

DOLOS

SOURCE CODE SIMILARITY DETECTION

Rien Maertens

Supervisors:

Prof. Dr. Peter Dawyndt

Prof. Dr. Ir. Bart Mesuere

A dissertation submitted to Ghent University in partial fulfilment of the requirements for the degree of
Doctor of Computer Science.

Academic year: 2024 – 2025

Despite giving commentary on the technology, this dissertation made use of GenAI to improve the writing and style of the text in the form of Mistral AI's service *Le Chat*. The model was prompted to improve draft text snippets for clarity, writing style, and formulations. Actual human intelligence conducted the conceptualisation, research and programming underpinning this dissertation.

The cover of this dissertation is designed by Gert Maertens and depicts an abstract interpretation of a plagiarism graph as astrological signs.

Samenvatting

Door de toenemende digitalisering van onze samenleving doceren steeds meer opleidingen een vak programmeren. Leren programmeren is echter helemaal niet evident; deze vaardigheid bestaat uit verschillende bouwstenen die op elkaar voortbouwen. Als een student slechts één van die bouwblokken niet onder de knie heeft, bijvoorbeeld door een gemiste les, is de kans al groot dat die student er niet in slaagt om een werkende oplossing voor een programmeeropdracht te maken.

In dergelijke situaties kan **druk** om goed te presteren, de **rationalisatie** om de leerstof later bij te benen, en de **opportuniteit** om elders een werkende oplossing te bemachtigen, leiden tot de beslissing van die student om broncode te plagiëren. Deze vorm van academisch fraude houdt in dat broncode van een externe bron wordt voorgesteld als eigen werk, zonder de originele bron te vermelden. Vaak worden lichte aanpassingen aangebracht om het plagiaat te verdoezelen.

Omdat deze vorm van plagiaat nefast is voor het leereffect bij de student en het evaluatieproces ondermijnt, maken sommige lesgevers gebruik van similariteitsdetectieprogramma's. Deze programma's detecteren overeenkomsten tussen broncode die wijzen op plagiaat. De huidige programma's zijn vaak verouderd, onhandig, en voldoen niet aan moderne privacyverwachtingen. Daarom is het aantal lesgevers die deze programma's gebruiken, beperkt. Om hieraan tegemoet te komen, hebben wij een nieuw similariteitsdetectieprogramma ontwikkeld, genaamd **Dolos**. Dolos heeft als doel om gelijkenissen tussen broncodebestanden die zijn ingediend voor een programmeeropdracht snel en efficiënt te detecteren. De resultaten worden getoond in een gebruiksvriendelijke interface die de zoektocht naar plagiaat vergemakkelijkt. Ons doel is om Dolos zo toegankelijk mogelijk te maken zonder dat de gebruikers inboeten aan flexibiliteit. Dit proefschrift beschrijft de totstandkoming van Dolos en duikt in de verschillende aspecten die hierbij aan bod komen.

Dolos is de godheid van bedrog en misleiding in de Griekse mythologie.

Hoofdstuk 1 begint met een inleiding over plagiaat van broncode. We behandelen wat precies wordt bedoeld met deze vorm van plagiaat, hoe vaak het voorkomt, wat studenten drijft om plagiaat te plegen, en

welke technieken lesgevers kunnen toepassen om het tegen te gaan. Daarnaast bespreken we de verschillende vormen van plagiaat van broncode en de methoden die studenten gebruiken om hun code aan te passen en zo plagiaat te verdoezelen.

In hoofdstuk 2 gaan we dieper in op hoe we door deze verdoezelingen heen kunnen kijken om gelijkenissen te detecteren die kunnen wijzen op plagiaat. We bespreken een selectie van de meest gebruikte similariteitsdetectietools waaronder Moss, JPlag, Plaggie, Sherlock Warwick, Sherlock Sydney en Compare50.

Daarna beschrijven we het algoritme onder de motorkap van Dolos in hoofdstuk 3. Dolos zet de onderzochte broncode om in een syntaxboom, een systematische voorstelling van de onderliggende structuur van een programma. Deze syntaxboom is onafhankelijk van witruimte en namen die programmeurs vrij kunnen kiezen, zaken die gemakkelijk aangepast worden om plagiaat te verhullen. Vervolgens wordt de syntaxboom omgezet in digitale vingerafdrukken die vergeleken worden tussen de verschillende broncodebestanden. Deze vingerafdrukken zijn robuust tegen verdoezelingen die de onderliggende structuur van broncode aanpassen, zoals het verwisselen van de volgorde van functies. De resultaten van deze similariteitsanalyse worden geaggregeerd in een digitaal rapport dat alle informatie visueel weergeeft aan de gebruiker.

Hoofdstuk 4 behandelt het ontwerp en de ontwikkeling van de gebruikersinterface van Dolos. Hier is veel aandacht besteed aan een goede gebruikerservaring en toegankelijkheid. We beschrijven de verschillende visualisaties en hoe deze kunnen worden gebruikt om plagiaat op te sporen. Dolos houdt de resultaten overzichtelijk door middel van interactieve pagina's die informatie tonen over clusters en individuele bestanden, in tegenstelling tot paarsgewijze vergelijkingen die snel overweldigend worden bij veel plagiaat. Vergeleken met andere programma's biedt Dolos een uitgebreide gebruikerservaring met meerdere innovatieve visualisaties.

We beschrijven in hoofdstuk 5 de implementatie van Dolos. Zoals goed programmaontwerp dicteert, is Dolos verdeeld in verschillende componenten, elk met hun eigen verantwoordelijkheid. Deze modulaire softwarearchitectuur biedt veel flexibiliteit en maakt het mogelijk om verschillende gebruikersgroepen te ondersteunen. De functionaliteiten van Dolos worden gratis aangeboden via een website¹, een online API, een CLI, en in softwarebibliotheken. Één van de belangrijkste kenmerken van Dolos is losse koppeling tussen de algoritmen

¹dolos.ugent.be/server

en de ondersteuning voor verschillende programmeertalen. Door gebruik te maken van een bibliotheek die syntaxbomen kan genereren voor meerdere programmeertalen, ondersteunt Dolos een breed scala aan programmeertalen en is het eenvoudig om nieuwe talen toe te voegen. De broncode van Dolos is gratis en publiek beschikbaar op github.com/dodona-edu/dolos.

In hoofdstuk 6 vergelijken we Dolos met andere similariteitsdetectie-tools op verschillende aspecten. Met gebruiksstatistieken tonen we aan dat Dolos steeds vaker opgepikt wordt door diverse gebruikersgroepen. Middels een aantal benchmarks tonen we dat Dolos qua prestaties, uitvoeringstijd en geheugengebruik minstens gelijkwaardig is aan, of beter presteert dan, bestaande alternatieven. Een gebruikersenquête toont aan dat de gebruikerservaring positief wordt beoordeeld. Ten slotte beschrijven we hoe Dolos een essentieel onderdeel is geworden van onze strategie om plagiaat te voorkomen en te bestrijden in onze eigen cursussen.

Hoofdstuk 7 bespreekt de verschillende experimenten die zijn uitgevoerd om Dolos en similariteitsdetectieprogramma's in het algemeen te verbeteren. Deze experimenten richten zich voornamelijk op het evalueren van bestaande tools, het ontwikkelen van verbeterde algoritmen voor similariteitsdetectie, en het creëren van betere visualisaties om plagiaat op te sporen. Hoewel niet alle experimenten succesvol waren, hebben hun resultaten onmiskenbaar bijgedragen aan de kwaliteit en effectiviteit van Dolos.

Tot slot, in hoofdstuk 8, vatten we alles nog eens samen in een conclusie en geven we mogelijke verbeterpunten aan Dolos met opportuniteiten tot verder onderzoek. We bespreken de impact van Dolos, niet alleen in educatieve contexten, maar ook in andere gebieden zoals cybersecurity en onderzoek naar artificiële intelligentie. We sluiten dit hoofdstuk en dit proefschrift af met een kritische blik op de toekomst, met speciale aandacht voor de uitdagingen die generatieve artificiële intelligentie met zich meebrengt. Deze ontwikkelingen vragen om aandacht van studenten, lesgevers en onderzoekers.

Summary

The increased digitalisation of our society has caused more programming courses to be embedded in our curricula. However, learning to code is far from straightforward; this skill comprises various building blocks that build upon each other. If a student misses even one of these building blocks, for instance, due to an absent lesson, there is already a substantial chance that the student will fail to create a working solution for a programming assignment.

In such situations, the **pressure** to perform well, the **rationalisation** to catch up on the material later, and the **opportunity** to obtain a working solution elsewhere can lead to a student to decide to plagiarise source code. This form of academic dishonesty involves presenting source code from an external source as one's own work without citing the original source. Students often make minor adjustments to disguise their plagiarism.

Since this form of plagiarism is detrimental to the student's learning experience and undermines the evaluation process, some educators use similarity detection programs. These programs detect similarities between source codes that indicate plagiarism. Unfortunately, these programs are often outdated, inconvenient, and do not comply with modern privacy standards. Because of this, only few educators use these programs. To address this, we have developed a new similarity detection program called **Dolos**. Dolos aims to quickly and effectively detect similarities between source files submitted for a programming assignment. The results are displayed in a user-friendly interface that facilitates the search for plagiarism. Our goal is to make Dolos as accessible as possible without compromising flexibility for the users. This thesis describes the development of Dolos and explores the various aspects involved.

Dolos is named after the spirit of trickery and guile from Greek mythology.

Chapter 1 begins with an introduction to source code plagiarism. We explore what exactly this form of plagiarism entails, how frequently it occurs, what motivates students to commit plagiarism, and the techniques educators can employ to counteract it. Additionally, we discuss the various forms of source code plagiarism and the methods students use to modify their code to disguise plagiarism.

In chapter 2, we delve deeper into techniques to see through these disguises to detect similarities that may indicate plagiarism. We review a selection of the most commonly used similarity detection tools, including Moss, JPlag, Plaggie, Sherlock Warwick, Sherlock Sydney, and Compare50.

Subsequently, in chapter 3, we describe the algorithm underpinning Dolos. Dolos converts the examined source code into a syntax tree, a systematic representation of a program's underlying structure. This syntax tree is independent of whitespace and names that programmers can freely choose, elements that are easily modified to conceal plagiarism. The syntax tree is then transformed into digital fingerprints, which are compared across different source files. These fingerprints are robust against obfuscations that alter the underlying structure of the source code, such as rearranging the order of functions. Dolos aggregates the results of this similarity analysis into a digital report containing all the information needed to visually present it to the user.

Chapter 4 addresses the design and development of Dolos's user interface, with a strong emphasis on user experience and accessibility. We describe the various visualisations and how they can be utilised to detect plagiarism. Dolos keeps the results organised through interactive pages that display information about clusters and individual files, as opposed to pairwise comparisons that can quickly become overwhelming when dealing with extensive plagiarism. Compared to other programs, Dolos offers an enhanced user experience with multiple innovative visualisations.

In chapter 5, we describe the implementation of Dolos. Following good program design principles, Dolos is divided into several components, each with its own responsibilities. This modular software architecture provides great flexibility and enables support for different types of users. Dolos's functionalities are offered free of charge via a website¹, an online API, a CLI, and software libraries. One of Dolos's key features is the loose coupling between the algorithms and the support for various programming languages. By utilising a library that can generate syntax trees for multiple programming languages, Dolos supports a wide range of languages and makes it easy to add new ones. The source code of Dolos is freely available on github.com/dodona-edu/dolos.

Chapter 6 compares Dolos with other similarity detection tools across various aspects. Using usage statistics, we demonstrate that Dolos is increasingly being adopted by diverse types of users. Through several

¹dolos.ugent.be/server

benchmarks, we show that Dolos performs at least as well as, if not better than, existing alternatives in terms of performance, execution time, and memory usage. A user survey indicates that the user experience is positively rated. Finally, we describe how Dolos has become an essential component of our strategy to prevent and combat plagiarism in our own courses.

Chapter 7 discusses the various experiments conducted to improve Dolos and similarity detection programs in general. These experiments primarily focus on evaluating existing tools, developing improved algorithms for similarity detection, and creating better visualisations to detect plagiarism. Although not all experiments were successful, their results have undoubtedly contributed to the quality and effectiveness of Dolos.

Finally, in chapter 8, we summarise our conclusions and suggest potential improvements for Dolos, along with opportunities for further research. We discuss the impact of Dolos, not only in educational contexts but also in other areas such as cybersecurity and research on artificial intelligence. We conclude this chapter and thesis with a critical look into the future, paying special attention to the challenges posed by generative artificial intelligence. These developments require the attention of students, educators, and researchers alike.

Dankwoord — Acknowledgements

Wanneer een doctoraat wordt afgewerkt, is het vaak die éne kersverse doctor die in de schijnwerpers wordt geplaatst. Ik geloof echter oprecht dat het de mensen zijn die ons omringen, die ons toelaten om dit soort verwezenlijkingen te bereiken. Met dit dankwoord draag ik mijn doctoraat op aan jullie allen: familie, vrienden, collega's, kennissen, en alle andere medewezens waarmee we samenleven.

Zelfs de katten die mij gezegend hebben met hun kopjes verdienen een bedanking.

Wat volgt, is een poging om mijn dankbaarheid in woorden te vatten. Want zonder de berg aan begeleiding, de oceaan aan steun en de eindeloze stroom aan liefde die mij doorheen de jaren werd gegund, was ik nooit begonnen aan dit avontuur, laat staan dat ik het had volbracht.

De personen die het dichtst bij mijn doctoraat stonden, zijn ongetwijfeld mijn promotoren prof. Peter Dawyndt en prof. Bart Mesuere. Dankjewel om in mij te geloven, vanaf mijn eerste onzekere dagen als jobstudent tot de laatste punt achter mijn doctoraatsproefschrift. Jullie deur stond altijd open, en ik kreeg de tijd, de vrijheid en het vertrouwen om mijzelf uit te dagen als informaticus.

I would also want to thank my examination committee (“the jury”) to spend their time to critically evaluate this dissertation, give constructive feedback, and grill me with thoughtful questions. Thank you to prof. Chris Cornelis, prof. Veerle Fack, prof. Christophe Scholliers, prof. Frederik Gailly, and especially thanks to dr. Ari Korhonen for coming all the way to Ghent for both the internal and public defence. It is during these intense evaluations, when knowledge from different domains collide, that we truly extend the boundaries of our knowledge.

Ik ben ook ontzettend dankbaar voor alle lesgevers die mij met vuur en toewijding hebben gevormd: leerkrachten, assistenten, lectoren, coaches, professoren en sprekers die niet enkel kennis overdroegen, maar ook de passie en bevlogenheid voor hun vakgebied. Jullie hebben mij niet alleen opgeleid, maar ook geïnspireerd zelf lesgever te worden. In diezelfde adem bedank ik ook alle studenten voor hun scherpe vragen en verwarde blikken; jullie zijn mijn beste leraren die mij telkens opnieuw uitdagen om iets beter uit te leggen. Dat ene moment waarop

het dan “klikt” is mijn mooiste beloning en herinnert mij eraan dat onderwijs geen eenrichtingsverkeer is — *when one teaches, two learn*. In het bijzonder dank ik de masterproefstudenten en jobstudenten die hebben meegewerkt aan mijn onderzoek: Arne, Maxiëm, Maarten, Michiel en Raymond.

Het is dankzij het vertrouwen van de vakgroep dat ik als assistent mijn roeping kon volgen om de volgende generatie informatici mee op weg te zetten. Het is ook dankzij de fijne collega's binnen én buiten de vakgroep dat ik mij er helemaal thuis voel. De UGent is een bijzondere gemeenschap om in terecht te komen en het geeft voldoening om samen te werken met fantastische onderzoekers, professoren, secretariaatsmedewerkers, faculteitsraadsleden, studenten, resto-medewerkers, enzovoort.

Special thanks to all (ex-)colleagues for creating this wonderful atmosphere to work in, and to join on the extracurricular activities, notably the TWIST/WINST weekends. Alexis, Arne, Asmus, Bart, Benjamin, Boris, Charlotte, Dieter, Fatemeh, Felix, Francis, Heidi, Inga, Jari, Jonathan, Jorg, Karel, Linde, Louis, Louise, Maarten, Mustapha, Nico, Niko, Robbert, Pieter, Simon, Steven, Thijs, Thomas, Tibo, Tom, Toon, Wannes, and Wout, whom I am thankful to count among my friends.

I would also like to thank the wonderful people from Aalto University in Finland that welcomed me so warmly in their culture. Thank you to Archie, Artturi, Guido, Jaakko, Juho, Mikael, Otto, and all others that I had the honour to spend time with in Finland.

Een groep die een grote invloed gehad heeft op mijn studies is de studentenwerkgroep / hackerspace Zeus WPI. Jullie zijn écht met teveel om op te noemen, en elk van jullie brengt eigen, unieke inzichten en projectjes waar we tot in de vroege uurtjes over kunnen bezig zijn. Daarom wil ik ook alle oude, huidige, en toekomstige leden van Zeus bedanken om elkaar te blijven inspireren

Ik mag zeker niet vergeten om enkele oud-Zeusleden in het bijzonder te bedanken. Bedankt Feliciaan & Annelieke, Heidi & Felix, Stijn & Myrjam, Titouan & Deborah, en Tom voor de inzichtvolle discussies over alternatieve samenlevingsvormen en om samen diepgaande videoreportages te bekijken over de mysteries in de menselijke psyche.

Bedankt ook aan de medestudenten die doorheen mijn opleiding mijn vrienden voor het leven geworden zijn en met wie ik vele leuke herinneringen heb van LAN-parties, huwelijken, trektochten, *game jams* en D&D-sessies: Arne & Liesbeth, Jarre, Jorg & Louise, Niko, Nils & Ciel, Pieter, Sam & Chloë en Sander.

As warm as a
sauna.

Ook met het zootje ongeregeld van Theater Volta deel ik prachtige herinneringen die ik koester: Astrid, Bert, Charlotte, Filip, Jan, Jensen, Lisa, en Louise, bedankt voor jullie artistieke kronkels en wonderbaarlijke apocalypsen.

An unlikely, but magnificent group of friends also deserve a place in this section: Aline & Sarah, Demi, Elena & Ludwig, Ilya & Galina, and Sigurd & Elynn, thank you for the epic stories we create together.

Daarnaast heb ik het grootste geluk om in een bijzonder hartelijke familie te zijn opgegroeid. Bedankt aan mijn fantastische grootouders, tantes, nonkels, neven, nichten, en ook mijn schoonfamilie, om mij te omringen met zoveel warmte. In het bijzonder bedankt aan mijn mama & papa, mijn zussen Marthe & Fauve, en schoonbroers Dereje & Remi. Dank jullie allemaal voor dit nest vol liefde, gezelligheid en geduld, waarin ik zorgeloos kon opgroeien tot wie ik ben.

Dan rest mij nog één iemand in het bijzonder om te bedanken, en dat is mijn beste vriendin, lief, en vrouw Charlotte. Dank je voor al onze zotte avonturen samen, ik kijk uit naar alles wat we samen nog zullen beleven.

