Olympiads in Informatics, 2026, Vol. 20, 1–15

© 2026 S. Azizova, S. Muradov, J. Hasanov. Published by Vilnius University
DOI: https://doi.org/10.15388/ioi.2026.01

Empirical Study of Coding Behavior Under Time Constraints in the International Olympiad in Informatics

Shahla AZIZOVA, Seyidshah MURADOV, Jamaladdin HASANOV
School of IT and Engineering, ADA University, Baku, Azerbaijan
e-mail: sazizova11542, smuradov16150, jhasanov@ada.edu.az

Abstract. Competitive programming contests like the International Olympiad in Informatics (IOI) generate rich behavioral data beyond final scores, yet little attention has been paid to systematic analysis of coding styles and code evolution under time constraints. This paper presents an empirical framework for analyzing contestant coding behavior using data from the previous IOI contest. The authors introduce two complementary analytical pipelines: a structural feature extraction system that generates unified structural fingerprints for clustering analysis, and a temporal code evolution tracker that monitors token-level changes and similarity scores across consecutive submissions to reconstruct problem-solving trajectories. The analysis reveals that task structure is the dominant determinant of coding fingerprints, with medal performance associated with a gradient in code richness rather than qualitative stylistic shifts. The authors also present interactive visualization tools enabling coaches and contestants to trace submission histories, distinguish incremental refinements from complete rewrites, and identify behavioral patterns invisible to scoreboard analysis alone. This work contributes actionable insights for contest training, problem design, integrity monitoring, and programming education, demonstrating that systematic examination of how contestants produce solutions reveals meaningful patterns that final scores alone cannot capture.

Keywords: contest, CMS, real-time, coding behavior, stress

1. Introduction

Competitive programming contests provide a unique environment for observing how individuals solve complex algorithmic problems under strict time constraints. Among these contests, the International Olympiad in Informatics (IOI) is widely regarded as one of the most prestigious global competitions for secondary and high school students in computer science. Each year, hundreds of highly talented contestants from around the world compete by solving algorithmic problems that require deep knowledge of data structures,

algorithms, and efficient implementation techniques. While the primary evaluation criterion in such contests is the correctness and efficiency of the submitted solutions, the process through which contestants develop these solutions also contains valuable information about problem-solving strategies and programming practices.

In competitive programming, contestants typically work under strict time limits and high cognitive load. As a result, their coding behavior often reflects pragmatic decisions aimed at maximizing efficiency rather than maintaining conventional software engineering standards. Nevertheless, these coding artifacts provide a rich source of data for understanding how expert-level programmers structure algorithms, organize code, and adapt their implementation styles in a high-pressure environment. Examining these patterns can offer insights not only into competitive programming practices but also into broader aspects of programming education and algorithmic thinking.

Previous studies have explored different aspects of contest data analysis in IOI. In (Alnahhas et al., 2020; Audrito et al., 2025; Kostadinov et al., 2018), authors investigated how contest statistics and operational data can be used to improve contest organization and contestant preparation strategies. Their study demonstrated that analyzing real-time and post-contest data can reveal patterns that help organizers and coaches better understand contest dynamics and participant performance. More recently, (Mammadli et al., 2024; Hasanov et al., 2021) introduced analytical models for evaluating contestant progress and code similarity in real-time coding contests. Their work highlighted the importance of systematic analysis of submissions for monitoring contestant progress and ensuring the integrity of competitive environments. (Lee et al., 2024) analyzes IOI performance data, showing that countries' results correlate with demographic indicators such as population and Human Development Index, but even more strongly with the level of competitive programming interest in the country.

While these studies have demonstrated the value of analyzing contest-related data, relatively little attention has been paid to the systematic evaluation of coding styles and development patterns among contestants. Understanding how participants structure their solutions and evolve their code during the contest may provide valuable insights into algorithmic thinking, efficient programming practices, and competitive programming training methodologies. In particular, examining code organization, use of common templates, implementation patterns, and stylistic characteristics can help identify recurring strategies used by successful contestants.

This study aims to analyze contestant activities and evaluate coding styles in IOI competitions using empirical data derived from contest submissions and development traces. Specifically, the research investigates how contestants structure their programs, what stylistic patterns emerge across different solutions, and how these patterns relate to successful problem solving. The analysis focuses on several aspects, including code organization, naming conventions, modularization practices, and the use of common competitive programming templates. In addition, the temporal sequence of submissions and development iterations is examined to better understand how contestants refine their solutions during the contest.

