013: 215–220.
- [4] ROBLES G, PONS G, ESCOBAR-MORALES D, et al. Dr. Scratch 2.0 (CT4ALL): Advanced modes and rubrics for assessing computational thinking in Scratch projects. *SoftwareX*, 2025, 32: 102359.
- [5] FEIN B, OBERMÜLLER F, FRASER G. LitterBox+: An Extensible Framework for LLM-enhanced Scratch Static Code Analysis. In: *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2025: 3989–3992.
- [6] FELDMEIER P, FRASER G, HEUER U, et al. A Block-Based Testing Framework for Scratch. In: *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. New York: ACM, 2024: Article 1, 1–12.
- [7] FELDMEIER P, SCHMELZ K, FRASER G. Many-Objective Neuroevolution for Testing Games. In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2025: 244–254.
- [8] GÖTZ K, FELDMEIER P, FRASER G. Model-based Testing of Scratch Programs. In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022: 411–421.
- [9] QURADAA F H, SHAHZAD S, ALMOQBILY R S. A systematic literature review on the applications of recurrent neural networks in code clone research. *PLOS ONE*, 2024, 19(2): e0296858.
- [10] BRENNAN K, RESNICK M. New frameworks for studying and assessing the development of computational thinking. In: *American Educational Research Association Annual Meeting*. Vancouver, 2012.
- [11] SCRATCH FOUNDATION. Scratch-vm: src/serialization/sb3.js. develop branch. Accessed September 23, 2026.
- [12] PYTHON SOFTWARE FOUNDATION. difflib—Helpers for computing deltas. Python 3.12 documentation. Accessed September 23, 2026.