Table of contents

Samenvatting	iii
Summary	vii
Dankwoord — Acknowledgements	xi
Table of contents	xv
List of acronyms	xix
List of publications	xxi
1. Educational source code plagiarism	1
1.1. Educational Setting	2
1.1.1. Formative and summative assessment	2
1.1.2. Programming exercise platforms	3
1.1.3. Cheating	3
1.2. Source code plagiarism	4
1.2.1. Prevalence of source code plagiarism	5
1.2.2. When does plagiarism occur in programming assignments?	5
1.3. How students plagiarise	9
1.3.1. Sources for plagiarism	9
1.3.2. Obfuscations: how students evade detection	11
1.4. Countermeasures	14
1.4.1. Plagiarism detection using Source Code Similarity	15
1.5. Goal and structure of this dissertation	16
1.5.1. Research goal	16
1.5.2. Structure of this dissertation	17
2. Related work	19
2.1. Detecting plagiarism	20
2.2. Plagiarism detection fundamentals	21
2.3. Similarity detection algorithms	22
2.3.1. Greedy String Tiling	23
2.3.2. Winnowing	23

2.4.	Source code similarity detection tools	24
2.4.1.	Moss	26
2.4.2.	JPLag	27
2.4.3.	Plaggie	27
2.4.4.	Sherlock Warwick	27
2.4.5.	Sherlock Sydney	28
2.4.6.	Compare50	29
3.	Algorithmic underpinnings	31
3.1.	Tokenisation	33
3.1.1.	Parsing to a syntax tree	33
3.1.2.	Serialisation and location mapping	35
3.1.3.	Removing comment tokens	36
3.2.	Fingerprinting	37
3.2.1.	Hashing tokens	39
3.2.2.	Hashing k -grams	40
3.2.3.	Winnowing	41
3.2.4.	Building the fingerprint index	44
3.3.	Reporting	45
3.3.1.	Comparing pairs	46
3.3.2.	Computing similarity	47
3.3.3.	Computing the Longest Common Substring	48
3.3.4.	Delayed calculation of fragments	48
4.	User Interface and User Experience design	51
4.1.	Design methodology	52
4.1.1.	Usability Testing	53
4.1.2.	Philosophy	53
4.2.	General UI structure	56
4.2.1.	Navigation	56
4.2.2.	Metadata	58
4.2.3.	Global settings	58
4.3.	Overview	59
4.3.1.	Similarity Histogram	61
4.3.2.	Automatic similarity threshold estimation	62
4.4.	Plagiarism Graph	63
4.5.	Clusters	65
4.5.1.	Cluster detail	65
4.6.	Pairs	68
4.6.1.	Pairwise comparison	69
4.7.	Submissions	72
4.7.1.	Submission detail	73
4.8.	Evolution of the UI	73
4.8.1.	TUI before v1.0.0	75

4.8.2.	v1.0.0 – A web UI for Dolos	75
4.8.3.	v1.6.0 – Clusters and files	78
4.8.4.	v2.0.0 – Major UI redesign	80
4.8.5.	Further development	80
5.	Implementation	83
5.1.	Software architecture	83
5.1.1.	Choice for TypeScript	85
5.1.2.	Repository structure	86
5.1.3.	Continuous Integration and Deployment	86
5.1.4.	Software Licence	88
5.2.	Parser module	88
5.2.1.	Vendoring parsers	90
5.3.	Software libraries	91
5.3.1.	dolos-core	91
5.3.2.	dolos-lib	95
5.4.	Command-line interface	98
5.4.1.	CSV-format	99
5.4.2.	Launching the Web UI	99
5.5.	Web interface	100
5.5.1.	Vue philosophy	100
5.5.2.	Report ingestion and initialisation	102
5.5.3.	D3 Visualisations	102
5.5.4.	The Monaco editor	105
5.5.5.	Server mode	106
5.6.	API server	107
5.6.1.	API submission flow	108
5.6.2.	External integrations	109
5.7.	Additional components	113
5.7.1.	Documentation	113
5.7.2.	Samples	115
5.7.3.	Containers	116
5.7.4.	Nix flake	117
6.	Evaluation	119
6.1.	Usage metrics	119
6.2.	Benchmarks	122
6.2.1.	Datasets	122
6.2.2.	Method	125
6.2.3.	Results	126
6.3.	Usability and User Experience	134
6.3.1.	User Experience Questionnaire	134
6.3.2.	Methodology	135
6.3.3.	Results	135

6.3.4. Limitations	137
6.4. Case study	138
6.4.1. Course structure	138
6.4.2. Plagiarism prevention	140
6.4.3. Impact of COVID-19 pandemic	143
6.4.4. Impact of GenAI	145
7. Experimental prototypes	151
7.1. Evaluation	151
7.1.1. Challenges	152
7.1.2. Dataset annotation and benchmark standardisation	153
7.1.3. Simulated plagiarism dataset	155
7.1.4. Lessons learned	157
7.2. Matching algorithms	157
7.2.1. Tree-matching	158
7.2.2. Syntax tree preprocessing	158
7.2.3. Suffix trees	160
7.2.4. Lessons learned	162
7.3. Visualisations	163
7.3.1. Interestingness metric	163
7.3.2. Semantic analysis	163
7.3.3. Lessons learned	164
8. Conclusions	167
8.1. Results	168
8.1.1. Research contributions	169
8.2. Impact	170
8.2.1. Plagiarism detection in education contexts	171
8.2.2. Detecting plagiarism by LLMs	171
8.2.3. Malware classification	173
8.3. Future work	174
8.3.1. Multi-file and multi-submission analysis	174
8.3.2. Improving the similarity threshold estimate	176
8.3.3. Supporting instructors to report plagiarism	177
8.4. Generative AI	178
8.5. Concluding remarks	180
A. Dolos CLI commands and options	181
A.1. <code>dolos</code> command-line options	181
A.2. <code>dolos serve</code> command-line options	181
A.3. <code>dolos run</code> command-line options	182
Bibliography	185

List of acronyms

AI	Artificial Intelligence
API	Application Programming Interface
AST	Abstract Syntax Tree
BYOD	Bring-Your-Own-Device
CLI	Command-Line Interface
CS	Computer Science
CSS	Cascading Style Sheets
CST	Concrete Syntax Tree
CSV	Comma-Separated Values
FOSS	Free and Open-Source Software
GenAI	Generative Artificial Intelligence
GHCR	GitHub Container Registry
GLR	Generalised left-to-right, rightmost derivation in reverse (parser)
GPL	GNU General Public License
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoC	Indicator of Compromise
IP	Internet Protocol

List of acronyms

IR	Intermediate Representation
JSON	JavaScript Object Notation
LCS	Longest Common substring
LLM	Large Language Model
LMS	Learning Management system
LR(1)	Canonical left-to-right, rightmost derivation in reverse (parser)
LTI	Learning Tools Interoperability
MIT	Massachusetts Institute of Technology
MLE	Machine Learning Engineering
MVC	Model-View-Controller
MVVM	Model-View-Viewmodel
OS	Operating System
RKR-GST	Running Karp-Rabin Greedy-String Tiling
SDK	Software Development Kit
SFC	Single-File Component
SOCO	Source Code re-use
SPA	Single-Page Application
SVG	Scalable Vector Graphics
TNR	True Negative Rate
TPR	True Positive Rate
TUI	Terminal User Interface
UEQ	User Experience Questionnaire
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
WASM	WebAssembly

List of publications

Included in this dissertation

List of publications directly related to this dissertation and for which I am the first author. As such, the contents of these publications are included in this dissertation.

Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt and Bart Mesuere (Mar. 2022). “Dolos: Language-agnostic Plagiarism Detection in Source Code”. In: *Journal of Computer Assisted Learning* 38.4, pp. 1046–1061. issn: 1365-2729. doi: [10.1111/jcal.12662](https://doi.org/10.1111/jcal.12662).

Rien Maertens, Peter Dawyndt and Bart Mesuere (June 2023). “Dolos 2.0: Towards Seamless Source Code Plagiarism Detection in Online Learning Environments”. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2. ITiCSE 2023*. New York, NY, USA: Association for Computing Machinery, p. 632. isbn: 979-8-4007-0139-9. doi: [10.1145/3587103.3594166](https://doi.org/10.1145/3587103.3594166).

Rien Maertens, Maarten Van Neyghem, Maxiem Geldhof, Charlotte Van Petegem, Niko Strijbol, Peter Dawyndt and Bart Mesuere (May 2024). “Discovering and Exploring Cases of Educational Source Code Plagiarism with Dolos”. In: *SoftwareX* 26, p. 101755. issn: 2352-7110. doi: [10.1016/j.softx.2024.101755](https://doi.org/10.1016/j.softx.2024.101755).

Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Maarten Van Neyghem, Maxiem Geldhof, Arne Carla Jacobs, Peter Dawyndt and Bart Mesuere (Oct. 2024). *Dolos*. Zenodo. doi: [10.5281/zenodo.7966722](https://doi.org/10.5281/zenodo.7966722).

Rien Maertens, Peter Dawyndt and Bart Mesuere (June 2025). “Source Code Plagiarism Detection as a Service with Dolos”. In: *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 2. ITiCSE 2025*. New York, NY, USA: Association for Computing Machinery, pp. 729–730. isbn: 979-8-4007-1569-3. doi: [10.1145/3724389.3731274](https://doi.org/10.1145/3724389.3731274).

With Team Dodona

List of publications by Team Dodona for which I am a co-author.

Charlotte Van Petegem, Louise Deconinck, Dieter Mourisse, Rien Maertens, Niko Strijbol, Bart Dhoedt, Bram De Wever, Peter Dawyndt and Bart Mesuere (Mar. 2023). “Pass/Fail Prediction in Programming Courses”. In: *Journal of Educational Computing Research* 61.1, pp. 68–95. issn: 0735-6331. doi: [10.1177/07356331221085595](https://doi.org/10.1177/07356331221085595).

Niko Strijbol, Charlotte Van Petegem, Rien Maertens, Boris Sels, Christophe Scholliers, Peter Dawyndt and Bart Mesuere (May 2023). “TESTed—An Educational Testing Framework with Language-Agnostic Test Suites for Programming Exercises”. In: *SoftwareX* 22, p. 101404. issn: 2352-7110. doi: [10.1016/j.softx.2023.101404](https://doi.org/10.1016/j.softx.2023.101404).

Charlotte Van Petegem, Peter Dawyndt and Bart Mesuere (June 2023). “Dodona: Learn to Code with a Virtual Co-teacher That Supports Active Learning”. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2. ITiCSE 2023*. New York, NY, USA: Association for Computing Machinery, p. 633. isbn: 979-8-4007-0139-9. doi: [10.1145/3587103.3594165](https://doi.org/10.1145/3587103.3594165).

Charlotte Van Petegem, Rien Maertens, Niko Strijbol, Jorg Van Renterghem, Felix Van der Jeugt, Bram De Wever, Peter Dawyndt and Bart Mesuere (Dec. 2023). “Dodona: Learn to Code with a Virtual Co-Teacher That Supports Active Learning”. In: *SoftwareX* 24, p. 101578. issn: 2352-7110. doi: [10.1016/j.softx.2023.101578](https://doi.org/10.1016/j.softx.2023.101578).

Charlotte Van Petegem, Kasper Demeyere, Rien Maertens, Niko Strijbol, Bram De Wever, Bart Mesuere and Peter Dawyndt (Apr. 2024). *Mining Patterns in Syntax Trees to Automate Code Reviews of Student Solutions for Programming Exercises*. doi: [10.48550/arXiv.2405.01579](https://doi.org/10.48550/arXiv.2405.01579). arXiv: [2405.01579 \[cs\]](https://arxiv.org/abs/2405.01579).

With Team Unipept

List of publications by Team Unipept for which I am a co-author.

Felix Van der Jeugt, Rien Maertens, Aranka Steyaert, Pieter Verschaffelt, Caroline De Tender, Peter Dawyndt and Bart Mesuere (June 2022). “UMGAP: The Unipept MetaGenomics Analysis Pipeline”. In: *BMC Genomics* 23.1, p. 433. issn: 1471-2164. doi: [10.1186/s12864-022-08542-4](https://doi.org/10.1186/s12864-022-08542-4).

Chapter 1.

Educational source code plagiarism

We find ourselves in a thrilling epoch where technological advancements unfold at a breathtaking pace. Humans have demonstrated the capacity to develop vaccines for novel diseases in under a year (European Commission 2021), sustain nuclear fusion (McGrath 2024), and generate realistic videos from a single descriptive sentence (OpenAI 2024). These remarkable strides are underpinned by our ability to program computational machines. Through meticulous selection of program instructions, we can task computers with performing intricate biological analyses, simulating interactions between elementary particles, and construct sophisticated deep learning algorithms that capture the nuanced relationship between text and images.

However, as potent as programming is, mastering it is far from straightforward (Robins et al. 2003; Simões and Queirós 2020). Confronted with the formidable challenge of learning to program in computer science courses, students may succumb to the temptation to cheat (Sheard, Carbone et al. 2003). One form of academic dishonesty in this context is *source code plagiarism* (Cosma and Joy 2008), where individuals submit others' code as their own.

This dissertation addresses this issue by introducing Dolos, a modern source code similarity detection system. We commence this introductory chapter by delineating the educational setting in which source code plagiarism is observed (section 1.1). Subsequently, section 1.2 details the intricacies of source code plagiarism, exploring its definition, prevalence, and the factors contributing to this form of academic misconduct. Section 1.3 examines methods students employ to plagiarise, while section 1.4 discusses potential strategies to mitigate this behaviour. Finally, section 1.5 clarifies the overarching goal and structure of this dissertation.

1.1. Educational Setting

Introductory programming courses adopt a “learning by doing” approach, where students acquire programming skills by tackling **programming exercises**. Each programming exercise comprises an assignment that outlines a problem, which students must address by coding a solution. These exercises progressively increase in difficulty throughout the course, touching on more and more programming language concepts and presenting increasingly challenging problems.

To support students in completing numerous programming exercises, educators leverage automated assessment platforms. These platforms provide students with immediate feedback on their submissions (Van Petegem, Dawyndt et al. 2023)

Instructors typically set a **deadline** for a programming exercise, requiring students to submit their solutions by a specified date and time. For smaller exercises, this solution might consist of a single source file, while larger projects might necessitate multiple source code files. Automated assessment systems often allow students to submit multiple attempts, receiving feedback during both formative and summative assessments (section 1.1.1).

We call it a solution, but not all student solutions successfully solve the problem.

We will refer to a student’s (temporary or attempted) solution for a programming exercise as a **submission**. The final submission before the deadline implicitly represents the students’ solution to the given exercise.

1.1.1. Formative and summative assessment

The usage of programming assignments can be categorised into two distinct types:

Formative assessment: These are regular assessments designed to monitor student learning, with instructors providing feedback to enhance the learning process. Students are often encouraged to collaborate in small groups (Walker 1997; Williams et al. 2003). This form of assessment is low-stakes, as it typically carries little to no weight in the final grade. Regular programming exercises solved during hands-on practical sessions are an example of this form of assessment.

Summative assessment: These evaluations assess student learning, often at the end of a course. Students are typically required to work independently, where external assistance from instructors or peers is prohibited. Summative assessments significantly contribute to the

final grade and are thus high-stakes. Examples include evaluations, exams, or projects.

1.1.2. Programming exercise platforms

Typically, teachers in programming education organise both formative and summative assessments using a programming exercise platform. This platform is a learning management system (LMS) where teachers can manage their classroom and course contents. In addition to course management, a programming exercise platform allows students to submit their programming solution, evaluate this solution against a test suite, and receive automated or manual feedback.

The Dodona Platform

Dodona¹, is a programming exercise platform developed in 2016 at Ghent University (Van Petegem, Maertens et al. 2023). The platform focuses on qualitative and timely automated feedback towards students. Dodona employs a strict separation between the platform (managing courses, students and teachers), the exercises (describing the programming assignments), and the judges (the testing framework deciding whether submissions are correct). Both exercises and judges are stored in version-controlled repositories. This allows teachers to maintain ownership over their course content while still being able to share this content with their students and other teachers through Dodona.

The research on source code similarity detection described in this dissertation is part of the larger research group around Dodona. Previous research by Team Dodona involved a language-agnostic testing process for programming exercises (Strijbol et al. 2023), and advanced techniques supporting manual feedback to speed up the grading process (Van Petegem, Demeyere et al. 2024).

1.1.3. Cheating

Cheating transpires when students attempt to subvert the assessment to gain an unfair advantage. This behaviour is also referred to as academic misconduct or a breach of academic integrity. Dick et al. (2002) enumerate a non-exhaustive list of 53 cheating methods, including plagiarism, bribing staff, data manipulation, and using cheat sheets

¹dodona.be

during exams. Cheating is a critical issue because it not only undermines the student's own learning process (Lupton et al. 2000) but it also harms society, the profession, the reputation of the academic instruction, and the value of the degree being pursued (Clarke and Lancaster 2013; Dick et al. 2002).

What constitutes cheating behaviour varies widely, and the perceptions of academic misconduct may differ between students and staff (Brimble and Stevenson-Clarke 2005). One factor influencing the severity of cheating is the form of assessment. During summative assessments, instructors rigorously monitor for cheating, as it compromises the evaluation of a student's acquired skills. The higher stakes in such assessments may also amplify the temptation to cheat (McCabe, Trevino et al. 1999). When caught cheating during an exam, students often face severe consequences, as this is deemed serious misconduct.

In contrast, formative assessments focus on student learning, leading to a more lenient approach for instructors. In this low-stakes environment, while instructors may monitor some forms of cheating to safeguard the learning experience of all, the repercussions for misconduct are typically minimal.

1.2. Source code plagiarism

This dissertation addresses a specific form of plagiarism in programming courses known as *source code plagiarism*. Cosma and Joy (2008) define source code plagiarism as:

Source-code plagiarism occurs when students reuse source code authored by someone else, either intentionally or unintentionally, and fail to adequately acknowledge the fact that the particular source-code is not their own.

In practice, however, it remains challenging to distinguish between harmless code reuse and source code plagiarism. Factors considered in this distinction are similarity, rules, and intent (Gibson 2009; Simon, Sheard et al. 2016). One such “grey area” is self-plagiarism, where a student reuses their own work previously submitted for other assignments (Collberg and Kobourov 2005).

Perceptions of what constitutes plagiarism and its severity can vary significantly among students, staff, and cultures (Brimble and Stevenson-Clarke 2005; Husain et al. 2017; Sheard, Dick et al. 2002). It is therefore imperative that institutions and instructors clearly communicate their

definitions and expectations regarding plagiarism (McCabe, Treviño et al. 2002; Wager 2014).

1.2.1. Prevalence of source code plagiarism

A report from Stanford University, aggregating a decade of academic misconduct cases, revealed that 37% of these incidents involved computer science students, despite computer science enrollments constituting only 6.5% of the total student population (Roberts 2002). Alam (2004) found that 35% of students self-reported plagiarising on programming assignments, identifying this form of assessment as having the highest prevalence of plagiarism in this study. Similarly, a survey conducted at a Slovakian university reported that 33% of responding students admitted copying and modifying source code for at least one of their classes (Chuda et al. 2012).

Sheard, Dick et al. (2002) investigated academic misconduct at two Australian universities, finding that 33.6% and 28.2% of computer science students self-reported copying and modifying a friend's assignment. Notably, a follow-up survey conducted a decade later, after the implementation of countermeasures, observed a decrease to 21% of students engaging in such behavior (Sheard and Dick 2011). We discuss some of these countermeasures in section 1.4.

1.2.2. When does plagiarism occur in programming assignments?

Understanding the motivation behind certain behaviours is crucial for their prevention. Consequently, numerous academics have endeavoured to identify the factors contributing to student plagiarism. In their review on plagiarism in programming assessments, Albluwi (2019) applies the *Fraud Triangle* framework to categorise and analyse research contributions. This model, used in the financial sector to determine under which circumstances fraud could occur, was initially described by Cressey (1953), and has since been integrated into official auditing standards. The model comprises three elements, visualised in figure 1.1: perceived *pressure* that necessitates fraud, an *opportunity* that the individual perceives to be exploitable, and a *rationalisation* that justifies the fraudulent actions as ethical.

Each of these elements has been extensively studied to identify contributing factors that could mitigate the likelihood of plagiarism.