By systematically analyzing these aspects, the study seeks to contribute to the understanding of programming behavior in high-performance algorithmic environments. The findings may provide insights for educators designing programming curricula, researchers studying algorithmic problem solving, and competitive programming communities interested in improving training methodologies. Ultimately, exploring the coding practices of top-performing contestants can help reveal patterns that characterize efficient algorithmic thinking and rapid software development under time constraints.

2. Problem Statement

The IOI format consists of two competition days (separated with one break day), each containing three algorithmic tasks to be solved within five hours. Contestants submit their solutions to an automated evaluation system, and the final public outcome is typically represented only by the scoreboard, which shows the achieved scores for each participant. While this final result reflects performance, it reveals little about the underlying problem-solving process that leads to the outcome.

In reality, the process of solving algorithmic tasks under strict time constraints involves multiple stages, including planning, coding, testing, debugging, and refining solutions. These stages are reflected in the sequence of submissions and in the evolution of the source code written by contestants. However, this behavioral dimension of competitive programming is rarely visible to observers or researchers because the available public data usually focuses on final scores rather than the development process of solutions.

Previous work has attempted to extract insights from contest activity data. For example, (Hasanov et al., 2021) analyzed submission patterns to understand the strategies of medal-winning contestants based on submission order and frequency. Such approaches demonstrate that submission logs can provide valuable information about the strategies participants employ during contests. Despite these efforts, there is still limited understanding of how contestants differ in their coding behavior under time pressure. In particular, little is known about how participants' coding styles, the evolution of their programs during the contest, and the programming constructs they use may reflect different problem-solving approaches. Understanding these aspects could reveal distinct contestant personas and strategies that are not observable from final scores alone.

Therefore, the problem addressed in this paper is the lack of empirical analysis of coding behavior and code evolution during high-level programming contests. By examining submission histories and source code changes of IOI participants, this study aims to identify patterns in coding style, development dynamics, and programming constructs used under strict time constraints, and to characterize different behavioral profiles of contestants during the competition.

3. Methodology

The study in this paper is performed in two directions: analysis of the coding style of the contestants based on the programming constructs and visualization of the task-based performance to see the dynamics of the code evolution.

The first of the proposed studies has been conceptualized as a multi-stage analytical pipeline that processes the submissions for the programming contests and performs a subsequent analysis and visualization. An overview of the entire workflow is depicted in Figure 1. The colors codes of the first and the second processes are depicted in light brown and green, respectively. As shown in the diagram, the workflow starts with the retrieval of the submissions from the CMS database, which is triggered by the submission control code. Each submission has been considered as an individual unit of analysis. This represents an individual step in the iterative problem-solving behavior of the contestants. After the retrieval process, the source code is subjected to a pre-processing step, in which the code is cleaned and normalized in order to ensure consistency in the input data. As a next step, the pre-processed source code is fed into the syntax analysis component, wherein the code is transformed into an Abstract Syntax Tree (AST). The use of a syntax tree plays a vital role in the methodology, as the structure of the program, as opposed to the surface features of the source code, can be analyzed. Subsequently, the pipeline extracts the features from the AST, yielding a concise set of structural descriptors for the submissions. Finally, the extracted features are subjected to the fingerprint generation module, which transforms the submissions into a unified structural signature.