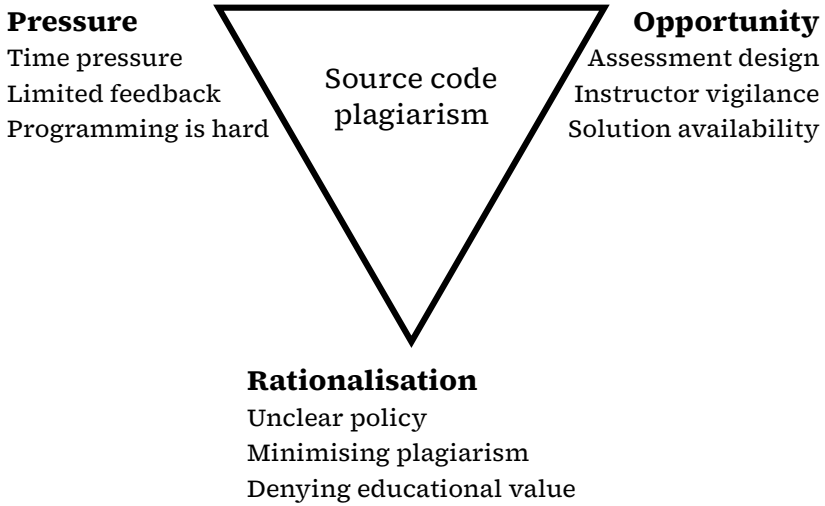


Figure 1.1. Diagram symbolising the three factors in the *Fraud Triangle* framework: pressure, opportunity, and rationalisation.

Pressure

The *pressure* to commit plagiarism encapsulates the reasons or the needs that drive individuals to consider such actions. Students who perceive a lack of skills to achieve satisfactory grades experience greater pressure to cheat than those confident in their abilities. This factor remains the least studied amount the three categories of the Fraud Triangle (Albluwi 2019).

One hypothesis for this research gap could be widespread perception that programming is inherently a challenging skill to master (Robins et al. 2003; Simões and Queirós 2020). Robins (2010) highlights the high interdependence of knowledge required for programming, where missing even one seminar can significantly diminish a student's chances of success. Students corroborate this, citing the cumulative nature of programming, along with equipment failures and software issues, as external factors contributing to their struggles in programming courses (Sheard, Carbone et al. 2003). Additionally, Luxton-Reilly and Petersen (2017) observe:

Students with fragile knowledge in any of the areas required may be unable to produce a working solution, even when they may know most of the required material.

Fragile knowledge refers to limited or incorrect knowledge that stu-

dents cannot apply effectively (Perkins and Martin 1985). Consequently, a single technical issue or missed class can cause a student to fall behind, leading to a cascade of challenges that hinder their ability to achieve a passing grade.

Time pressure and high workload are also significant factors contributing to cheating (Sheard and Dick 2012). Programming courses often employ weekly graded assignments, thus allowing students only limited time for each task. A rapidly approaching deadline, coupled with the inability to produce a working solution, can motivate students to cheat. Some instructors mitigate this pressure by sending an email close to this deadline reminding students to seek help from them, rather than from peers (Sheard, Simon et al. 2017).

Students may also feel additional pressure to cheat based on automated feedback: Kyrilov and Noelle (2015) found that students receiving binary feedback (correct or incorrect) plagiarised twice as many exercises on average compared to those receiving no feedback. They hypothesise that binary feedback provides insufficient information for novice programmers to correct their mistakes, potentially demotivating them. They advocate for more detailed feedback in automated assessment systems (Kyrilov and Noelle 2016).

Opportunity

The perceived *opportunity* to cheat successfully without detection significantly contributes to plagiarism. Assignment design plays a pivotal role in the number of sources a student can plagiarise a solution from. Ideally, each student would receive individualised assignments tailored to their learning trajectory. However, this approach demands considerable resources from instructors and complicates equitable evaluation. Consequently, all students in a programming course typically receive the same assignments, facilitating the sharing of solutions and enabling submissions of others' work, including solutions from previous semesters.

Even with individualised assignments, students can engage in contract cheating by having someone else complete the assignment. The advent of Large Language Models (LLMs) has exacerbated this issue, making it easier to generate a solutions using Generative Artificial Intelligence (GenAI).

While accessing a valid solution may be straightforward, submitting it as one's own without detection is more challenging. Handing in a plagiarised submissions unaltered carries a high risk of discovery

and prosecution. Consequently, students often modify the original solution to conceal their misconduct, requiring confidence in their ability to evade plagiarism detection systems. However, finding an applicable solution and modifying it still requires less skill than solving the programming assignment independently. We will discuss how students plagiarise in section 1.3.

The instructor's attitude towards plagiarism also influences the opportunity to cheat. If students perceive that an instructor is indifferent to cheating, they may be tempted to engage in such behaviour (Chuda et al. 2012). Unfortunately, only a minority of instructors actively check for plagiarism (Chuda et al. 2012; Lancaster and Culwin 2004; Simon and Sheard 2016), with some even ignoring it when discovered (Coren 2011). Instructors often cite insufficient time as a reason for overlooking plagiarism, as addressing it requires constant vigilance, careful examination of suspected cases, and thorough documentation of evidence (Coren 2011).

Rationalisation

Individuals rarely perceive their own actions as wrong. Those who plagiarise often rationalise their behaviour as acceptable. A significant motivation might be the genuine belief that their actions are permissible. Even among instructors, there is disagreement about what constitutes cheating or how it should be penalised, if at all. Consequently, it is unsurprising that students are often unclear about the boundaries. It is the instructor's and the institution's responsibility to provide a clear policy that explicitly and unambiguously defines acceptable and unacceptable behaviour.

Even with a clear policy, students can be remarkably creative in rationalising plagiarism. In source code plagiarism, a student might believe it is acceptable to submit a modified solution as long as they understand the program's functionality (Simon, B. Cook et al. 2014). Other rationalisations include minimising the extent of plagiarism ("It was only a few lines" or "I only used it as inspiration") and denying the educational value ("I don't need this course", "Professionals do this all the time") (Chuda et al. 2012; Dick et al. 2002; Simon, B. Cook et al. 2013, 2014).

While a policy can objectively define the rules, it is not a foolproof countermeasure. Researches argue that the most effective way to address the rationalisation of plagiarism is to promote academic integrity among students. Sheard and Dick (2012) found that emphasising personal integrity could reduce plagiarism.

1.3. How students plagiarise

Plagiarism of source code is inherently multi-faceted, involving diverse sources and various techniques to conceal the misconduct, depending on the origin of the plagiarised material. The approach to detecting plagiarism hinges significantly on the source and methods employed to obscure it, with certain forms of source code plagiarism proving more challenging to identify than others.

The original solution a student plagiarises does not necessarily need to be correct. Students may assume that submitting a flawed solution will yield better grades than submitting nothing at all. Additionally, students might introduce errors themselves to correct solutions, either inadvertently while attempting to obfuscate the code but failing to preserve its functionality, or intentionally, under the belief that instructors may only scrutinise correct submissions for plagiarism.

1.3.1. Sources for plagiarism

To commit plagiarism, a student must access an existing solution for the given programming assignment. The sources of the original source code can be categorised as follows:

Students in the same course

A common form of plagiarism involves using the source code of another student in the same course. Students may request solutions from peers or share their submissions openly. Additionally, students might divide programming exercises among themselves, solving only their share and sharing the results to reduce effort.

Instructors grading final submissions are likely to notice identical submissions, prompting students to apply obfuscations to differentiate their derivative copies from the original. We discuss these obfuscations in section [1.3.2](#).

Collusion

A related yet distinct form of plagiarism occurs when students collaborate without official approval and submit the results as individual work, often termed collusion (Jones et al. [2008](#)). This can easily transpire

when students work together during formative assessments. Each collaborating student might apply obfuscations to make their submission appear distinct, but the final results will remain highly similar.

Submissions from previous sessions

Many instructors reuse the same assignments across multiple offerings of a course (Gehringer 2004), creating an opportunity for students to submit solutions from previous sessions. Students may access these solutions from peers who have already taken the course, those retaking it, or from online postings. This can even include reference solutions provided by the instructor in previous sessions.

Themselves

When a student submits code from previously submitted work, either from another course or a previous session when retaking the course, this practice is known as **self-plagiarism** (Collberg and Kobourov 2005). Self-plagiarism is a contentious issue; not all instructors consider it academic misconduct, and some even view it a good practice (Simon, Sheard et al. 2016). Moreover, students consider self-plagiarism the most acceptable form of cheating (Joy, Cosma et al. 2011).

External sources

A significant source of plagiarism is external sources, both online (discussion boards, online encyclopedias, source code repositories) or offline (textbooks). In introductory programming courses, instructors often use well-known exercises with numerous applicable solutions available online. For example, mergesort, a popular sorting algorithm that most computer science students will program at least once as part of an assignment, yields over 7 000 repositories on GitHub, providing ample sources for plagiarism.

This form of plagiarism can be challenging, as not all available solutions are of sufficient quality, and students must modify them to fit the current assignment. Instructors can design assignments to increase the difficulty of incorporating existing solutions (Simon 2017).

Contract cheating

Another manifestation of plagiarism is **contract cheating**, wherein a student engages someone else to complete the assignment, often for compensation (Clarke and Lancaster 2006, 2013). Specialised sites exist to offer and bid on services to outsource solving the programming assignments at hand (D’Souza et al. 2007). The solution handed in is in this case an original and unique solution, unless the contractor themselves plagiarised their solution. However, the author of the program is not the student tasked with the assignment. Luckily, this is one of the least practiced forms of plagiarism (Brimble and Stevenson-Clarke 2005) and generally considered unacceptable by students (Simon, Sheard et al. 2016).

Generative AI

The recent rise of **GenAI**, particularly LLMs, has democratised a practice similar to contract cheating by enabling the generation of programming assignment solutions free of charge. The accessibility of GenAI has led to a consensus that it is “unbannable” (Prather, Leinonen et al. 2025). Consequently, only a minority of educators explicitly disallow GenAI, with some purposefully integrating it into their courses.

Most students (95%) consider generating a solution with GenAI and submitting it without understanding to be unethical. A smaller majority (60%) also believe that understanding the solution is insufficient (Prather, Denny et al. 2023). Initial research suggests that GenAI is increasing the prevalence of plagiarism in programming assignments and shifting the method of plagiarism towards GenAI (B. Chen et al. 2024).

1.3.2. Obfuscations: how students evade detection

Students engaging in source code plagiarism often modify the original source using **obfuscations** to disguise their academic misconduct (Faidhi and Robinson 1987). Novak et al. (2019) identified 16 distinct obfuscation methods, categorised into four groups: lexical changes, structural changes, advanced structural changes, and logical changes. These methods are arranged according to a “pyramid of program modification levels” (G. Chen et al. 2011), reflecting increasing complexity and impact on the original source code.

The rationale behind this ordering is that students resorting to plagiarism will mostly do so due to fragile knowledge; they lack the skills to complete the programming assignment successfully. Consequently, they apply modifications within their capabilities. Adding comments to a program, for instance, is less challenging than rewriting a `for`-loop into a `while`-loop.

While it is always possible to transform one code fragment into another using these obfuscations, the ability to do so does not necessarily imply plagiarism. However, if two programs differ only by minor, low-complexity changes, this may strongly indicate plagiarism, warranting further inspection.

We will discuss the 16 obfuscations across four categories:

Lexical changes

Lexical changes require minimal knowledge of the programming language and can be rapidly applied using basic text editor functionalities like search and replace. These modifications do not alter the program's structure.

- **OM_01_L: Visual code formatting** involves adding whitespace such as newlines, spaces, and indentation.
- **OM_02_L: Comments modification** includes adding, removing, or altering comments.
- **OM_03_L: Translation of program parts** entails translating sections of the program from one natural language to another.
- **OM_04_L: Modifying program output** involves changes to the program's output, including alterations to a Graphical User Interface (GUI).
- **OM_05_L: Identifier rename** includes changing identifiers such as variable names, class names, or function names.
- **OM_06_L: Changing constant values** involves altering constants like numbers or strings.

Structural changes

Structural changes require a basic understanding of the programming language and involve modifying one or two lines of code.

- OM_07_S: **Reordering independent lines of code** alters the execution order of program statements but necessitates identifying which lines can be safely reordered without affecting program semantics.
- OM_08_S: **Adding redundant lines of code** involves inserting statements that do not meaningfully influence the program's result, such as adding print statements or assigning a variable to itself.
- OM_09_S: **Splitting up lines of code** modifies a program statement by splitting it into two distinct statements, such as transforming a return statement with an expression into storing the expression's result in a variable and returning that variable.
- OM_10_S: **Merging lines of code** is the inverse of splitting lines of code, such as combining a variable assignment with its declaration.

Advanced structural changes

Advanced structural changes demand a deeper understanding of the programming language and typically involve modifications to multiple lines of code. These changes require careful consideration of program flow to maintain functionality. Many modern integrated development environments (IDEs) offer automated refactoring tools to facilitate these modifications.

- OM_11_AS: **Changing statement specification** involves altering operations in expressions, such as transforming `x != y` to `!(x == y)`, and modifying data types or modifiers.
- OM_12_AS: **Replacing control structures with equivalents** such as converting a `while` loop into a `for` loop.

Logical changes

Logical changes are the most drastic changes, requiring thorough understanding of both the programming language and the code fragment.

They are the most complex obfuscation to apply, but also the most challenging to detect, as the resulting program significantly differs from the original.

- OM_13_LG: **Simplifying the code** involves removing no-essential statements and functions or even essential parts that the student does not understand and thus prefers not to submit.
- OM_14_LG: **Translation from another programming language** is an obfuscation often applied when plagiarising from an external source where the desired program is unavailable in the required programming language.
- OM_15_LG: **Changing the logic** is the most complex obfuscation, as it requires the skill to modify the original code to meet the students' specific needs. This occurs when the original program does not fully address the task at hand.
- OM_16_LG: **Combining copied and original code** includes integrating parts of another submission into a program where the student is the original author. When a student attempts to create a program, but struggles with certain parts, they may plagiarise only the sections needed to complete their own work.

1.4. Countermeasures

Having established the nature of source code plagiarism, the motivations behind it, and the methods employed, we can now explore strategies to mitigate this issue. We summarise effective approaches reported by Albluwi (2019) and Sheard, Simon et al. (2017):

Most countermeasures aim to diminish the perceived **opportunity** to cheat. As an initial approach, instructors can vary assignments to eliminate the possibility of handing in previous solutions. Assignments can be varied across semesters or even randomised or personalised for each student. Secondly, interviewing students as part of the grading process or evaluating their understanding of the assignment might reveal discrepancies between the student's understanding and their submitted solution. A third approach centres on detection, either by monitoring contact cheating websites for relevant requests or observing student progress for suspicious behaviour. Employing plagiarism detection tools, and informing students about their use is also deemed effective. This strategy can be enhanced by maintaining a

database of previous submissions and including them in the plagiarism detection analysis. Finally, raising awareness of the consequences of cheating, if they are sufficiently severe, could deter students from cheating.

To reduce **rationalisation**, educating students about ethics and academic integrity is paramount. Requiring students to explicitly commit to academic integrity policies and clearly communicating what is permissible can reinforce this education. It is also beneficial to focus on positive alternatives: teaching students when and how to cite code, motivating them with engaging assignments, providing sufficient resources and tools to complete assignments, and building relationships with students can further reduce rationalisation.

To alleviate **pressure**, lowering the stakes of assessments and ensuring adequate support, especially around deadlines, can be beneficial. Instead of offering only binary automated feedback, instructors should strive to provide actionable feedback equips students with the right tools to improve their solution.

Additionally, Albluwi (2019) presents a list of questions to aid instructors in assessing the risk of plagiarism in their course. Using this list and the strategies listed above, instructors can compliment their course with a suitable plagiarism detection and prevention strategy.

1.4.1. Plagiarism detection using Source Code Similarity

Plagiarism detection in source code involves identifying plagiarised source code despite various obfuscation modifications (Novak et al. 2019). Tools facilitate this process by measuring source code similarity between programs and flagging suspicious similarities. Instructors use these findings as clues when identifying plagiarism.

Tools for detecting similarity have been available for nearly 50 years (Ostenstein 1976), yet their adoption remains limited. Culwin et al. (2001) reported that only 26% of computer science instructors in higher education in the United Kingdom used plagiarism detection tools. Similarly, just 8% of Slovakian computer science instructors surveyed by Chuda et al. (2012) mentioned using such software.

For these tools to serve as effective deterrents, students must be aware that instructors use them and act on their results. Demonstrating the capabilities of a similarity detection tool is a powerful prevention method.

We will focus more in-depth onto the various types of tools, their underlying fundamentals, and popular tools in use today in chapter 2.

1.5. Goal and structure of this dissertation

Source code plagiarism is a recognised problem (section 1.1.3) with significant prevalence (section 1.2.1). A robust strategy to mitigate plagiarism involves using source code similarity detection tools (section 1.4). However, instructors often find the process of plagiarism detection and prosecution arduous, leading to an under-utilisation of these tools (Coren 2011). Contemporary similarity detection tools face numerous challenges (Albluwi 2019; Novak et al. 2019). Albluwi (2019) and Weber-Wulff (2019) highlight several issues with existing tools. Team Dodona's 2019 search for viable similarity detection tools revealed that current tools suffer from one or more of the following shortcomings:

- Inadequate similarity detection.
- Limited programming language support with no straightforward way to add new languages.
- Poor user interface (UI) and user experience (UX) design.
- Minimal or no visualisations.
- Installation difficulties.
- Unavailable source code.
- Poor source code quality and deprecated dependencies.
- Incompatible with privacy regulations such as GDPR.
- Security concerns.

1.5.1. Research goal

To address this gap, we embarked on developing a source code similarity detection tool that resolves all these issues. Specifically, we aim to create an open-source similarity detection tool with the following features:

- Proven, high-quality algorithms for similarity detection.

- Language-agnostic with broad programming language support and easy extensibility.
- Excellent UI and UX
- Visualisations that assist educators in preventing and detecting plagiarism in various settings (formative and summative assessment).
- Easy to use and install.
- Open-source and flexible to accommodate diverse use cases.
- Good software design, well-maintained and up to date dependencies.
- Respecting teacher and student privacy and compliant with privacy regulations.
- Conforming to modern security standards.

Our efforts resulted in Dolos², a new source code similarity detection ecosystem.

1.5.2. Structure of this dissertation

This dissertation describes the algorithms, design, and implementation behind Dolos.

Chapter 2 provides context by exploring techniques, algorithms, and tools related to source code plagiarism detection.

Chapter 3 discusses the algorithmic foundations of Dolos, explaining its similarity detection pipeline step by step.

Chapter 4 presents the rationale behind Dolos's UI and UX, highlighting its main features and evolution.

Chapter 5 provides implementation details, including software components and user interactions.

Chapter 6 demonstrates Dolos's effectiveness using metrics, benchmarks, questionnaire results, and a case study on its use in teaching programming courses.

Chapter 7 highlights experiments conducted to enhance Dolos.

²dolos.ugent.be

Finally, chapter 8 concludes the dissertation by summarising results, assessing Dolos's impact, and suggesting avenues for further improvement.

Chapter 2.

Related work

This chapter extends the related work section of our journal article “Dolos: Language-agnostic plagiarism detection in source code” (Maertens, Van Petegem, Strijbol, Baeyens, Jacobs et al. 2022). My contributions to this section include conducting the literature study and writing the original draft. I have modified the original text to align with the style and structure of this dissertation. Furthermore, I have updated certain sections to incorporate the latest advancements and insights.

One of the earliest documented attempts to detect plagiarism in programming courses is attributed to Ottenstein (1976). Leveraging the observation by Bulut and Halstead (1973) that the likelihood of two programs sharing identical constellations of unique operators, operands and their total counts is exceedingly low, Ottenstein employed a program to count these four properties of FORTRAN programs. These 4-tuples served as fingerprints, with programs exhibiting identical 4-tuples flagged as potential plagiarism candidates warranting visual inspection. While this pioneering approach still encapsulates the essence of contemporary source code plagiarism detection, the field has involved significantly since then.

This chapter provides an overview of the relevant properties, algorithms and tools in plagiarism detection for programming assignments. We begin with a broad examination of plagiarism types and how tools can aid in detection (section 2.1). We then explore commonalities among plagiarism detection tools (section 2.2) and briefly discuss the algorithms they employ (section 2.3). Following this, we present a selection of popular tools that facilitate source code plagiarism detection (section 2.4). Later in this dissertation (chapter 6) we evaluate our similarity detection tool, Dolos, with some of the tools described herein.

2.1. Detecting plagiarism

Recognizing that students may plagiarise from different sources, and often attempt to disguise this, academics and industry professionals have devised numerous methodologies and identify suspicious submissions. Based on the type of plagiarism, these tools utilise diverse techniques, metrics, and algorithms.

This dissertation specifically addresses detecting similarities within a **closed document space**: all solutions submitted for a single assignment. This primarily targets instances where a student copies a solution from another student in the same class. Students may attempt to disguise their plagiarism through modifying the original source code by applying superficial obfuscations. Source code plagiarism detection tools tackling this issue will attempt to group similar programs for further manual inspection.

When code is plagiarised from an **external source**, the original source is absent from the collection of submissions under analysis. This includes copying existing solutions from the internet or from previous offerings. Detecting this form of plagiarism necessitates identifying similarities between a single solution against a vast, ever-expanding corpus, akin to the functioning of internet search engines (Brin et al. 1995).

Another manifestation of plagiarism is **contract cheating**, wherein a student engages someone else to complete the assignment, often for compensation. Detecting this form of plagiarism is challenging, as the solution itself is original, but the submitting student is not the true author. Authorship attribution (Bogomolov et al. 2021), a method to determine the author of a source code fragment, offers a potential approach to identify this form of plagiarism.

The recent surge in **Generative Artificial Intelligence (GenAI)**, particularly Large Language Models (LLMs), has democratised contract cheating by offering the ability to generate solutions for programming assignments free-of-charge. While rudimentary attempts exist to detect GenAI-generated code using metrics like perplexity and burstiness (Nguyen et al. 2024; Z. Xu and Sheng 2024), these approaches struggle to accurately discern authorship between humans and artificial intelligence (AI) (Pan et al. 2024). Other attempts focusing on educational contexts try to detect certain “tells” of AI programming assistants such as a high number of verbose comments and programming speed Strozanski (2024), but still remain in the prototype phase.

Some methods of plagiarism detection shift the focus from analysing final submission source code to examining student behaviour. This approach has been applied to individual student submission patterns (Hellas et al. 2017; Tahaei and Noelle 2018), or by identifying aberrant response patterns across all students (Alexandron et al. 2017) to detect suspicious interactions between multiple accounts.

2.2. Plagiarism detection fundamentals

The techniques and tools discussed in this chapter aim to support instructors in detecting and preventing plagiarism. However, determining plagiarism itself should not be reliant on automated tools, given the potentially severe implications. Therefore, algorithms designed to aid this process focus on identifying similarities in various forms. Although commonly referred to as *plagiarism detection tools*, we emphasise the use of the term *similarity detection tools* to reflect their true function.

In this dissertation, we focus on similarity detection in a closed monolingual collection of source files submitted as solutions for programming assignments. The collection contains both the source files under scrutiny for plagiarism and the source files they could have originated from (closed collection). All source files in the collection use the same programming language (monolingual collection). Typically, student submissions comprise tens or hundreds of lines of code, of varying code quality. We expect some source files to contain syntactical errors, but still want to detect plagiarism in those faulty submissions.

Prior to examining state-of-the-art software tools for detecting source code plagiarism in educational settings, we discuss the general workflow for plagiarism detection in practice and explore key concepts. The process commences with the aggregation, preparation and management of source file collections and their associated metadata such as submission timestamps and additional student information, which will serve as input for plagiarism detection tools. This preparatory process involves file manipulations like filtering, formatting, arranging and packaging files in the expected structure. For tools expecting a single file per submission, files from multi-file projects must be concatenated. For tools that are only able to analyse one programming language per analysis, files must be grouped per programming language. These preliminary data wrangling steps can be time-consuming if not adequately supported by online learning environments or custom scripts (Sheahen and Joyner 2016). Regrettably, leading source

code plagiarism similarity tools lack robust interoperability with external software platforms, offering only non-standard command-line interfaces (CLIs).

2.3. Similarity detection algorithms

Similarity detection tools primarily offer rapid algorithms to identify similarities among source files and assist reviewers in determining whether these similarities indicate plagiarism or are merely coincidental. Although extensive research has been conducted on the algorithmic aspects of screening source code for similar fragments (Roy et al. 2009), contemporary leading tools uniformly employ a two-step approach, albeit with varying implementation details. The first step transforms each source file into a list of tokens to mask local obfuscations. Tokens represent structural elements in the source code, such as keywords, variables, or operators. Tokenization utilises software components typically found in the front end of a compiler: a lexer for lexical analysis, a parser for syntax analysis, and a semantic analyser (Aho et al. 2006). The token stream captures the source code's structural elements and excludes whitespace. Literal values, identifier names or comments are denoted as anonymous syntactic constructs. The second step involves searches for similar code fragments by performing pairwise alignment on the token list of each submission pair to account for more global obfuscations, such as insertions, deletions, substitutions and transpositions (Wise 1993).

Contemporary similarity detection tools predominantly employ one of two algorithms to detect similarities between source code: Winnowing, or Running Karp-Rabin Greedy-String Tiling (RKR-GST). While some alternative techniques have shown promising initial results, they have not advanced beyond the proof-of-concept stage to yield practical tools that validate the authors' conclusions. Novak et al. (2019) have put it this way:

In spite of the large production of tools in recent years, most of the tools are not available to the public, they are used only by the authors that developed them and are mentioned in only one article.

Alternative techniques for source code similarity detection include tree-based algorithms (Li and Zhong 2010; Zhao et al. 2015), graph-based algorithms (Chae et al. 2013; Liu et al. 2006) Latent Semantic Analysis information retrieval (Cosma and Joy 2012), fuzzy-based match-

ing (Acampora and Cosma 2015), program logic analysis (Cheers et al. 2019), and program behavioural analysis (Cheers et al. 2021).

2.3.1. Greedy String Tiling

The RKR-GST algorithm, introduced by Wise (1993), is used by similarity detection tool such as JPlag (section 2.4.2) and Plaggie (section 2.4.3). The Greedy String Tiling algorithm iteratively searches for matches between two token streams T_1 , T_2 with a minimum match size s using the following high-level steps:

1. Start with all tokens unmarked.
2. Identify the longest match between unmarked tokens in T_1 and T_2 .
3. Mark all tokens in this longest match.
4. Repeat steps 2 and 3 until no further matches of at least length s can be found.

The RKR-GST algorithm extends Greedy String Tiling by leveraging rolling Karp-Rabin hashes to quickly identify maximal matches (Karp and Rabin 1987), while applying additional optimisations to expedite the average-case matching process.

Wise (1993) demonstrates that the worst-case time complexity of RKR-GST is $\mathcal{O}(m^3)$ for comparing two token sequences comprising m tokens in total. However, an informal estimate suggests that the practical complexity lies between $\mathcal{O}(m)$ and $\mathcal{O}(m^2)$. This complexity pertains to the comparison of two token sequences; consequently, computing all pairwise comparisons among n files results in a worst-case time complexity of $\mathcal{O}(n^2m^3)$.

2.3.2. Winnowing

Moss (section 2.4.1), Compare50 (section 2.4.6), and Dolos use the Winnowing algorithm introduced by Schleimer et al. (2003) to derive fingerprints from hashed syntax tokens of submissions. Matching fingerprints between program submissions suggest similarities that may indicate plagiarism. Given that the Winnowing algorithm is used by Dolos, which is presented in this dissertation, we provide a detailed exposition of the algorithm in chapter 3.

2.4. Source code similarity detection tools

The landscape of source code plagiarism detection tools is fragmented, having evolved organically from in-house scripts to published tools. There is no standardised format for reporting similarity analysis results, although all tools compute a similarity score from each pairwise alignment of source files and present a filtered or sorted list of these scores. Pairwise similarities are expressed either as values between 0 and 1 or as percentages, with higher scores indicating a higher likelihood of plagiarism. However, the different similarity measures used by different tools hinder direct comparison. Some tools offer functionalities to inspect high-similarity pairs or cluster source files into larger groups based on similarity. Unfortunately, support for advanced plagiarism exploration is generally lacking, and some tools do not even output analysis results in a machine-readable format for further processing.

Novak et al. (2019) identified 120 tools for source code similarity detection in the scientific literature, but most have never been publicly available or are no longer accessible. In section 6.2 we compare Dolos against the leading tools identified by Novak et al. (2019): Moss (Schleimer et al. 2003), JPlag (Prechelt et al. 2002), Plaggie (Ahtiainen et al. 2006), Sherlock Warwick (Joy and Luck 1999), Sherlock Sydney, and Compare50. Table 2.1 summarises the relevant properties of these tools.

Before we explore the inner workings, advantages and disadvantages of these tools, it is essential to clarify a potential source of confusion. In the realm of source code similarity detection, two distinct tools share the name “Sherlock”, yet only one is accompanied by a publication. Despite being developed in different programming languages and at different universities, there is evidence suggesting that some publications have inadvertently referenced one tool while utilising the other. To mitigate this confusion, we include both tools in our discussion and will consistently refer to them with their respective university suffixes throughout this manuscript: Sherlock Warwick and Sherlock Sydney.

In addition to free-to-use similarity detection tools found in literature, there are also commercial platforms offering source code plagiarism detection services. Examples include Codequiry¹, Copyleaks², and Gra-

¹codequiry.com

²copyleaks.com/code-plagiarism-checker

2.4. Source code similarity detection tools

Table 2.1. Properties of similarity detection tools benchmarked in this study.

	Dolos	Moss	JPlag	Plaggie	Sherlock Warwick	Sherlock Sydney	Compare50
Initial release	2020	1997	2001	2002	1999	2011	2017
Active Development	Yes	No	Yes	No	No	No	Yes
Pairwise Inspection	Yes	Yes	Yes	Yes	Unknown	No	Yes
Visualisations	Yes	No	Yes	No	Yes	No	Yes
Open-source	Yes	No	Yes	Yes	Yes	Yes	Yes
Local execution	Yes	No	Yes	Yes	Yes	Yes	Yes
Web server	Yes	Yes	No	No	No	No	No
Self-hostable web server	Yes	No	No	No	No	No	No
Programming language parser	Yes	Yes	Yes	Yes	Yes	No	Yes (lexer)
Supported programming languages	17 (400+ possible)	25	15	1 (Java)	1 (Java)	0	300+
Graphical UI	Yes	Yes	Yes	Yes	Yes	No	Yes
Data export	Yes (CSV)	No	Yes (CSV)	No	Unknown	Yes (TXT)	No

deScope³. However, in this dissertation we limit ourselves to published free-to-use similarity detection tools.

2.4.1. Moss

Moss (Measure Of Software Similarity) was developed at Stanford University (USA) and is offered as a web service⁴ freely accessible for non-commercial purposes. Although its source code remains proprietary, the foundational Winnowing algorithm was published by Schleimer et al. (2003). Moss supports the following programming languages: C, C++, Java, C#, Python, Visual Basic, JavaScript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, and HCL2.

To utilise Moss, users must register via an automated email service, which provides them a unique user id. Submissions of source file collections to the Moss server are facilitated through a command-line Perl script. The web service tokenises these files, extracts fingerprints, and calculates pairwise similarities. Moss presents results as HTML pages, listing pairs of highly similar source files and highlighting shared code snippets in a side-by-side comparison.

According to modern standards, Moss has a rather archaic user interface. Nevertheless, the Moss community has contributed submission scripts in other programming languages, alternative Graphical User Interfaces (GUIs), integrations with other web services, and tools to convert Moss HTML reports into machine-readable formats.

Using Moss requires sending source code of students to an external server located in the United States, potentially raising security concerns and conflicting with local privacy regulations. Moreover, the service occasionally experiences responsiveness issues due to high demand or unscheduled downtimes, particularly during peak exam periods in January and June.

Since November 2022, Moss has imposed a daily submission limit of 100 submissions per user to mitigate the impact of automated bots submitting tens of thousands of jobs daily. Regrettably, this restriction renders Moss impractical for similarity detection in larger courses.

³gradescope.com

⁴theory.stanford.edu/~aiken/moss/

2.4.2. JPLag

JPlag is a similarity detector crafted in Java, is still actively developed and maintained at the Karlsruhe Institute of Technology (KIT) in Germany. Its inception dates back to 1996, with an initial publication as a web service in 2001 by Prechelt et al. (2002), though the source code remained undisclosed at that time.

Some years later, JPlag was released as a Moodle plugin and as a command-line tool executing locally. In June 2007, the JPlag maintainers open-sourced their software under the GNU General Public License (GPL). The tool tokenises source files for supported programming languages and uses the RKR-GST algorithm (Wise 1993) to compute similarities based on the fraction of tokens covered by matching string tiles. JPLag reports results in CSV and HTML format.

Until 2021, JPlag was mostly in *maintenance mode*, with only little development going on. Recent efforts by the current maintainers have introduced new features and techniques. These innovations enhance resilience against sophisticated automated obfuscation attacks by applying token sequence normalisation (Sağlam, Brödel et al. 2024) and subsequence merging (Sağlam, Hahner et al. 2024). Sağlam (2025) has comprehensively documented these contributions in a doctoral dissertation.

As of now, JPlag supports detecting similarities in 15 programming languages (Java, C, C++, C#, Python, JavaScript, TypeScript, Go, Kotlin, R, Rust, Swift, Scale, LLVM IR, an Scheme), 3 modelling languages (EMF Metamodel, EMF Model, and SCXML), and plain text.

2.4.3. Plaggie

Ahtiainen et al. (2006) developed Plaggie as an open-source alternative to the then closed-source JPlag. Created in Java at the Helsinki University of Technology (HUT) in Finland, Plaggie mimics JPlag's functionality, employing the RKR-GST algorithm for detecting similarities. It supports Java exclusively and reports results in HTML format.

HUT has since then merged into Aalto University.

2.4.4. Sherlock Warwick

Sherlock Warwick, developed in Java at the University of Warwick (UK), is an open-source tool released under the GPLv2 licence. This

command-line utility supports most programming languages as plain text, and offers specific optimisations for Java (Joy and Luck 1999).

Sherlock Warwick employs an incremental comparison approach, analysing three versions of the source code: the original text, a version with stripped from comments and whitespace, and a tokenised version. The first two comparisons are applicable to all programming languages, while the tokenised comparison is exclusive to Java. Sherlock Warwick determines a pairwise similarity score based on the longest common matches of characters or tokens, with a configurable allowance of insertions and deletions. Unfortunately, the original article does not mention any details about the used algorithm (Joy and Luck 1999).

Despite our best efforts, we were unable to compile and execute Sherlock Warwick from its source code. Essential dependencies are either unavailable online or require a specific version that we could not identify. Although a compiled JAR file exists, executing it yielded no results in the GUI. Furthermore, extracting similarity values from generated reports proved impossible due to unreadable formats and insufficient documentation. These obstacles prevented us from including Sherlock Warwick in our validation benchmark.

2.4.5. Sherlock Sydney

Sherlock Sydney, developed in C at the University of Sydney (Australia), is a command-line tool that processes all source code as plain text, without specific support for individual programming languages.

The tool parses text files, including source code, into streams of words. It then hashes sequences of these words and discards non-zero hashes after applying a bitmask, employing a Winnowing algorithm distinct from that used by Moss (Schleimer et al. 2003). The remaining hashes serve as digital fingerprints, with pairwise similarity calculated as the ratio of shared fingerprints between text files. Sherlock Sydney reports results in a text file or directly in the terminal. While processing source code as plain text is efficient, it is less effective against obfuscation methods typically observed in educational source code plagiarism.

The source code of Sherlock Sydney was initially available on the university's website in 2011 but was removed in 2018. Other developers recovered a snapshot from the Internet Archive for further maintenance on GitHub.

2.4.6. Compare50

Compare50⁵ is an open-source similarity detection tool developed by Harvard University's (USA) as part of their open university course CS50⁶. It is a Python CLI program that uses five different comparison methods. Four comparison methods use the Winnowing algorithm (section 2.3.2), each on a different version of the code, ranging from the program structure to the exact text. A fifth comparison method looks for identical misspellings in the comments of the code, which are highly indicative of plagiarism. Compare50 use the lexer of the Pygments⁷ syntax highlighting library to support over 300 programming languages and templating languages.

Compare50 generates Hypertext Markup Language (HTML) files with the analysis results. It will output the top N matches between a pair of submissions, with $N = 50$ by default, and list those in an overview page. Compare50 rates each match with a score ranging from 1 to 10, combining and normalising the output of the enabled comparison methods.

⁵github.com/cs50/compare50

⁶cs50.harvard.edu

⁷pygments.org

Chapter 3.

Algorithmic underpinnings

The end goal of Dolos is straightforward: to identify similarities between source files that may indicate potential plagiarism. Dolos achieves this through a complex process comprising multiple steps working in tandem. Data scientists use the analogy of a *pipeline* for analysis processes that involve multiple stages. By dividing the analysis into discrete steps with clearly defined inputs, outputs, and responsibilities, we maintain a high-level overview that aids in understanding and reasoning about the pipeline.

We will describe the algorithms underpinning Dolos as the Dolos Source Code Similarity Detection Pipeline, or *the pipeline* for short. This chapter details the pipeline, focussing on the algorithms in each step. The implementation details are left out, as those will be discussed in chapter 5.

Figure 3.1 visualises the steps comprising the Dolos pipeline. We can group these steps into three stages:

- **Tokenisation:** This stage transforms the source files into sequences of syntax tokens (section 3.1).
- **Fingerprinting:** This stage extracts *fingerprints* from the syntax tokens, where identical fingerprints may indicate plagiarism (section 3.2).
- **Reporting:** This stage collects and aggregates the fingerprints, to build a similarity report for manual inspection (section 3.3).