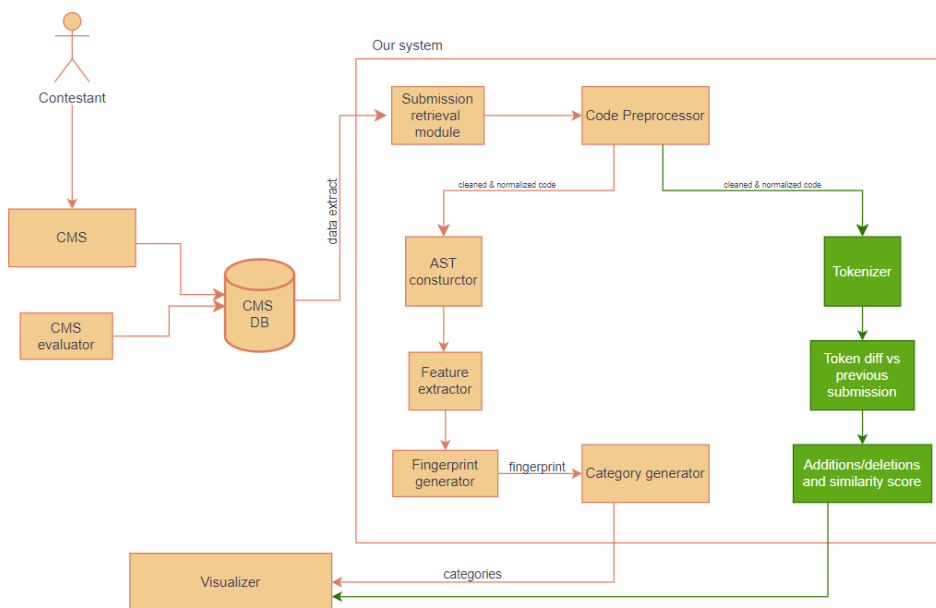

Figure 1. Overview of the Proposed Code Evolution Analysis Pipeline with the structural feature extraction, categorization.

It is noteworthy that the extracted features are transformed into a unified structural signature, allowing for the efficient comparison of the submissions based on the abstraction of the coding style and structure. Following these representations, the category generation module categorizes the contest submissions according to similarities observed in their structural fingerprints, as demonstrated in the flow diagram. These categories represent the implementation patterns and coding behaviors observed among the contestants. Lastly, the results are displayed through a visualization module, which allows for the interpretation of the observed structural patterns at different levels of aggregation, from individual contest submissions, tasks, and finally, contestants. This is in line with the objective of identifying the differences in the approaches adopted by the contestants in solving the problems and their coding styles. Overall, as demonstrated in Figure 1, the pipeline represents a systematic transformation from the raw contest submissions to the results, with a clear emphasis on modularity, reproducibility, and the structure of the programs as opposed to their programming characteristics.

3.1 Structural Analysis Pipeline Implementation

As mentioned earlier, the structural analysis pipeline is implemented through automated scripts that process raw source code submissions to extract structural representations from them. Each script within the process chain completes particular computations and outputs actual results, ensuring that the whole procedure can be replicated. The process starts with processing each source code file received in the course of the coding competition, where every file corresponds to one submission. Within the framework of the chosen approach, each submission will be analyzed individually, providing opportunities to examine the task and author-related features. In the initial step, the input code is subject to normalization. During normalization, non-structural elements such as comments, redundant whitespace, and formatting variations are removed, and this ensures consistency across inputs and is stored in the *normalized_code* file, associated with the unique submission. A copy of the code in its original form is saved under the file name *raw_submitted_code* and can serve for further referencing if needed. Following normalization, an Abstract Syntax Tree (AST) is generated using the Tree-sitter framework (Sklower, M. 2018). The AST represents the hierarchical syntactic structure of the source code, enabling analysis at the structural level rather than relying on raw text. Finally, the created abstract syntax tree is translated into the JSON format and saved in the *ast.json* file associated with the unique submission. Each stage of the pipeline yields a structure that is stored in a separate file with the reference to the unique submission. This approach provides end-to-end traceability from the original inputs to their final output forms enabling individual examination and validation of each step separately.

Based on the AST, several structural attributes are calculated to provide quantitative analysis of the code. These attributes cover various aspects of software development, such as structural complexity and flow control structures. The following list shows the calculated attributes:

- Maximum depth of the AST (proxy for structural complexity)
- Number of functions
- Number of loops and conditional statements
- Number of return statements
- Counts of control flow modifiers such as break and continue
- Number of included libraries and macro definitions

The calculated attributes are saved in two formats – as a CSV file features.csv and JSON file features.json.

3.2 Feature Vector Construction and Fingerprinting

For the purpose of comparing all submissions efficiently, the extracted features are converted into a feature vector with a fixed order. The order of the features is specified explicitly to maintain uniformity while comparing features. The feature vector is serialized into a concise format (for example, "11|2|0|4|6|0|0|2|0") and acts as a signature of the structural details of the program. A SHA-256 hash function is applied to this string to create a fingerprint of the structure of each submission. The fingerprint is saved in fingerprint.json along with the original feature vector and feature order. The above operation facilitates comparison between submissions, enables the identification of structurally similar programs, and supports grouping of coding styles.