Let's walk through the pipeline with a high-level overview: We feed a collection of source files into the pipeline, initiating the first stage: **tokenisation**. The first step is *parsing* each source file into its concrete syntax tree, described in section 3.1.1. This erases identifiers such as variable names, and is robust against different usage of whitespace. Next, the *serialisation* step flattens the syntax tree while tracking each

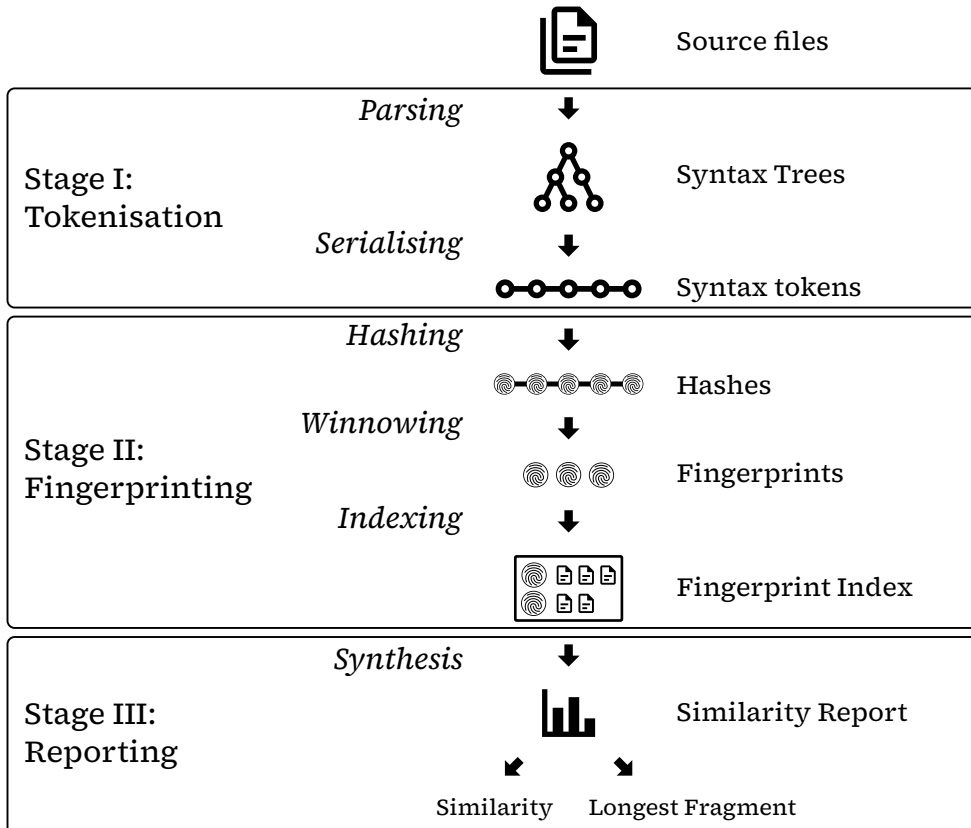


Figure 3.1. Diagram of the Dolos Source Code Similarity Detection Pipeline visualising how the pipeline processes a collection of source files into a similarity report. Concrete transformations are listed left of the arrows, the intermediate artefacts are located on the right of a symbolic representation.

token's location in the original source file, detailed in section 3.1.2. This step includes removing comment tokens, described in section 3.1.3.

The subsequent **fingerprinting** stage begins by *hashing* the list of tokens (section 3.2.1), aggregating them into *k*-grams, which are then hashed (section 3.2.2). From the resulting *k*-grams hashes, we select a representative subset of fingerprints using the Winkling algorithm (section 3.2.3). A fingerprint index aggregates the fingerprints of all submissions included in the analysis, maintaining the information needed to map fingerprints back to their original file locations (section 3.2.4).

In the final **reporting** stage, Dolos builds a similarity report by synthesising information from the fingerprint index. This involves generating all pairs of submissions (section 3.3.1), calculating their similarity (section 3.3.2), and determining their longest common substring (section 3.3.3). The output provides all information needed to report the analysis results to the instructor, who can then search for indications for plagiarism.

In the following sections, we will give a detailed explanation of the above steps comprising the pipeline.

3.1. Tokenisation

The first stage of the pipeline, illustrated in figure 3.3, converts each source file into a sequence of syntax tokens, encapsulating the syntactical structure of the source file. These tokens mask identifier names, comment contents, and constant values within anonymised tokens. In addition, these tokens ignore whitespace, and we remove comment tokens as part of the serialisation step (section 3.1.3). These anonymised tokens are robust against all obfuscations in the category of lexical changes (section 1.3.2). Source files modified only by these obfuscations result in an identical sequence of syntax tokens and will have a 100% similarity.

3.1.1. Parsing to a syntax tree

Converting a source code file into syntax tokens begins with parsing the source code into a syntax tree or parse tree. Compilers perform this step as part of their front-end to process source code into its intermediate representation (IR). Dolos uses parsers implemented using

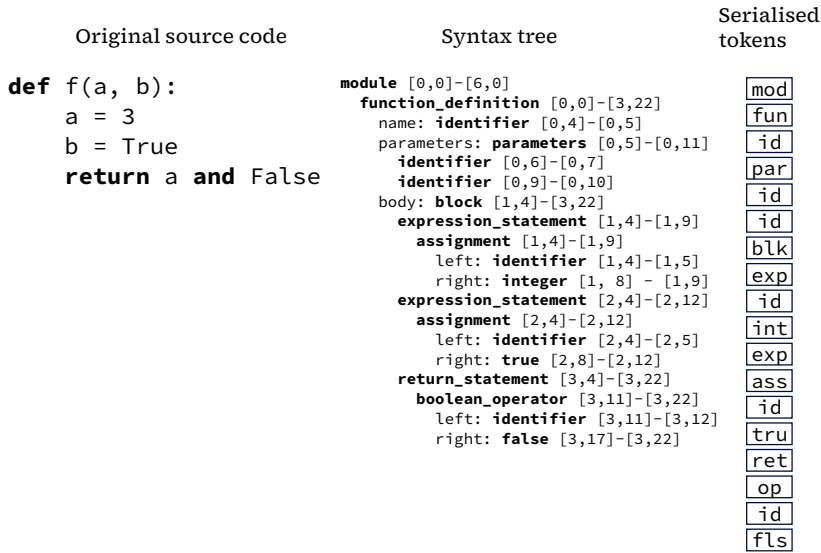


Figure 3.2. Example of the syntax tree and serialised tokens corresponding to a small source code snippet. The actual syntax tokens themselves are highlighted in bold in the syntax tree. A list of tokens is the end result of the tokenisation phase of the similarity detection pipeline.

the Tree-sitter¹ parser generator framework by Brunfeld et al. (2024). This framework provides a unified Application Programming Interface (API) for parsing source files into their concrete syntax tree (CST) for a broad range of programming languages.

The parse tree generated by tree sitter makes a distinction between unnamed nodes (semicolons, parentheses, ...), and named nodes (identifiers, actual keywords, ...). When considering all nodes, this parse tree is considered a CST. Dolos only takes the named nodes into account, which is closer to an abstract syntax tree (AST). However, because an AST includes more information as the result of subsequent processing, we will henceforth use the term syntax tree when referring to the tree generated by the parsing step. We provide more details on how we use Tree-sitter in section 5.2.

Parser types

Most modern programming languages are classified as context-free languages (Chomsky and Schützenberger 1959), parseable by a non-

¹tree-sitter.github.io


```

1 let x = (y);
2 let x = (y) => true;

```

Listing 3.1. Example JavaScript code where the syntax token assigned to `y` is ambiguous when an LR parser processes the closing bracket. The first line assigns to `x` the value of `y`, meaning that `y` would have to be an `identifier`. The second line assigns to `x` an arrow function expression where `y` is a `function_argument` syntax token. LR(1) parsers are unable to handle this difference, because they decide on the token as soon as it processes the closing bracket. GLR parsers handle this ambiguity by keeping track of all possible parsing states, discarding states as the code progresses.

deterministic pushdown automaton. Conventionally, canonical LR (LR(1)) parsers (left-to-right, rightmost derivation in reverse) are considered algorithms to parse those programming languages in linear time. However, LR(1) parsers are only applicable to *deterministic* context-free languages, whereas real-life programming languages are often *ambiguous* and *non-deterministic*, as demonstrated in listing 3.1.

Generalised LR (GLR) parsers, introduced by Tomita (1985), offer a solution to this problem by introducing a *graph-structured parse stack* that forks the internal parser state upon encountering ambiguity, collapsing as soon as the ambiguity resolves. This approach also facilitates error handling: programs with incorrect syntax result in multiple possible parser stacks at the end of the program. From these stacks, we can select the stack closest to a working program and place an `ERROR` syntax token in the parse tree corresponding to the unparseable region (Visser 1997). The Tree-sitter parser generator used by Dolos, generates GLR parsers supporting incremental parsing.

Other, widespread parser implementations for code highlighting use regular expression-based parsers. These parsers are simpler to implement and write grammars for, but they are limited because regular expressions can only fully process regular grammars. Thus, these parsers cannot produce the same quality of syntax tree or handle more complex programs.

3.1.2. Serialisation and location mapping

Searching for similarities between syntax trees can be reduced to the NP-complete subgraph isomorphism problem (S. A. Cook 1971). To mitigate this computational expense, we serialise each syntax tree into

```

1  def generate_tokens(code: str) -> List[Token]:
2      tree = tree_sitter.parse(code)
3      tokens = []
4      tokenize_node(tree.root, tokens)
5      return tokens
6
7  def tokenize_node(node: Node, tokens: List[Token]) -> Location:
8      current_location = Location(node.location)
9      if not in node.type.includes("comment"):
10         tokens.push(Token("(", current_location))
11         tokens.push(Token(node.type, current_location))
12         for child in node.children:
13             child_location = tokenize_node(child, tokens)
14             if child_location.start < current_location.end:
15                 current_location.end = child_location.start
16         output.push(Token(")", current_location))
17     return current_location

```

Listing 3.2. Psuedocode of the algorithm to serialise the syntax tree into a list of tokens while keeping track of their corresponding location in the source file. The algorithm walks through the tree in pre-order, adding a token to the output list before descending to its children and inserting pseudo-tokens (and) to indicate descent and ascent in the tree. This snippet already includes code to filter out comment nodes.

a sequence of syntax tokens, which is more efficient to compare. We use the algorithm in listing 3.2 to traverse a syntax tree in pre-order.

While performing this transformation, we also track each syntax node's corresponding location in the original source file. This allows us to highlight matching code blocks when presenting results (section 4.6.1). The location Tree-sitter sets for nodes in the syntax tree includes the whole region covered by that syntax node. For example, a `function_declaration` token will include the function body as its corresponding location in the file. This behaviour does not align for our use-case, as a matching function declaration would mark the full function as a match, even if the function body differs. We address this by subtracting a parent node's end location with the start location of its children.

The output of this step is a list of tokens for each source file, consisting of a string with the syntax token type and its corresponding location.

3.1.3. Removing comment tokens

Programmers often include comments in source code to document and clarify parts of their programs. Instructors encourage students to do the same, as comments help in following the students' thought

process during code evaluation. However, the syntax tree generated by Tree-sitter includes comments, so adding or removing comments alters the syntax tree of a submission. Consequently, this obfuscation is the only one in the category of lexical changes (section 1.3.2) that affects the syntax tree. It is trivial for students to add comments to their code, and doing so is an effective way to evade detection by syntax tree based similarity detection.

As a countermeasure, the pipeline removes comment tokens. Assuming syntax node types include the literal string “comment”, Dolos excludes tokens from the serialised list matching this string. This filter removes both block comments and line comments.

Without this fix, Dolos’s similarity detection pipeline would report low similarities for submissions differing only in their comments. For example, in the π -ramidal constants evaluation dataset ², one student plagiarised another and added a few comments to evade detection. With only four added comments and no other structural changes, this reduced the similarity to 87%. Adding a comment on every line would have lowered that pair’s similarity to 48%, below that of other non-plagiarised submission pairs. With the current pipeline filtering out comments, these submissions are 100% similar.

3.2. Fingerprinting

Now that we have a sequence of syntax tokens for each submission, we proceed to identify similarities between submissions. Finding common subsequences among these submissions is a complex and computationally expensive task, especially as the number and length of sequences increase. Additionally, we need to be able to configure the pipeline’s sensitivity. For small exercises, we want to detect even a few plagiarised lines. For larger projects, we expect more coincidental small matches and focus only larger chunks of plagiarised code.

To address these requirements, we apply techniques described by Schleimer et al. (2003), who describe a Winnowing algorithm to extract *local* document fingerprints for detecting copies. These metaphorical fingerprints serve a similar purpose as real fingerprints in crime investigations: they provide clues about who was involved. The extracted fingerprints aim to reduce a submission’s data as much as possible to

Fingerprints is of course used as a metaphor, as source code does not have fingers.

²dolos.ugent.be/demo/pyramidal-constants/evaluation/

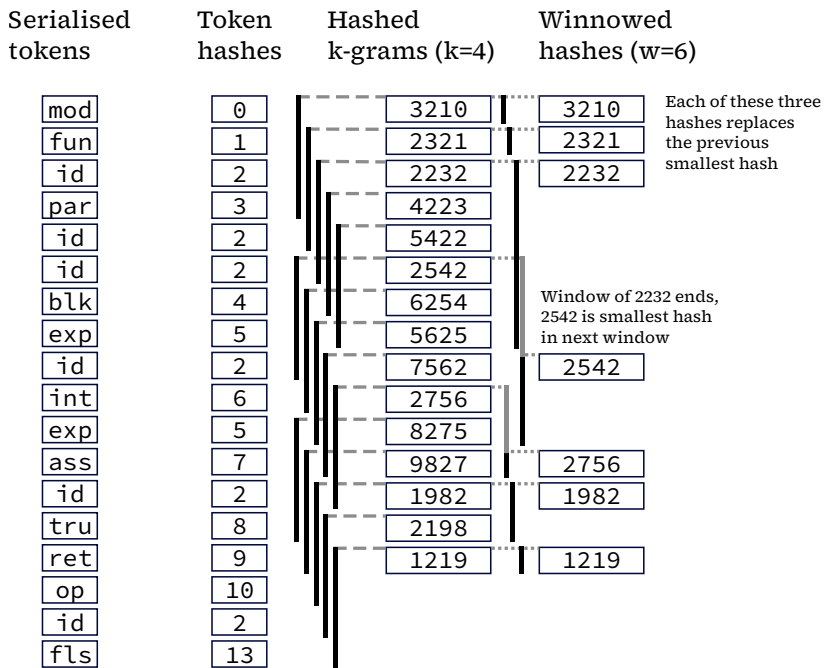


Figure 3.3. Example illustrating how the Winnowing algorithm extracts fingerprints from a serialised list of tokens. This example illustrates the two situations when the Winnowing algorithm selects a new fingerprint: when a new hash has a smaller value than the previously smallest hash, and when the smallest hash exits window and a new smallest hash is selected.

ensure efficient similarity detection while preserving essential information that could indicate plagiarism. Figure 3.3 illustrates this process with a concrete example.

We compute a hash for each group of k subsequent tokens, known as k -grams. We then sample a representative set of k -grams using the Winoing algorithm. This representative set constitutes the fingerprints of a submission. The final step of this stage involves building a fingerprint index that facilitates fast queries to compute similarity metrics when generating the final report.

3.2.1. Hashing tokens

The tokens in the serialised subsequences are strings with a clear description of the syntax token itself, such as `identifier`, `statement_block`, and `return_statement`. To speed up the fingerprinting step, we first reduce the data by hashing these strings using a polynomial hash function. We employ a hashing function similar to the Rabin-Karp rolling hash scheme (Karp and Rabin 1987), but with an infinite window instead of a rolling hash. This calculates the token hash H_T for a token with length l as:

This hash is often called Rabin fingerprinting, but those fingerprints interpret their input as bits instead of characters.

$$\begin{aligned} H_T &= a_T^l \cdot c_0 + a_T^{l-1} \cdot c_1 + \dots + a_T \cdot c_{l-1} \mod m \\ &= \sum_{i=0}^{l-1} a_T^{l-i} \cdot c_i \mod m \end{aligned}$$

Where a_T is the multiplier, c_i is the i -th character of the token string starting at 0, and m is the modulus of the hash function. We can efficiently calculate this hash incrementally by observing that, given the hash up until the i -th character of the token string, we can calculate the hash including the $i + 1$ -th character as:

$$H_{T,i+1} = (H_{T,i} + c_{i+1}) \cdot a_T \mod m$$

Where $H_{T,0} = c_0 \cdot a_T \mod m$. Note that this incremental version avoids expensive exponentiation, requiring only an addition, a multiplication, and a modulo operation.

We can freely choose values for a_T and m , but for an optimal hashing function that distributes the hash values uniformly while avoiding collisions, we must apply some requirements. First, the modulus m determines the maximum value of the hash outputs and should be chosen while keeping the maximum integer range in mind. Second, a good multiplier a_T should spread the input values uniformly across

```

1 def token_hash(token: str) -> int:
2     h = 0
3     for char in token:
4         h = (h + char) * base % mod
5     return h

```

Listing 3.3. Pseudocode of the algorithm transform a syntax token from its string representation to a hash using the Rabin fingerprinting scheme. The value of *base* is the radix of hashing function and *mod* is the modulus (a hash can never exceed this value).

the integer range dictated by m . A suitable choice of a_T is the largest prime such that $a_T \cdot c_{\max} < m$ where c_{\max} is the maximum value of a character in our token string. We explain the constants chosen in Dolos’s implementation in section 5.3.1.

As an alternative to token hashes, we can use unique integer identifiers for each syntax token. For example, each parser additionally includes a token type identifier with each syntax node. This approach would be slightly faster, as it does not require iterating over each character in the token type string. Tree-sitter assigns these tokens incrementally, so the integer identifiers will not use all bits evenly. If we were to use these identifiers, the multiplier a_R used in the next step should be chosen similar to a_T to spread out these bits more optimally.

3.2.2. Hashing k -grams

In the next step, we hash k -grams of k subsequent hashed tokens using a polynomial rolling hash function. This hashing scheme is employed within the Rabin-Karp string matching algorithm (Karp and Rabin 1987). It is similar to the hashing scheme Dolos uses to hash tokens (section 3.2.1), but each hash only applies to the window of k previous tokens. Specifically, for a series of token hashes t_0, t_1, \dots, t_n , the rolling hash of k subsequent tokens ending with t_i is defined as:

$$\begin{aligned}
 H_{R,i} &= a_R^{k-1}t_{i-k} + a_R^{k-2}t_{i-k+1} + \dots + t_i \mod m \\
 &= \sum_{j=0}^{k-1} a_R^j \cdot t_{i-j} \mod m
 \end{aligned}$$

We can compute this rolling hash incrementally using only the token hash entering and the token hash leaving the rolling hash window:

$$H_{R,i+1} = H_{R,i} + t_{i+1} - a_R^k \cdot t_{i-k} \mod m$$

This incremental approach includes a potentially expensive exponentiation in the last term. However, we need only calculate $a_R^k \mod m$ once upon initialisation of the rolling hash function and reuse this value when needed.

The value of k is a parameter of this hash function that determines the minimum detectable length of identical syntax token sequences between two submissions. This value is modifiable according to the user's needs. A small k results in a more sensitive pipeline that will recognise smaller fragments of duplicated code, while potentially generating more false positives due to coincidental similarities. Larger values of k are less prone to false positives, but may miss smaller plagiarised code fragments.

The constants a_R and m should be chosen for the implementation at hand. The value of m determines the largest value of a hash and can be the same for both the token hash and rolling hash functions. However, multiplier a_R should be distinct from a_T . Otherwise, the result of hashing a token "abc" and hashing three subsequent tokens "a", "b", and "c" in the rolling hash would be identical. Although this situation is unlikely to occur in practice, we can avoid it by choosing a different values for a_T and a_R . Additionally, when using values from the token hash as inputs for our rolling hash, we can assume that our input values are already evenly distributed over the integer range $[0, m)$. This allows us to pick a different value for a_R that further scrambles previous hash values.