3.3 Feature Selection and Representation

In terms of choosing the structural features, the objective is to represent each program in such a way that sensitivity to surface variation is minimized while preserving key structural characteristics of the program. In this study, the analysis does not rely directly on the raw source code, but rather on features derived from its AST. The reason behind this decision is to ensure that the representation is independent of surface variations such as formatting or individual programming style. The features consist of counts of fundamental program structures, such as functions, loop constructs, and conditional statements. The attribute set deliberately does not contain any elements that can only be obtained based on specific naming schemes, indentation styles, etc. This ensures that structural information about the programs written by various contestants can be represented in a uniform manner. It should not be assumed that these attributes describe the full range of coding behavior exhibited by the participants. Instead, they represent a simplified model of structural information about their work. For purposes of comparison, all the extracted features are put into an ordered vector. This way, every submission will have its features in a uniform form so that comparisons can be made among various programs. This vector is then converted to string form and hashed using a normal cryptographic algorithm. The hashing process serves merely the purpose of creating unique identifiers for each structural feature set and does not make any assumptions beyond this.

3.4 Code and performance evolution measurement

The second study pursues a different correspondence pipeline, which involves monitoring the dynamic progress of individual contestants regarding the dynamic development of their code in successive submissions of a particular task. As demonstrated in (Kostadinov et al., 2018), analyzing the sequence of submissions rather than only the final score reveals behavioral patterns that are not observable from the scoreboard alone. An overview of the proposed workflow is depicted in Figure 1.

Similar to the first study, the pipeline is driven by submissions retrieved from the CMS database. Each submission is uniquely identified by its submission ID, contestant identifier, task name, and the score assigned by the automated evaluation system. Submissions are grouped by contestant and task, then ordered chronologically by submission ID to reconstruct the development timeline of each solution.

After the submissions are ordered, every source file continues to a pre-processing phase. Single line comments, multi-line comments and preprocessor directives are eliminated. Tab characters are converted to single spaces, successive spaces are merged, and all empty lines are removed. This step is used in order to make sure that the later structural analysis is not influenced by superficial variations in the formatting or commenting style which range widely across contestants, but which do not carry algorithmic significance. Table 1 illustrates the transformation applied to a representative code fragment across the three stages: original source, cleaned source, and normalized token sequence.

The processed source code is then subjected to the tokenization module which converts the processed source code into a normalized sequence of abstract tokens. Inspired by established token-based structural analysis approaches (Prechelt et al., 2002), all identifiers and keywords are replaced with the generic token *ID*, all numeric literals are replaced with *NUM*, and operators are retained in their original form. This normalization ensures that two structurally identical solutions written by different contestants but using different variable names produce identical token sequences, enabling a fair structural comparison. After tokenization, pairs of subsequent submissions of the same contestant and task are compared by using a diff function directly on the token sequences. The *diff* calculates the number of tokens that were added and removed between the current and the previous submissions, which gives a quantitative measure of the extent to which the code structure was modified at each step. As a by-product of the same comparison, a similarity score is calculated with the Ratcliff-Obershelp algorithm (Ratcliff et al., 1988)and results in a ratio of 0 to 100 percentage, with 100 percentage being a structurally identical submission and lower values reflect a more substantial reorganization of the solution.

Table 1. Example of the three-stage pre-processing transformation applied to a contestant's source code.

Original code	**Cleaned code**	**Token sequence**

#include <bits/stdc++.h>
#define MAX 100
/* initialize sum */
int n, sum = 0;
// loop over input
for(int i=0;i<n;i++){
sum=sum+a[i]; // add
}
	(removed)
	(removed)
	(removed)
	int n, sum = 0;
	(removed)
	for(int i=0;i<n;i++){
	sum=sum+a[i];
	}
		ID ID , ID = NUM ;
		ID (ID ID = NUM ;
		ID < ID ; ID ++) {
		ID = ID + ID
		[ID] ;
		}

8 lines · 4 comments/directives	4 lines · comments removed	19 tokens · 3 types: ID, NUM, operators

Finally, the results are passed to the visualization module which shows the progress of the selected contestant on the given task over the time. The progress contains the additions/removals to the code at each submission, with the received score.

4. Experiments

4.1 Dataset and Experimental Setup

The proposed models have been run on 8113 submissions from International Olympiad in Informatics 2020 hosted in Singapore, held online from 13–19 September 2020, which gathered 343 contestants from 87 countries and awarded 29 gold, 57 silver, and 85 bronze medals. Each record in the fingerprint table corresponds to a single (contestant, task) pair and is identified by a composite key of the form *contestant_taskname*. From each submission, nine structural features were extracted via abstract syntax tree (AST) analysis: (1) *ast_max_depth*, the maximum nesting depth of the parse tree; (2) *num_functions*, the number of distinct function definitions; (3) *num_loops*, the total count of loop constructs; (4) *num_conditionals*, the number of conditional branching statements; (5) *num_returns*, the count of return statements; (6) *num_breaks*, the count of break statements; (7) *num_continues*, the count of continue statements; (8) *include_count*, the number of preprocessor include directives; and (9) *macro_count*, the number of macro definitions. Together, these nine dimensions constitute what we term the *coding fingerprint* of a submission, capturing structural and stylistic properties of a contestant's solution independent of correctness.

4.2 Data Processing and Cluster Analysis

Prior to analysis, the fingerprint table was joined with the medal registry on the contestant identifier extracted from the composite key. All nine features were normalized by their respective observed maxima across the entire dataset to produce values in the range [0, 1], enabling cross-feature comparison on a common scale. To identify latent groups of coding behavior, we applied k-means clustering (Lloyd's algorithm, $k = 6$, 20 random initializa-

tions, random seed 42) to the standardized (zero-mean, unit-variance) feature vectors. The number of clusters was selected to balance interpretability with granularity; preliminary runs with $k \in \{4, 5, 6, 7\}$confirmed that six clusters yielded the most semantically coherent groupings without over-segmenting the data. Cluster centroids were back-transformed to the normalized scale for visualization. To characterize coding styles across outcome groups and tasks, we additionally computed per-group mean normalized profiles for the four medal tiers and the three tasks separately. All profiles were visualized as nine-axis radar charts, with each axis corresponding to one of the nine extracted features.

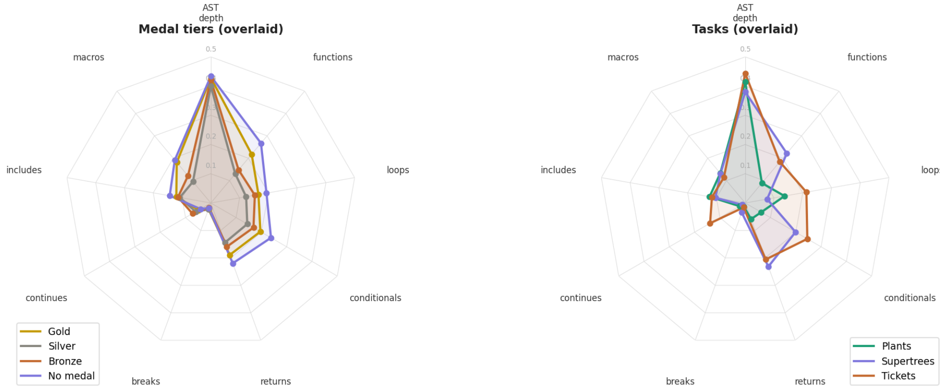

Figure 2. Generalized coding fingerprint profiles per medal tiers and tasks.

The radar profiles for the four medal tiers (Figure 2a) reveal a consistent but moderate gradient across all nine features: Gold medalists exhibit higher mean values than Silver, Bronze, and unmedalled contestants on *ast_max_depth*, *num_functions*, *num_conditionals*, and *num_returns*, while the pattern on loop-related and flow-control features is less discriminative. Crucially, the shape of the radar polygons is broadly concentric - the qualitative structure of a submission's fingerprint does not change categorically between medal tiers; rather, the magnitudes scale monotonically with performance. This suggests that higher-performing contestants write structurally richer code, particularly with respect to functional decomposition and branching complexity, rather than adopting a qualitatively different programming style.

The task-level profiles (Figure 2b) present a strikingly different picture. The three tasks impose markedly distinct structural demands: *supertrees* elicits deep AST structures and high loop and continue counts, reflecting the iterative graph-theoretic nature of the problem; *plants* rewards functional decomposition, with the highest mean *num_functions* and *num_returns* across all three; and *tickets* yields the sparsest profiles overall, with notably low conditional and return counts, consistent with its more algebraic character. These task-level differences in fingerprint shape are substantially larger than the inter-tier differences within any single task.

The same clustering framework can be naturally extended to the individual level by mapping each contestant's submissions onto the derived clusters over time. This enables track-

ing how a specific participant's coding behavior evolves, allowing a coach to compare their structural patterns, consistency, and progression against those of other team members or against aggregate profiles. Such an analysis could provide actionable insights for targeted training and performance improvement. However, due to the increased complexity of visualizing individual trajectories alongside cluster-level summaries, as well as considerations related to participant privacy, we deliberately chose not to include identifiable individual-level results in this study.

Table 2. Summary of the six identified coding style clusters. n = number of (contestant, task) pairs assigned to each cluster.