3.2.3. Winnowing

Now that we have integer hashes representing each group of k subsequent tokens, we can use the Winnowing algorithm presented by Schleimer et al. (2003) to select a smaller, representative set of fingerprints for detecting similarities. The Winnowing algorithm has two parameters: k for the k -gram size and w for the window size. The parameter k directly defines the *noise threshold*, limiting the smallest detectable match. The window size w indirectly determines a *guarantee threshold* $t = k - 1 + w$, guaranteeing matching sequences longer than this threshold. Listing 3.5 shows the pseudocode of the Winnowing algorithm used in the Dolos source code similarity detection pipeline.

```

1 class RollingHash:
2     def init(k: int)
3         self.k = k
4         self.hash = 0
5         self.i = 0
6         self.max_base =  $-a_R^k \bmod m$ 
7         self.memory = [0 for _ in 0..k]
8
9     def next_hash(token_hash: int) -> int:
10        self.hash = (self.base * self.hash + token_hash +
11        ↪ self.max_base * self.memory[this.i]) % this.mod
12        self.memory[this.i] = token_hash
13        self.i = (self.i + 1) % self.k
14        return self.hash

```

Listing 3.4. Pseudocode of the algorithm for transforming a syntax token from its string representation to a hash using the Rabin fingerprinting scheme. The value *base* is the multiplier of the hashing function and *mod* is the modulus (a hash can never exceed this value).

```

1 def winnow(int w, int k, tokens: List[str]) -> List[int]:
2     rolling = RollingHash(k)
3     r = 0 # Window right end
4     imin = 0 # Index of minimum hash
5     output = [] # Winnowed output hashes
6     # Circular buffer implementing a window of size w
7     window = [+∞ for _ in 0..w]
8     # At the end of each iteration, min holds the position
9     # of the rightmost minimal hash in the current window
10    for token in tokens:
11        r = (r + 1) % w; # Shift window
12        window[r] = rolling.next_hash(token_hash(token))
13        if imin == r:
14            # The previous minimum left the window
15            # Scan the window for the rightmost minimal hash
16            i = r + 1 % w
17            while i != r:
18                if window[i] <= window[imin]:
19                    imin = i
20                i = i + 1 % w
21            output.push(window[imin])
22        else:
23            # The previous minimum is still in the window
24            if window[r] <= window[imin]:
25                imin = r
26            output.push(window[imin])
27    return output

```

Listing 3.5. Pseudocode of the Winkowing fingerprinting algorithm applied on a sequence of syntax tokens. This pseudocode uses the token hash algorithm from listing 3.3 and the rolling hash algorithm from listing 3.4. The output of this Winkowing algorithm is a selection of local fingerprints.

Fingerprint properties

The Winnowing algorithm guarantees that it extracts at least one common fingerprint from two submissions sharing a matching series of tokens that is at least as long as the guarantee threshold t . This matching series does not necessarily need to be identical, as the algorithm extracts only one representative fingerprint for that series, allowing for small variations between two series of tokens. Shorter matching series also have a chance of being detected, as long as they are at least as long as the noise threshold k .

To achieve the *guarantee threshold* t , the Winnowing algorithm selects the minimum hash from every window. By selecting the minimum hash, this fingerprinting selection algorithm remains robust against modifications of the syntax tree, which can alter the order of hashed tokens. This occurs with many obfuscations that apply structural changes to the submission (section 1.3.2).

However, with enough structural obfuscations, it is still possible to inhibit proper fingerprint selection, thereby avoiding detection when using the Winnowing algorithm. The Winnowing algorithm is known to be vulnerable to injecting many redundant code fragments, which can substantially lower the similarity reported by many similarity detection tools (Devore-McDonald and Berger 2020). However, these attacks require advanced knowledge or the use of specialised tools, resulting in abnormal and suspicious-looking submissions. We emphasise that the primary purpose of Dolos is to detect obfuscations made with fragile knowledge. Advanced programmers can often rewrite programs in ways that even expert eyes would not discern as plagiarism, making this kind of attack unlikely in practice.

Winnowing output

In addition to selecting fingerprints from the stream of hashed tokens, an actual implementation must also track fingerprint locations in the submitted source files. These locations allow us to map matching fingerprints back to the original source file, showing instructors which parts of the source files share similarities. The visualisation comparing pairwise matches, discussed in section 4.6.1, utilises this fingerprint-location mapping.

The Winnowing algorithm outputs the selected fingerprints in the same order as the hashed syntax tokens. Thus, we can consider this algorithm as a *filter* that retains only the fingerprints while preserving their order in the *list* of output hashes. This property is important

```
1 def build_index(submissions: List[str]) -> Dict[int,  
  ↳ Fingerprint]:  
2     index = {}  
3     for submission in submissions:  
4         tokens = generate_tokens(submission)  
5         for hash, location in winnow(k, w, tokens):  
6             if hash not in index:  
7                 index[hash] = Fingerprint(hash)  
8                 index[hash].update(submission, location)  
9                 submission.fingerprints.add(index[hash])  
10    return index
```

Listing 3.6. Algorithm used to build the index applied on a sequence of syntax tokens. This pseudocode uses the token hash algorithm from listing 3.3 and the rolling hash algorithm from listing 3.4.

when calculating the longest duplicated fragment, as discussed in section 3.3.3.

3.2.4. Building the fingerprint index

As the final step in the fingerprinting phase, a fingerprint index collects winnowed fingerprints from all submissions, as illustrated by listing 3.6. The fingerprint index supports quick insertion of new fingerprints during its construction and efficient queries when building the similarity report. This index stores a record for each fingerprint containing information about which submissions it appears in, and its locations within those submissions. Due to internal code duplication, a fingerprint may occur multiple times in a single submission. Additionally, the index tracks which fingerprints belong to each submission.

Ignoring template code

Assignments for programming exercises often include a template with predefined code, where students must write new code in designated places. This template code is not intended for modification, so all students will share these common code fragments, resulting in many shared fingerprints that are not indicative of plagiarism. These fingerprints can overshadow those that actually indicate plagiarism. To mitigate this effect, the similarity detection pipeline includes an option to hide potentially harmless fingerprints.

As a first method to ignore certain fingerprints, Dolos allows the submission of template code as *ignored files*. The pipeline parses and winnows this code in the same way as normal submissions but marks their fingerprints as *ignored*. The similarity metrics exclude these fingerprints, and the match comparison editor (section 4.6.1) indicates that these code fragments were ignored.

Ignoring common fingerprints

The similarity detection pipeline also supports ignoring fingerprints that occur more than a user-defined threshold. There are situations where submissions can share code fragments that do not indicate plagiarism:

- The programming assignment requires using a specific API with initialisation code that will be very similar across student submissions.
- The instructor provided an example that solves part of the assignment.
- Students need to implement a given interface from scratch, which will naturally result in the same class structure.

Since instructors cannot always predict which code will be innocent, they can instruct the pipeline to ignore fingerprints if they occur in more than m submissions. Instructors can set this parameter m as an absolute value (e.g. 10 submissions or more) or as a fraction (e.g. in 60% of the submissions under analysis). As soon as a fingerprint occurs in m submissions or more, the pipeline marks it as *ignored*.

This creates an interesting prisoner's dilemma (Luce and Raiffa 1957) for students: if enough students share the same code fragment, their plagiarism might go undetected. We recommend setting this threshold conservatively when using this option. Dolos does not have a threshold set by default, but instructors can set one using the `-m` or `-M` command-line options (see chapter A).

3.3. Reporting

Up to this point, the Dolos source code similarity detection pipeline has extracted fingerprints for each submission, which may indicate code fragments with identical underlying structures. Note that a single fingerprint still corresponds to a single k -gram of k subsequent tokens

from the syntax tree. The pipeline extracts fingerprints such that if two submissions share a matching sequence of at least $t = w + k - 1$ tokens, they will share at least one fingerprint. There is also a very small chance of a hash collision, causing submissions to share a fingerprint that does not correspond to the same syntax tokens. Now everything is ready to query this index to build a similarity detection report.

The main result in a similarity detection report is the list of all submission pairs and three calculated metrics:

- **Total overlap:** Represents the total number of fingerprints shared between a pair, providing an absolute measure of the amount of shared code.
- **Similarity:** Indicates the fraction of shared fingerprints relative to the total number of fingerprints in the two submissions, revealing how similar the two files are.
- **Longest fragment:** Denotes the maximum number of consecutive fingerprints shared between the submission pair, calculated as the longest common substring.

3.3.1. Comparing pairs

The similarity detection pipeline compares all pairs of submissions under analysis. With n submissions, the total number of pairs is $\frac{n(n-1)}{2}$, making this step an inherently $\Theta(n^2)$ process. Comparing all pairs and computing their metrics, specifically calculating the longest common substring (LCS) (section 3.3.3), consumes most of the pipeline's execution time.

Although this part of the pipeline has undergone significant optimisations, such as postponing expensive computations (section 3.3.4), the quadratic nature of this step is still dominating the execution time. However, instructors are rarely interested in all pairs. For example, 100 submissions result in 4 950 pairs, and 200 submissions result in 19 900 pairs. When inspecting a similarity detection report, only the most similar or suspicious pairs are of interest.

It is still an open research question whether we can efficiently find the topmost similar pairs without comparing all submissions, as this would speed up Dolos's analysis speed dramatically. As of writing, we are building a prototype to construct an index using a generalised suffix tree. This alternative index could facilitate gathering the most

similar pairs without calculating all pairs. We discuss this approach in more detail in section 7.2.3.

3.3.2. Computing similarity

When comparing a pair of submissions, the similarity detection pipeline gathers all fingerprints associated with each submission. For one submission x in a submission pair, we define the following values:

- T_x is the total number of fingerprints extracted from that submission (including ignored fingerprints).
- I_x be the number of fingerprints marked as ignored.
- S_x is the number of fingerprints from x also occurring in the other submission of this pair, excluding ignored fingerprints.

With these values, we can define the **total overlap** and **similarity** metrics.

Similarity

The similarity between a submission pair is inspired by, but distinct from, the Jaccard similarity index (Jaccard 1901). We calculate the similarity $S(a, b)$ between two submissions a and b as:

$$S(a, b) = \begin{cases} \frac{S_a + S_b}{T_a + T_b - I_a - I_b} & \text{if } T_a + T_b - I_a - I_b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that S_a and S_b are not necessarily the same, as a fingerprint might have a different number of occurrences in submissions a and b due to internal duplication of that fingerprint.

Total Overlap

The similarity metric is highly dependent on submission size. It can easily inflate for small files, or students attempting to hide plagiarism can artificially lower this metric by adding a lot of superfluous content. In such situations, it might be interesting to only look at the numerator of the similarity metric. We call this metric the total overlap $O(a, b)$, calculated as:

$$O(a, b) = S_a + S_b$$

3.3.3. Computing the Longest Common Substring

Another interesting metric when analysing submissions for plagiarism, is the length of the longest duplicated code fragment. When many submissions under analysis share small code fragments that are not the result of plagiarism, this can inflate their similarity and total overlap metrics, potentially hiding longer shared code fragments that are actually plagiarised.

Dolos captures these longer fragments using the **longest fragment** metric. This metric is the length of the LCS of fingerprints between the two submissions in a submission pair, calculated by the pseudocode in listing 3.7.

When both submissions in a pair have n extracted fingerprints in total, counting the shared fingerprints and calculating the similarity and total overlap metric require $\Theta(n)$ time. In contrast, calculating the LCS takes $\Theta(n^2)$ time with the dynamic programming algorithm used in our pipeline. This step is the most computationally expensive in the whole similarity detection pipeline.

Algorithms exist to compute the LCS in $\Theta(n)$ time using suffix trees (Gusfield 1997). However, these algorithms are complex and might not necessarily result in an absolute speedup when used for this part alone. Experiments are currently ongoing to test an alternative index using generalised suffix trees. We describe our progress and the preliminary results in section 7.2.3.

3.3.4. Delayed calculation of fragments

Using the matching fingerprints between a pair of submissions, we can map these fingerprints back to their corresponding locations in the original source files. When doing this, we want to combine multiple subsequent identical fingerprints into matching code **fragments**.

Previous versions of Dolos would always reconstruct all matching code fragments for all submission pairs. The length of the longest fragment was used for its namesake metric, and the fragments themselves were serialised for use in the web user interface (UI). However, reconstructing full fragments is computationally expensive, and the serialised data requires significant disk space. When we discovered that the longest fragment length was equivalent to calculating the LCS, we switched to that implementation.

```

1 def lcs(left: List[Hash], right: List[Hash]) -> int:
2     prev = [0 for _ in left.length]
3     curr = [0 for _ in left.length]
4     longest = 0
5     for r in right:
6         i = 0
7         for l in left:
8             if l == r:
9                 if i == 0:
10                    curr[i] = 1
11                else:
12                    curr[i] = prev[i - 1] + 1
13                longest = max(curr[i], longest)
14            i++
15        prev = curr
16        curr = [0 for _ in left.length]
17    return longest

```

Listing 3.7. Dynamic programming algorithm to calculate the longest common substring using dynamic programming. After each iteration of the outer for-loop, `prev` contains in each index of `prev` the longest substring up until the currently processed part of the `right` hashes. The run-time of this algorithm is $\Theta(n^2)$ for strings of length n .

The Dolos web UI now calculates the fragments *on-demand* when viewing the comparison page (section 4.6.1), as it is more useful to highlight full fragments instead of individual matching fingerprint pairs. Since the web UI only needs to construct fragments for a single pair, the current unoptimised algorithm is still fast enough in most situations.

The algorithm used to construct fragments from fingerprints comprises multiple steps:

1. Create all fingerprint pairs between fingerprints shared between the current submission pair
2. Put each fingerprint pair in its own fragment
3. Iteratively merge subsequent fragments
4. Remove smaller fragments completely enveloped by bigger fragments

The resulting fragments include the fingerprints they envelop, and their start and end location in each of the source files. Note that as fingerprints may occur multiple times in the same source file, a single line of code may appear in multiple fragments and may have matching counterparts in multiple locations in the other source file. The number of fragments increases quadratically with the number of identical

fingerprints present in a submission pair. There are known degenerate cases where fragments of repetitive tokens, such as array literals with many elements, result in an explosion of fragments.

Chapter 4.

User Interface and User Experience design

In computer science, problems are typically well-defined, and the software and algorithms addressing these problems are logical and objective. We have established methods for verifying the correctness, providing a clear indication the problem is *solved*.

In the realm of user interface design, or design in general, identifying and defining the exact problem to solve is often much more challenging than solving it. This observation also applies to the field of source code plagiarism detection tools. Academics have invested significant effort and time in developing sophisticated algorithms to detect and circumvent attempts to conceal plagiarism. Still, instructors are often reluctant to investigate and prosecute plagiarism. This leads Albluwi (2019) to the following conclusion about source code similarity detection tools:

Building better tools, with user experience in mind, that offer features across the whole workflow is important. Currently, such tools are not available.

With Dolos, we make an attempt to fill this gap.

The following sections present the design of the Dolos user interface (UI), with a focus on the Dolos web UI. Although Dolos also features a command-line interface (CLI), users will spend most of their time interacting with the Dolos web UI. Those wishing to conduct a similarity analysis can also do so exclusively through the web UI, thanks to the Dolos web server (section 5.6). Henceforth, unless specified otherwise, the term UI refers to the Dolos web UI.

Section 4.1 outlines the design methodology and the philosophy of the UI. Subsequent sections explore the major components of the UI, descending the result hierarchy from top to bottom: beginning

with a bird's-eye overview in section 4.3, zooming in on our flagship plagiarism graph visualisation in section 4.4, and then delving into more granular details with clusters in section 4.5, pairs in section 4.6, and submissions in section 4.7. We end this chapter by showing how the Dolos UI evolved over time in section 4.8.

While reading this chapter, we invite you to experience the Dolos UI firsthand by visiting one of our demos on dolos.ugent.be/demo. The screenshots in the UI description primarily feature the π -ramidal constants exercise, either in evaluation setting or as a mandatory exercise. We encourage you to inspect these reports closely: whom would you suspect of committing plagiarism? Who was the source? What did the plagiariser do to conceal their actions?

4.1. Design methodology

Dolos first started as a prototype in 2019 and has since undergone iterative improvements. By adhering to the philosophy of “release early, release often”, we maintain a short feedback loop between developing new features and testing them. Since the initial release in 2021, we have published 33 releases in total, averaging a new release every one-and-a-half months.

Within this development cycle, we first built a prototype by implementing well-studied and battle-tested algorithms for similarity detection (chapter 3). Once this prototype was complete, we created benchmarks to compare its performance with state-of-the-art tools (section 6.2). These benchmarks indicated that Dolos performed on par with JPlag and Moss, the two most widely used similarity detection tools.

Following this, we focussed on creating a smooth user experience (UX). The Dolos UI evolved incrementally and iteratively: we added new pages with novel visualisations, improved existing visualisations and UI elements, and occasionally removed functionality that overly complicated the UI. Throughout this process, we enhanced the UI by applying fundamentals from UX design.

The design process of Dolos culminated in 2022 with a UI/UX design course in collaboration with UXCoach¹. This course reinforced the fundamentals of UX design: applying design principles, creating design prototypes, and validating a design with users. During this course, we

¹www.uxcoach.be

used Dolos as a case study, resulting in a redesign of the UI, which has continued to evolve into the current Dolos UI.

4.1.1. Usability Testing

One effective way to validate and improve UX design is through the use of *usability testing*, also known as *user testing*. In these tests, a researcher evaluating a design invites participants to use the tool to perform specific tasks. The researcher often asks the participant to think out loud and explain their rationale behind their interactions with the UI. Meanwhile, the researcher observes these actions and notes any challenges the participants face, along with their feedback.

This approach is well-suited to assess whether the UX design is as effective as the designer intended. The designer structures the UI elements based on their assumptions about user expectations. However, users have to *discover* this structure, and this rarely matches perfectly with the designer's hypothesis. The issue is that designers cannot effectively test their own designs, as they already have a good understanding of the UI structure.

During the summer of 2022, the Dolos team conducted extensive user studies to identify improvements for the Dolos UI. Participants used Dolos to search whether there was any plagiarism in demo datasets. We observed that participants struggled with UI elements focussing on file pairs, often preferring to focus on single submissions instead. The introduction of the submission detail page (section 4.7), along with refinements of the overview page (section 4.3) and the cluster detail page (section 4.5.1), are direct and valuable outcomes of this study. We consider these views among the most useful for gathering detailed information about potential plagiarism.

4.1.2. Philosophy

The philosophy behind the Dolos UI comprises two major parts: UI design, which ensures that UI elements are clear and form a coherent, modern interface, and UX design, which defines how the users interact with Dolos.

Use Interface (ui)

The Dolos UI aims to provide a coherent and modern user interface by adhering to the Material Design specification². Developed by Google and applied in many of their products, this specification leverages print design principles, such as typography, grids, space, scale, colour, and imagery, to create contemporary user interfaces.

Material Design employs a metaphor of physical material, designing UIs that resemble paper and ink, using colour, shadows, and edges to convey depth. By emphasising bold typography, vibrant colours and intentional white-space, the UI is purposeful and directs the user's attention to the most critical information. Unlike physical paper, digital interfaces can incorporate motion: animations and transitions provide meaningful context and feedback, enhancing the overall user experience.

Cards are fundamental UI elements in Material Design, organising information concisely and effectively. Layouts typically feature a collapsible navigation drawer that slides in from the side of the screen, offering quick access to an application's main pages.

Dolos utilises UI components from the Vuetify³ component library, which includes components and layouts designed according to the Material Design specification. Section 5.5 explains in more detail how we incorporated these components into the Dolos UI.

User Experience (ux)

To consider the UX of the Dolos UI, we first clearly define the goal. Dolos's primary purpose is supporting educators to prevent, detect, and prosecute source code plagiarism. However, the similarity detection pipeline only identifies similarities between files in the analysed dataset. Therefore, the dashboards do not merely display these similarities, but use them to provide visualisations that aid in discovering potential plagiarism. Additionally, we target two primary scenarios where instructors will use Dolos:

- **Tests and exams (summative assessments):** Students individually solve programming exercises, often under supervision, to evaluate their learning. Solutions should be independent, so we expect low similarity between non-plagiarised submissions.