Cluster	Label	n	Dominant task	Top medal tier
1	Loop-heavy	246	Supertrees (79%)	Bronze / None
2	Break-intensive	50	Mixed	None (68%)
3	Macro-rich	145	Plants + Tickets	Balanced
4	Function-first	84	Plants (71%)	Balanced (Gold 25%)
5	Minimal / sparse	430	Plants + Tickets	None (59%)
6	Include-heavy elite	8	Mixed	Gold + Silver (75%)

The six clusters (Table 2) reinforce the primacy of task structure. Cluster 1 (Loop-heavy, $n = 246$) is composed almost entirely of supertrees submissions (79%) and is characterised by elevated loop and continue counts. Cluster 5 (Minimal/sparse, $n = 430$, the largest cluster) captures the majority of plants and tickets submissions written with flat, structurally simple code; its medal distribution skews heavily toward unmedalled contestants (59%). Cluster 4 (Function-first, $n = 84$) is the most medal-predictive group: with 25% Gold, 29% Silver, and 24% Bronze assignments, it is the only cluster with a near-uniform medal distribution, suggesting that a habit of functional decomposition — as applied predominantly to the *plants* task — correlates with competitive success independently of task assignment. Cluster 6 (Include-heavy elite, $n = 8$) represents a small population of experienced contestants who employ extensive preprocessor machinery (mean normalised *include_count* ≈ 0.87, *macro_count* ≈ 0.41); 75% of its members are Gold or Silver medallists, consistent with the use of competitive-programming template headers as a marker of prior experience.

Cluster 2 (Break-intensive, $n = 50$) is notable for its disproportionately high *num_breaks* score (mean ≈ 0.22, more than ten times the dataset mean), despite otherwise unremarkable profiles; 68% of members are unmedalled. While the causal direction is unclear, the elevated break usage may reflect iterative or brute-force solution strategies that do not scale to full marks under IOI constraints.

A web-based application has been developed to trace the performance of the individual contestant. The dashboard of the application enables the selection of the contestant and the task from the list (Figure 3) and see the performance of the user over the time.

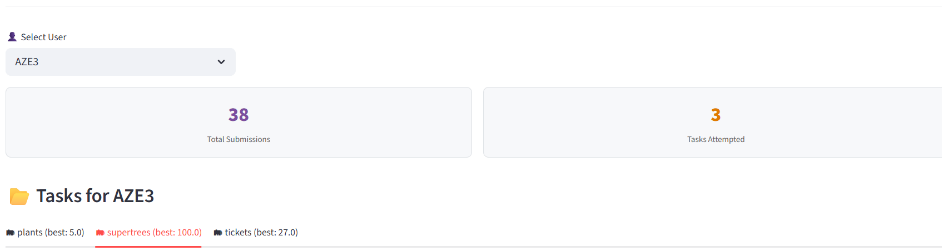

Figure 3. A visualization tool for tracing the contestant performance

Figure 4 shows the token addition/deletion graph of the same contestant in a series of submissions on a given task. The green bars indicate tokens added and the red bars indicate tokens removed on each submission step and the dotted orange line indicates the corresponding score trajectory on the right axis. Figure 5 shows a token-based similarity chart of that same contestant, which reveals how structurally similar each submission was to the previous submission. Combining these two visualizations enables the viewer to trace the structural code changes with performance improvements or decreases, and to draw a line between individual refinements and total rewrites of the solution. Overall, the second pipeline transforms raw submission histories into an interpretable representation of the problem-solving process, enabling the analysis of development patterns, revision strategies, and score progression at both the individual contestant and task levels.

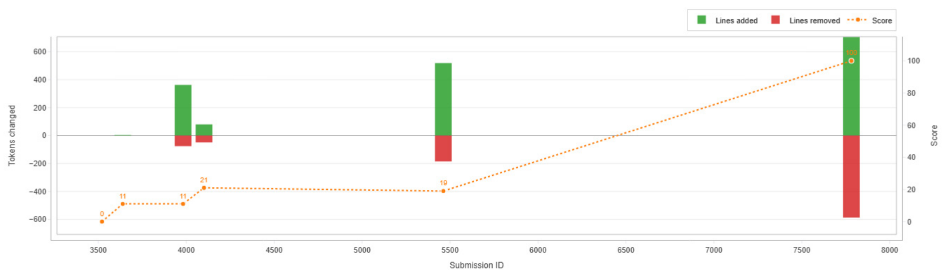

Figure 4. Token Additions and Deletions Across Consecutive Submissions with Score Trajectory for a Representative Contestant.

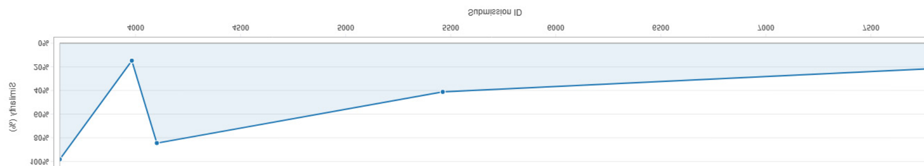

Figure 5. Token-Based Structural Similarity Between Consecutive Submissions for a Representative Contestant.

5. Conclusion

In this study, we have presented an empirical framework for analyzing coding behavior under time constraints in the International Olympiad in Informatics, introducing two complementary analytical pipelines: structural feature extraction for code fingerprinting and temporal code evolution tracking across submission sequences. Our methodology demonstrates that systematic examination of contestant submissions reveals meaningful patterns beyond final scores, offering valuable insights into problem-solving dynamics in high-pressure algorithmic environments. By shifting the focus from final scores to the underlying development process, the proposed multi-stage pipeline provides a granular view of how contestants navigate complex algorithmic challenges under strict time constraints and high cognitive load. The experimental application to 8,113 submissions from IOI 2020 demonstrates that our approach successfully identifies structural patterns and enables the objective grouping of coding styles across contestants from different countries and backgrounds. The provided visualization of changes alongside score trajectories allow observers to distinguish between incremental refinements and complete solution rewrites, thereby illuminating the strategic decisions contestants make when confronted with failing submissions or time pressure.

Our findings carry substantial practical implications for multiple stakeholders within the competitive programming community. For contestants, our analytical framework offers opportunities for systematic self-reflection and skill development. By examining their own submission histories, contestants can identify recurring patterns in their coding behavior, such as whether they tend to iterate gradually through small modifications or restructure solutions entirely after encountering errors. Understanding these personal tendencies can help contestants develop more effective revision strategies, improve time management during competitions, and ultimately enhance their performance. Moreover, comparing one's structural fingerprints with those of higher-performing peers may reveal algorithmic patterns or coding practices that warrant adoption. For coaches, our system provides a diagnostic tool that extends far beyond traditional scoreboard analysis. Rather than relying solely on final outcomes, coaches can examine how their students' code evolves over the course of a contest, identify recurring inefficiencies such as excessive restructuring or frequent minor fixes that indicate inadequate initial planning, and tailor training interventions accordingly. The ability to visualize token addition and deletion patterns alongside similarity scores enables coaches to deliver evidence-based feedback on problem-solving approaches, helping students internalize more efficient coding practices under competition conditions. For the IOI technical and scientific committees, our work contributes several actionable outcomes of both operational and strategic significance. The fingerprinting mechanism can enhance contest integrity by enabling the automated detection of unusually similar solution structures across contestants, potentially identifying unauthorized collaboration or code sharing. The evolution analysis can inform problem design and contest formatting: tasks that consistently produce erratic submission patterns, low structural stability, or high rates of complete rewrites may indicate poor problem specification, ambiguous statement wording, or unexpected difficulty spikes. Furthermore, our pipeline provides

a replicable framework for post-contest analysis, allowing committees to generate detailed behavioral reports that go beyond traditional scoreboards and medal distributions, thereby informing evidence-based decisions about contest duration, time allocation across tasks, and evaluation criteria.

The results of the cluster analysis lead to three principal conclusions. First, task structure is the dominant determinant of a contestant's coding fingerprint. The algorithmic demands of each problem, specifically graph traversal, recursive decomposition, or arithmetic reduction impose structural constraints on solutions that overshadow the effects of individual skill level. This has methodological implications for studies that seek to use static code features as proxies for programmer ability: any such analysis must control for problem identity to avoid confounding. Second, medal performance is associated with a gradient in code richness, not a qualitative shift in style. Gold-tier contestants write structurally deeper and more decomposed code, but the overall shape of their fingerprints is not categorically different from lower-tier contestants solving the same task. This gradient is most pronounced in function count and return statements - features that proxy for modular design and least pronounced in loop and flow-control features, which are more strongly determined by task requirements. Third, a function-first coding style is the strongest per-cluster predictor of medal success. Cluster 4, characterized by high function decomposition and applied predominantly to the plants task, is the only cluster in which Gold, Silver, and Bronze members are represented at approximately equal rates. This finding is consistent with the pedagogical literature linking structured, modular programming to problem-solving efficacy in competitive contexts. Taken together, these findings suggest that static structural fingerprints capture a mixture of task-specific and contestant-specific signal. Disentangling these contributions, for instance, through within-task normalization or multi-task fingerprint aggregation remains an important direction for future work.