²m3.material.io

³vuetifyjs.com

Plagiarism during official evaluations is a serious academic misconduct, so instructors are interested in each suspicious submission. In this situation, Dolos should always highlight pairs with abnormally similar code fragments.

- **Training exercises (formative assessment):** Students develop their programming skills by solving exercises over a longer period. Collaboration is common, with students helping each other by sharing ideas or code fragments, leading to higher similarities between submissions. In this context, instructors monitor the level of collaboration to safeguard the learning process. Thus, Dolos should also provide a global overview of the collaboration levels.

With these two use cases in mind, we developed a dashboard that allows exploring a dataset of source code files to determine which files contain similar code fragments that could indicate plagiarism. In addition to these primary use cases, Dolos adheres to several extra design goals:

Dolos should minimize obstacles for plagiarism detection. One of the reasons for developing Dolos is to encourage instructors to check for academic dishonesty. It is therefore crucial that there are no barriers preventing instructors from performing a quick similarity check. We have incrementally removed these obstacles: transitioning from a CLI to a web UI, releasing a web server, and integrating with external platforms (section 5.6.2). Instructors can now perform a plagiarism check from within the Dodona exercise platform with a single click of a button.

Instructors should be able to use Dolos efficiently. We aim to reduce the time spent checking for plagiarism. The first thing most instructors want to know is whether any plagiarism has occurred. Dolos's landing page prioritises displaying information that supports this decision, making suspicious cases immediately visible. If an instructor suspects plagiarism, the Dolos UI should provide further leads to inspect these cases closely, allowing them to confirm or dismiss their suspicions.

Dolos is an aid, not a judge. It does not decide whether high similarity is the result of plagiarism; that responsibility lies with the instructor. We do not attempt to influence this decision by preemptively flagging submissions. However, we do suggest a similarity threshold above which certain submissions are more visible than others, but the user can freely adjust this threshold.

Dolos deters plagiarism. When instructors indicate they are using a similarity checker, this should already dissuade students from committing plagiarism. Hopefully, demonstrating Dolos’s capabilities decreases the students’ perceived opportunity to plagiarise (section 1.2.2). Of course, while doing this, Dolos should respect students’ privacy and hide identifying information.

Dolos should be flexible. Users’ needs can vary greatly, and Dolos should not inhibit any use cases. While we do focus on providing access to a similarity checker with sensible default settings, we also support alternative approaches for more technically included power users and developers. This facilitates other uses of Dolos. In chapter 5, we describe the various points of access Dolos provides to its similarity detection pipeline. This approach has yielded creative applications of Dolos, described in section 8.2.

4.2. General UI structure

The Dolos UI features some UI elements visible on every page for easy access to navigation and global setting.

4.2.1. Navigation

The navigation drawer on the left (figure 4.1b) slides open to provide quick access to Dolos’s main pages. In *server mode*, the top navigation option links back to the upload page, allowing users to submit a new dataset for analysis. Below this, the drawer lists the main pages, highlighting the current page with a grey outline. External links to the GitHub repository, documentation, contact email address, and the current version’s release notes appear at the bottom, offering additional information. When collapsed, the navigation drawer disappears completely on small screens to save space. On larger screens, a collapsed navigation drawer still displays page icons for quick access.

Detail pages such as the cluster detail, submission detail, and comparison pages, are not immediately accessible through the navigation menu. When visiting these pages, *breadcrumbs* at the top provide information about the page hierarchy and link back to the parent page.

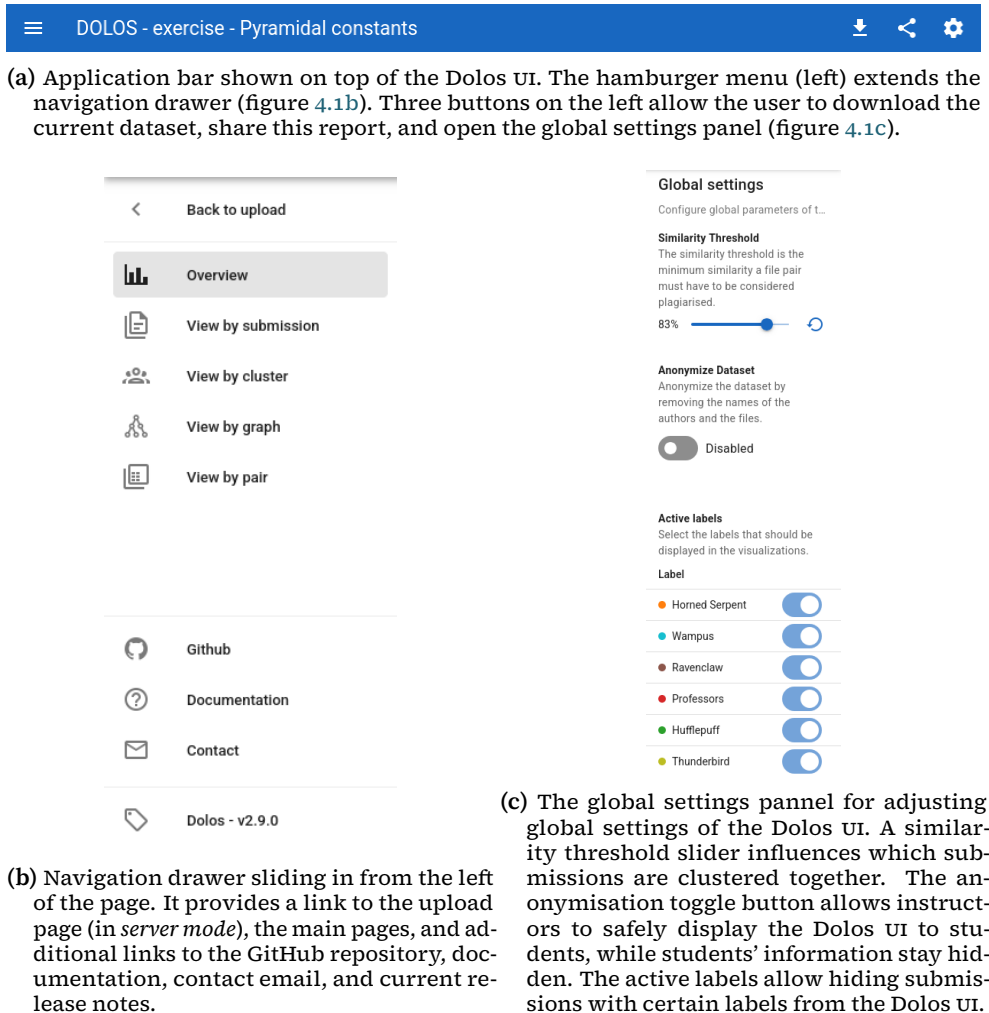


Figure 4.1. Sections of the Dolos UI visible on every page.

Application bar

An application bar, also known as a navigation bar, at the top displays the current report title on the left, aiding in distinguishing the current report when analysing multiple reports. On small screens, a hamburger menu on the left toggles the navigation drawer. The right side of the application bar features a cogwheel symbol to open the global settings pane (section 4.2.3). When the UI operates in *server mode* for the web server (section 5.5.5), the right actions of the application bar include a download button to download the current dataset, and a share button to distribute a link to the current report.

4.2.2. Metadata

Dolos allows the inclusion of metadata with submission files by providing an `info.csv` file (section 5.3.2). This metadata includes the author's name, submission timestamp, and a label. The Dolos UI progressively enhances when this metadata is included. Each label will have a colour assigned, which appears in visualisations such as the plagiarism graph (section 4.4). Other visualisations, such as the cluster timeline (section 4.5.1) require specific metadata and will only appear when this data is available. As metadata can provide valuable additional insights, the UI reminds users to include it when possible.

4.2.3. Global settings

The settings panel (figure 4.1c) allows users to modify the behaviour of the Dolos UI by adjusting global settings. These settings apply across all pages, but can also be quickly reverted.

Similarity threshold

The similarity threshold determines a threshold above which a similarity between a pair of submissions is considered suspicious. Submission pairs exceeding this threshold will receive more prominent attention in the UI. This threshold also dictates which submissions are connected in the plagiarism graph (section 4.4), and how submissions group together to form clusters (section 4.5).

Specialised pages, such as the plagiarism graph, feature an additional and more prominent slider. Manipulating this additional slider has the same effect as using the slider in the global settings panel.

Anonymisation

Demonstrating Dolos's capabilities to a group of students can reduce their perceived opportunity to commit plagiarism and evade detection, helping in preventing plagiarism. However, instructors may wish to avoid revealing students' personal information during such demonstrations. To accommodate this, Dolos features a global anonymisation toggle. When enabled, Dolos replaces authors' names, labels, and file names with randomised pseudonyms, concealing this personal information. The UI consistently uses these pseudonyms across its pages, allowing teachers to track individual students throughout the report.

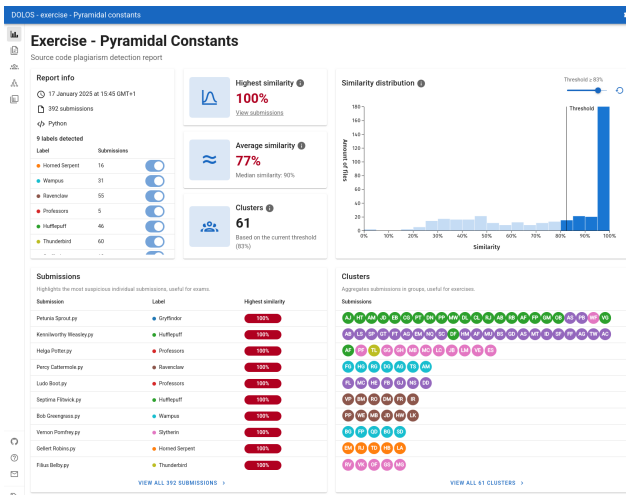
Labels

Dolos allows the inclusion of a label with each submission. While the meaning of this label is opaque for Dolos, instructors typically use labels to organise students into various groups. The UI visually groups submissions sharing the same label together by assigning a distinct colour to every label. The global settings feature a toggle button next to the label list, enabling users to filter out certain labels. For example, instructors can label submissions of teaching assistants in the dataset to easily exclude them from the report.

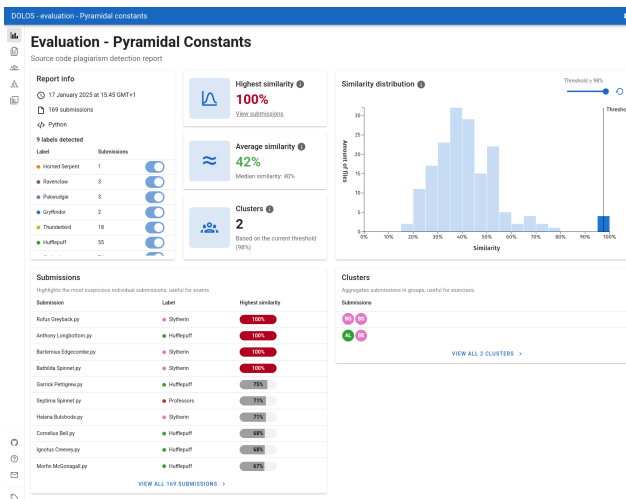
4.3. Overview

Exploration of the complete collection of submissions begins with an overview dashboard, as shown in figure 4.2. This dashboard summarises the plagiarism detection results through its analytics and visualisations, providing an initial indication of whether plagiarism is prevalent within the analysed submissions. The collection information card, located in the top left, displays basic statistics about the collection under analysis. Colour codes, representing the highest and average pairwise similarities between submissions, appear in the top centre and bottom left. These codes indicate the level of similarity, ranging from low (green) to average (orange) and high (red).

A histogram in the top right and a list in the bottom left detail the global similarity of each source file with its nearest neighbour. The composition of clusters, depicted in the bottom right, represents submissions as circles marked with an acronym derived from the author's name,



(a) Overview page shown on the π -ramidal Constants exercise page. As the students could collaborate freely, this dataset includes many copied submissions, noticeable as a big spike at 100% in the similarity histogram. The clusters shown indicate that there was not a single source of the plagiarism. Multiple clusters have formed, often between students of the same study programme (indicated by submission colour).



(b) Overview page shown on the π -ramidal Constants evaluation page. During this evaluation, communication was not permitted, resulting in the normal distribution shown in the similarity histogram. However, two suspicious pairs stand out with 100% similarity, each pair forming a cluster with two submissions.

Figure 4.2. Screenshots of the overview page in two different scenarios: an homework assignment (figure 4.2a) and an exam (figure 4.2b). This page is the first page shown when opening a Dolos report.

coloured according to their label. Both individual submissions (bottom left) and clusters (bottom right) are ranked in descending order of plagiarism suspicion.

The web application determines an appropriate initial similarity threshold for clustering using a simple heuristic. Users can adjust this threshold either via the histogram in the top right panel or in the global settings, accessible from the far right of the top navigation bar. Additionally, all dashboards feature a shared setting to anonymise analytics and visualisations, useful for in-class demonstrations, and a label-based filtering option for the collection of source files.

In summative assessments, highly suspicious submissions become immediately visible in the submission list. When analysing submissions during formative assessments, the similarity histogram and clusters list provide an impression of the severity of plagiarism among the analysed submissions.

4.3.1. Similarity Histogram

The top left visualisation on the overview dashboard displays the similarity distribution as a histogram. For each submission, it identifies the most similar other submission. The similarity histogram organises the similarities in buckets at 5% intervals and plots the submission count for each bucket. This results in a histogram that effectively assesses the situation.

Even without collaboration among students, we expect parts of submissions to exhibit random similarities with those of other students. In cases where there is no plagiarism, we expect the histogram to approximate a bell-shaped function. The bell curve will peak around the mean similarity value of the collection. This mean similarity typically depends on the exercise complexity: simpler exercises will exhibit less variation between submissions, resulting in a higher average and a bell curve shifted towards higher similarities.

In contrast, when students collaborate or plagiarise, their submissions exhibit greater similarity to one another. This results in a histogram displaying the superposition of two bell-shaped functions, creating two distinct peaks: one around the mean similarity value expected for the exercise, and another at a higher similarity percentage due to plagiarised submissions. We leverage this property to estimate a threshold similarity value, described in section 4.3.2. This threshold

appears as a vertical black line on the histogram and is adjustable via the slider on the top left.

The number of submission pairs increases quadratically with the number of submissions in the collection. Most of these pairs exhibit low similarity, even when plagiarism is prevalent. A histogram of all pairwise similarities, rather than just the nearest neighbour for each submission, would bucket most similarities in the lower percentages, overshadowing the high similarity pairs.

4.3.2. Automatic similarity threshold estimation

A global similarity threshold determines which pairs of source files are suspect, connecting them by an edge in the plagiarism graph and influencing how the dashboard clusters submissions. The goal is to minimise false positives (suspected or clustered files that are not plagiarised) and false negatives (undetected plagiarism events). However, selecting the optimal threshold can be challenging, as this depends on various factors, including expected file size, number of students, programming language used, and the diversity of the solution space. Thus, the threshold is highly dependent on the collection of source files under analysis.

To assist educators, the web app automatically estimates an initial threshold for a given collection. This estimate assumes that the similarity distribution comprises two bell curves: one centered around the lower half, corresponding to non-plagiarised submissions, and another near the maximum, corresponding to plagiarised submissions. Dolos employs a heuristic to estimate the intersection point of these two distributions as the initial threshold.

The web app assigns the pairwise similarities to bins with an interval width of 3%. Each bin with fewer associated pairs than its adjacent bins represents a local minimum and is considered a candidate for the estimated threshold. Dolos selects the local minimum corresponding to the bin with the highest value, resulting from multiplying the square root of the associated pairs count (to discourage large local minima) with a probability density distribution centred around 80%. This heuristic is based on an educated estimate and assumptions that may not always hold true. The estimated value serves as an initial guess, but users are encouraged to modify this value based on their experience. Selecting a good estimate for the similarity threshold remains an open research question, with possible improvements discussed in section 8.3.2.

4.4. Plagiarism Graph

The plagiarism graph, shown in figure 4.3, visualises all submission comparisons as an interactive force-directed graph. Nodes represent the submissions in the collection, and pairs of nodes with similarities exceeding the threshold are connected by an edge, whose thickness corresponds to the degree of similarity. Interconnected nodes form a cluster, outlined in the most prevalent node colour within that cluster.

Clicking a node reveals an information card with supplemental details about the corresponding submission and its cluster. Users can drag nodes to their liking to arrange nodes and clusters as desired. The bottom left of the interface features a play-pause toggle button to control the simulation.

If the metadata included in the analysis provides submission labels, nodes are coloured according to this label. A legend positioned in the top right reveals the relationship between node colours and labels. Clicking on an item in this legend temporarily filters out all submissions with that label from the analysis and removes the corresponding nodes from the plagiarism graph.

The similarity threshold can be adjusted interactively using a slider. A checkbox controls whether singletons — nodes not connected by an edge to any other node — should be displayed. Revealing singletons provides a better understanding of the sparsity of the solution space for a programming exercise and the prevalence of plagiarism within the submitted solutions.

The plagiarism graph offers effective visualisations for monitoring formative assessments, allowing teachers to interact intuitively with the analysis results. Cluster sizes provide an immediate impression of the collaboration level within the dataset. Revealing the plagiarism graph (with anonymisation turned on) also serves as an effective prevention method, as students can identify themselves and their friend group within the clusters. By using the study program as labels, we observe that most students share submissions with direct colleagues.

The plagiarism graph has undergone the most iterations of all visualisations. Section 5.5.3 provides more details on our specific implementation of the plagiarism graph.

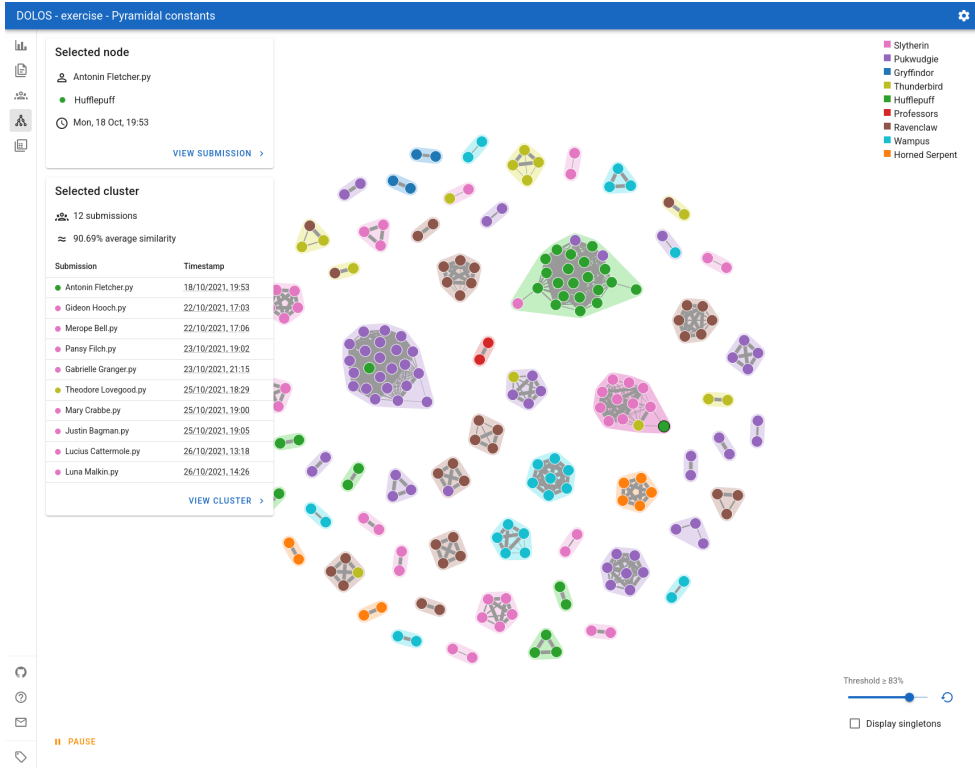


Figure 4.3. Plagiarism graph showing suspected cases of plagiarism within the same collection of source files used for figure 4.2a. Each node represents a source file and has a colour that corresponds to its file labels. The legend (top right) can be used to include or exclude files from the graph by label. Edges connect nodes whose pairwise similarity exceeds an adjustable threshold (bottom right), set at 83% global similarity. Clusters of connected nodes are grouped within regions whose background colour reflects the dominant colour of the cluster nodes. Cards on the left provide information on the current selected node and cluster (green node in the rightmost pink cluster).