We also see several promising avenues for future work emerging from our study. Expanding the feature set to include more sophisticated metrics, such as cyclomatic complexity, information-theoretic measures of code entropy, or dependency graph analysis, could provide deeper insights into the algorithmic design choices contestants make under time constraints. Applying our pipeline to multiple IOI editions would enable longitudinal studies of how coding styles and problem-solving strategies evolve across years and in response to changes in contest format or available programming languages. Integrating machine learning techniques could enable predictive models that identify contestants at risk of poor performance based on early submission patterns, potentially facilitating real-time interventions in training environments. Cross-contest comparisons between IOI and national contests or online judges could reveal how contest format and time pressure influence coding behavior across different contexts. Finally, developing an interactive web-based platform that makes our analytical tools accessible to the broader competitive programming community would maximize the practical impact of this work, allowing coaches, contestants, and researchers to explore submission data interactively.

In summary, our study demonstrates that systematic analysis of coding behavior in competitive programming contests reveals meaningful, actionable patterns that are entirely

invisible when examining final scores alone. The frameworks we have introduced for structural fingerprinting and evolution tracking provide valuable tools for understanding algorithmic problem-solving under time constraints, with direct applications for training methodologies, contest design, performance evaluation, and integrity assurance. As competitive programming continues to grow in global prominence, we believe that the ability to analyze not merely what contestants produce but how they produce it will become increasingly important for educators, coaches, organizers, and researchers alike.

References

Alnahhas, A., Mourtada, N. (2020). Predicting the performance of contestants in competitive programming using machine learning techniques. Olympiads in Informatics, 14, 3–20. DOI: 10.15388/ioi.2020.01

Audrito, G., Laura, L., Orlandi, A., Ostuni, D., Rizzi, R., Versari, L. (2025). Interactive problem solving in the classroom: Experiences with Turing Arena Light in competitive programming education. Olympiads in Informatics, 19, 1–25. DOI: 10.15388/ioi.2025.01

Hasanov, J., Gadirli, H., Baghiyev, A. (2021). On using real-time and post-contest data to improve the contest organization, technical/scientific procedures and build an efficient contestant preparation strategy. Olympiads in Informatics, 15, 23–36. DOI: 10.15388/ioi.2021.03

Kostadinov, B., Jovanov, M., Stankov, E. (2018). Platform for analysing and encouraging student activity on contest and e-learning systems. Olympiads in Informatics, 12, 85–98. DOI: 10.15388/ioi.2018.07

Lee, E., Reizin, T., Wu, F.E., Wu, F.E. (2024). Trends on returning contestants and geography at the International Olympiad in Informatics. Olympiads in Informatics, 18, 33–50. DOI: 10.15388/ioi.2024.0333

Mammadli, M., Mammadli, N., Hasanov, J. (2024). Analysis and evaluation of the contestant's progress in real-time coding contests. Olympiads in Informatics, 18, 51–62. DOI: 10.15388/ioi.2024.0451

Prechelt, L., Malpohl, G., Philippsen, M. (2002). Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science, 8(11), 1016–1031. DOI: 10.3217/jucs-008-11-1016

Ratcliff, J.W., Metzener, D. (1988). Pattern matching: The Gestalt approach. Dr. Dobb's Journal, 46.

Sklower, M. (2018). Tree-sitter: An incremental parsing system for programming tools. Tree-sitter. Available at: https://github.com/tree-sitter/tree-sitter

S. Azizova is a Computer Science graduate from ADA University. Her interests include data analysis, machine learning, and computer vision applications. She is currently working on different data analytics tasks that analyzes the computer code and biological data sequences.

S. Muradov is a last year Information Technology student at ADA University. His interests include data analysis and machine learning. As a part of his Senior Design Project, Seyidshah is using CMS and the contest data to design a system for the code progress review and analysis.

J. Hasanov is an Associate Professor of Computer and Information Sciences at ADA University. Dr. Hasanov is mainly focused on image processing and machine learning problems covering text and digital object recognition domains. Ad¬ditional to the research field, Dr. Hasanov teaches the management as¬pects of the IT in production and operation environments. Dr. Hasanov has been an ITC member for the period of 2017-2020 and chaired HTC during IOI 2019.