4.5. Clusters

Traditional source code plagiarism tools typically report potential plagiarism from the perspective of individual files or file pairs. However, larger groups of collaborating students quickly result in an unmanageable list of file pairs (e.g. 10 students closely working together result in 45 file pairs), which may be scattered across a list of reported file pairs. In contrast, visualising the same data as a clustered graph feels intuitive and provides a natural transition to the clustering pages.

The concept of a cluster is now an integral part of the Dolos dashboard design as a separate hierarchical level. This feature helps distinguish between peer-to-peer plagiarism events (two students sharing code) and broadcast events (larger groups of students sharing code, e.g. via social media). The cluster dashboard reconstructs the distribution timeline based on submission timestamps. This feature is useful for tracking the original author or observing how the distribution process has evolved over time.

Dolos groups submissions in clusters using the single-linkage clustering algorithm. This algorithm iteratively merges two clusters when a pair exists between nodes in the two clusters exceeding the global similarity threshold. Submissions that do not have an associated pair with a similarity exceeding this threshold do not belong to a cluster, instead they are considered *singletons*. This plagiarism graph implicitly applies this algorithm, where groups of interconnected groups form the clusters considered by Dolos.

The Dolos UI provides a list of all clusters for the current similarity threshold on the cluster *View by cluster* page (figure 4.4). Each cluster row includes a summary of the composition shown by circles with the author's initials, coloured according to the submission label. The cluster table also shows the average similarity and the number of submissions included in that cluster. Clicking on a row directs the user to the cluster detail page (figure 4.5).

4.5.1. Cluster detail

The cluster detail page allows further inspection of the current cluster. A table lists submissions and pairs included in this cluster, together with supporting information. The dashboard further visualises this information in three visualisations: the cluster graph, cluster timeline, and cluster heatmap.

Clusters

All clusters, formed by the similarity threshold.

Submissions	Average similarity	Size ↓
AJ HT AM AD EB CG FT GN PP MW DL CL RJ AD RB AF FP GH GU AS PB HY YG	91%	23 submissions
AB LS SP CT FT AQ EM NG SC DF HM AF MU BG GD AS MT ID SP FF AG TW AC	95%	23 submissions
AF PF TL GG GH MB MC LG JB LM VE ES	91%	12 submissions
FG HG BG DG AG TS AM	100%	7 submissions
FL MC HE FB GJ NZ GD	97%	7 submissions
VP BM ND DM FR BS	96%	6 submissions
PP WE MB AD HW LX	95%	6 submissions
BG FP QD BG SD	93%	5 submissions
EM RJ TD HB LA	100%	5 submissions
RV VK OF GS MG	100%	5 submissions
RP LT AD WR LR	96%	5 submissions
EB AF DB QW GS	94%	5 submissions
FB KO AM GC	92%	4 submissions
AG RM FB RB	94%	4 submissions
JP AD DV MW	100%	4 submissions

Figure 4.4. *View by cluster* page listing the clusters for the current similarity threshold. Circles with author initials, coloured according to the submission label, show each cluster's composition. Average similarity and number of submissions is located on the right of each row.

The cluster graph is a miniature version of the plagiarism graph, only including the submissions of the current cluster.

Cluster timeline

When the current collection includes metadata with the submission timestamps, a cluster timeline appears in the bottom left. This cluster timeline places a node on the timeline according to the submission timestamp, visualising the order of submissions. This information can help in discerning how plagiarised copies spread throughout the students involved in the current cluster.

The cluster on figure 4.5 clearly shows how Antonin Fletcher (labeled in green) submitted their solution on 19 October, with plenty of time to spare on the deadline of 26 October at 22:00. Other students from another study programme (Slytherin, labeled in pink) started handing in similar submissions after 23 October. We also observe four students submitting very close to the deadline, indicating a certain urgency.

Note that this Dolos report only includes the latest submission of each student. Students handing in plagiarised submissions and incrementally handing in new submissions with more obfuscations are not included in this report.

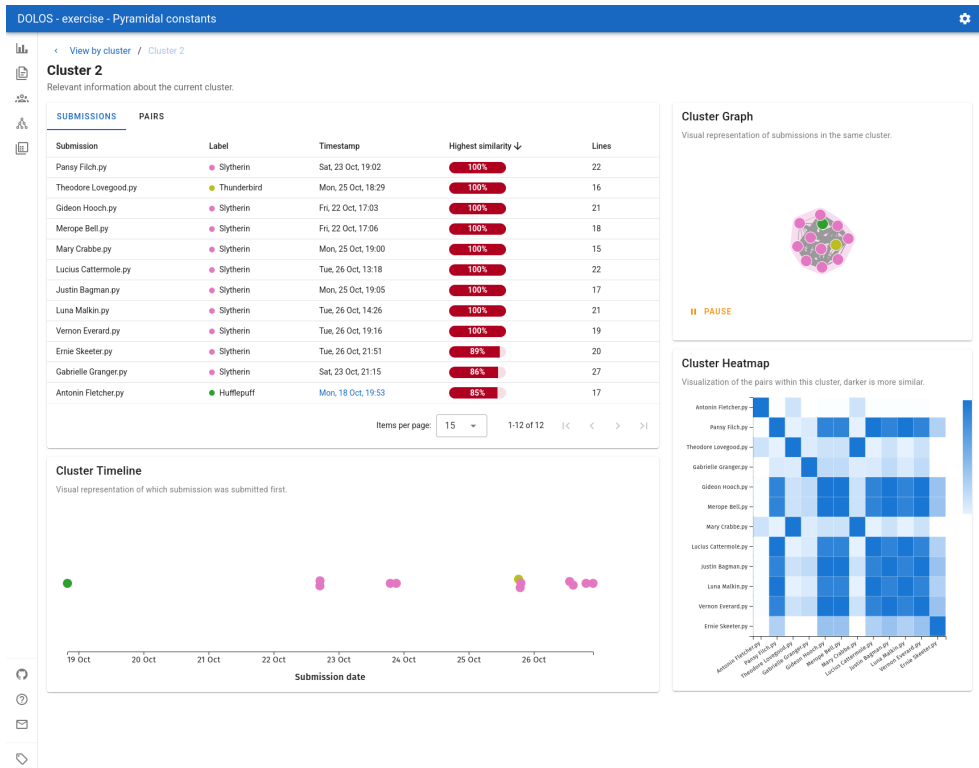


Figure 4.5. Cluster detail page presenting information on a group of similar submissions. A table with included submissions and pairs (top left), shows submission timestamps, with the first submission timestamp highlighted in blue. The cluster timeline (bottom left) provides a detailed timeline of the submission order. A plagiarism graph (top right) shows a visual representation of the current cluster. The cluster heatmap (bottom right) visualises pairwise similarities between submissions in this cluster.

Cluster heatmap

The cluster heatmap, shown in the bottom right, visualises all pairwise similarities using a heatmap visualisation. A square cell indicates the similarity between the submissions corresponding to its row and column, a darker colour meaning a higher similarity. The visualisation is symmetric around the diagonal, with cells on the diagonal itself representing 100% similarity between a submission with itself.

The heatmap in figure 4.5 reveals that the first submitter (Antonin) only appears to have a high similarity with Theodore Lovehood and Mary Crabbe. Other submissions in this cluster have very high pairwise similarities. This could indicate that Antonin obfuscated their original solution before sharing it with a colleague, after which that solution spread throughout the population with only minor adjustments.

4.6. Pairs

The primary output of Dolos's source code similarity detection pipeline is a collection of fingerprints shared among the submissions under analysis. A *pair* aggregates all information about the fingerprints shared between two submissions, facilitating further inspection to determine whether plagiarism has occurred. This pair contains all information needed to compute the shared *fragments*: the actual code blocks producing identical fingerprints. Three metrics summarise the shared fingerprints between a pair to indicate potential plagiarism: total overlap, similarity, and longest fragment:

- **Total overlap:** represents the total number of fingerprints shared between a pair, providing an absolute measure of the amount of shared code.
- **Similarity:** indicates the fraction of shared fingerprints relative to the total number of fingerprints in the two submissions, revealing how similar the two files are.
- **Longest fragment:** denotes the maximum number of consecutive fingerprints shared between the submission pair, calculated as the longest common subsequence (section 3.3.3)

We give more details on how these metrics are calculated in section 3.3.

The *View by pairs* page lists all pairs from the current analysis in a paginated table (figure 4.6). Each row displays the two submissions in the pair, along with three similarity metrics: similarity, longest

4.6. Pairs

File pairs

A pair is a set of 2 files that are compared for similarity and matching code fragments.



Left file	Right file	Similarity ↓	Longest fragment	Total overlap
Rufus Greyback.py	Bathilda Spinnet.py	100%	4	8
Anthony Longbottom.py	Bartemius Edgecombe.py	100%	51	102
Rufus Greyback.py	Garrick Pettigrew.py	75%	4	9
Garrick Pettigrew.py	Bathilda Spinnet.py	75%	4	9
Septima Spinnet.py	Helena Bulstrode.py	71%	23	76
Cornelius Bell.py	Ignotus Creevey.py	68%	23	94
Morfin McGonagall.py	James Johnson.py	67%	36	289
Alice Runcom.py	Lord Bell.py	63%	14	70
Horace Longbottom.py	Lord Bell.py	59%	14	61
Alice Runcom.py	Horace Longbottom.py	58%	14	57
Gilderoy Karkaroff.py	Angelina Cattermole.py	57%	18	51
Percy Cattermole.py	Poppy Fudge.py	56%	13	45
Albert Thicknesse.py	Wilhelmina Prince.py	55%	19	48
Newt Pettigrew.py	Justin Podmore.py	55%	9	28
Petunia Sprout.py	Xenophilus Moody.py	53%	26	52
Augustus Bagman.py	Gilderoy Karkaroff.py	53%	23	48
Petunia Sprout.py	Augusta Jordan.py	53%	13	43
Zacharias Karkaroff.py	Horace Longbottom.py	53%	8	49
Everard Dumbledore.py	Mary Pettigrew.py	52%	8	48
Fenrir Pomfrey.py	Filius Greyback.py	52%	15	32
Petunia Sprout.py	Mary Pettigrew.py	51%	22	49
Petunia Sprout.py	Everard Dumbledore.py	51%	8	40
Millicent Barty.py	Gilderoy Karkaroff.py	51%	6	44
Angelina Cattermole.py	Augusta Jordan.py	51%	7	44
Petunia Sprout.py	Merope Bell.py	51%	18	40

Items per page: 25 1-25 of 14196 < > >|

Figure 4.6. *View by pairs* page listing all pairs in a sortable and searchable table. The table columns include both submissions included in this pair, together with the three similarity metrics.

fragment, and total overlap. Users can sort the table by any metric, as each metric provides different insights. Clicking on a row directs the user to the comparison page.

4.6.1. Pairwise comparison

The pairwise comparison page, or comparison page for short, allows direct comparison of two submissions by showing their source files side by side (figure 4.7). Instructors can examine the source files for clues of plagiarism. Two editor modes highlight either differences (diff mode) or similarities (match mode). The Dolos UI selects the default mode based on the current similarity percentage. In addition to the code editor, the page presents the three similarity metrics and any supporting information available, such as submission time and label.

Comparing Anthony Longbottom.py with Bartemius Edgecombe.py

The compare view matches code fragments & differences between 2 files.

MATCHES | DIFF

Similarity: 100% | Longest fragment: 51 | Total overlap: 102

Anthony Longbottom.py | Hufflepuff | 6/11/2020, 17:43 | VIEW SUBMISSION

```

1 # redacted
2 alfa = input()
3 hoogte = int(input())
4
5 # redacted
6 alfa = alfa.replace(".", "")
7
8 # redacted
9 alfa = alfa.replace("0", "")
10
11 # redacted
12 alfa = int(alfa)
13 alfa = abs(alfa)
14
15 print(alfa)
16
17 alfa = str(alfa)
18 n = 1
19 lijst1 = list(alfa)
20 print(int(lijst1[0]))
21
22 while hoogte != 0:
23     som = int(lijst1[n]) + int(lijst1[n+1])
24     n = n + 1
25     if n == len(lijst1):
26         break
27     hoogte = hoogte - 1
28
29 print(som, end= " ")

```

Bartemius Edgecombe.py | Slytherin | 6/11/2020, 17:53 | VIEW SUBMISSION

```

1 # redacted
2 alfa = input()
3 hoogte = int(input())
4
5 # redacted
6 alfa = alfa.replace(".", "")
7
8 # redacted
9 alfa = alfa.replace("0", "")
10
11 # redacted
12 alfa = int(alfa)
13 alfa = abs(alfa)
14
15 print(alfa)
16
17 alfa = str(alfa) # redacted
18 n = 1
19 lijst1 = list(alfa) # redacted
20 print(int(lijst1[0])) # redacted
21
22 # redacted
23 while hoogte != 0:
24     som = int(lijst1[n]) + int(lijst1[n+1])
25     n = n + 1
26     if n == len(lijst1):
27         break
28     hoogte = hoogte - 1
29
30 print(som, end= " ")

```

(a) Pairwise matches

< Pairs / Anthony Longbottom.py & Bartemius Edgecombe.py

Comparing Anthony Longbottom.py with Bartemius Edgecombe.py

The compare view matches code fragments & differences between 2 files.

MATCHES | **DIFF**

The diff view has been automatically selected, as the files have a similarity >= 80%.

Similarity: 100% | Longest fragment: 51 | Total overlap: 102

Anthony Longbottom.py | Hufflepuff | 6/11/2020, 17:43 | VIEW SUBMISSION

```

1 # redacted
2 alfa = input()
3 hoogte = int(input())
4
5 # redacted
6 alfa = alfa.replace(".", "")
7
8 # redacted
9 alfa = alfa.replace("0", "")
10
11 # redacted
12 alfa = int(alfa)
13 alfa = abs(alfa)
14
15 print(alfa)
16
17 alfa = str(alfa)
18 n = 1
19 lijst1 = list(alfa)
20 print(int(lijst1[0]))
21
22 while hoogte != 0:
23     som = int(lijst1[n]) + int(lijst1[n+1])
24     n = n + 1
25     if n == len(lijst1):
26         break
27     hoogte = hoogte - 1
28
29 print(som, end= " ")

```

Bartemius Edgecombe.py | Slytherin | 6/11/2020, 17:53 | VIEW SUBMISSION

```

1 # redacted
2 alfa = input()
3 hoogte = int(input())
4
5 # redacted
6 alfa = alfa.replace(".", "")
7
8 # redacted
9 alfa = alfa.replace("0", "")
10
11 # redacted
12 alfa = int(alfa)
13 alfa = abs(alfa)
14
15 print(alfa)
16
17 alfa = str(alfa) # redacted
18 n = 1
19 lijst1 = list(alfa) # redacted
20 print(int(lijst1[0])) # redacted
21
22 # redacted
23 while hoogte != 0:
24     som = int(lijst1[n]) + int(lijst1[n+1])
25     n = n + 1
26     if n == len(lijst1):
27         break
28     hoogte = hoogte - 1
29
30 print(som, end= " ")

```

(b) Pairwise diff

Figure 4.7. Screenshots of the comparison page with its two editor modes: *matches* and *diff*. This screenshot shows a suspicious pair with 100% similarity from the π -ramidal constants evaluation dataset. The two source files are nearly identical, their only difference is the presence of some comments.

Pair matches

The *matches mode* of the editor on the comparison page (figure 5.2a) highlights code fragments sharing identical fingerprints. Hovering over and selecting highlighted code blocks in one file highlights the corresponding matched code blocks in the other file. The mode demonstrates which code fragments Dolos deems similar between the two source files.

This mode is automatically selected when the current pair's similarity falls below 80%. For high similarity percentages, the source files share most of their code, and in extreme cases, the entire code may be highlighted as one block, which is less meaningful. This mode is most effective for lower similarity percentages, immediately highlighting the corresponding parts when only a small portion of a source file is plagiarised.

As this mode uses fingerprints to construct the matching code blocks, and these fingerprints are designed to resist obfuscations, it is not ideal revealing which obfuscations students have applied to evade detection.

The scrollbar of this mode also serves as a minimap of the source code, indicating which parts of the source files are highlighted. This is mostly useful when the source files are long and do not fit within the viewport.

Pair diff

The *diff mode* employs a code differ to highlight exact differences between the two source files in the pair. This code differ highlights the differences by showing how to go from the code of one file to the other. One side highlights code fragments in red that need removal, while the other side highlights those fragments in green that need addition. When only parts of a line change, the entire line is highlighted in a low intensity, with the changed parts fully highlighted.

The diff mode is automatically selected when similarity exceeds 80%. For such pairs, it is more meaningful to focus on differences rather than similarities. As this mode operates on the exact code rather than fingerprints, student obfuscations become more apparent: added comments or renamed variables are immediately visible. Like the matches mode, the scrollbar serves as a minimap to indicate where the files differ.

DOLOS - exercise - Pyramidal constants

Submissions

All analyzed submissions with their highest similarity.

Submission	Label	Timestamp	Highest similarity ↓	Lines
Kingsley Goldstein.py	Thunderbird	Mon, 25 Oct, 16:36	84%	61
Salazar Dolohov.py	Wampus	Tue, 26 Oct, 09:34	84%	16
Elphias Macnair.py	Pukwudgie	Mon, 25 Oct, 20:55	83%	32
Emmeline Bones.py	Ravenclaw	Mon, 25 Oct, 15:35	82%	18
Heromine Shackebolt.py	Ravenclaw	Mon, 25 Oct, 15:46	82%	17
Alicia Diggle.py	Gryffindor	Tue, 26 Oct, 14:33	82%	28
Luna Smith.py	Ravenclaw	Mon, 18 Oct, 15:25	80%	27
Millicent Macnair.py	Ravenclaw	Mon, 25 Oct, 16:09	80%	20
Horace Longbottom.py	Pukwudgie	Tue, 19 Oct, 17:43	80%	17
Cho Riddle.py	Ravenclaw	Tue, 26 Oct, 15:45	80%	18
Gregory Vane.py	Pukwudgie	Tue, 26 Oct, 20:35	80%	15
Myrtle Davies.py	Pukwudgie	Mon, 25 Oct, 20:52	80%	15
Susan Clearwater.py	Ravenclaw	Mon, 18 Oct, 16:37	79%	22
Severus Nott.py	Pukwudgie	Sun, 24 Oct, 21:15	79%	23
Wilhelmina Parkinson.py	Slytherin	Tue, 26 Oct, 21:12	78%	18
Bill Krum.py	Slytherin	Tue, 26 Oct, 14:23	76%	21
Helena Bulstrode.py	Horned Serpent	Sun, 24 Oct, 10:10	76%	19
Arthur Bell.py	Wampus	Sat, 23 Oct, 16:51	76%	18
Draco Gregorovitch.py	Thunderbird	Sat, 16 Oct, 10:19	76%	35
Draco Bulstrode.py	Thunderbird	Sat, 23 Oct, 11:52	76%	28
Madam Jordan.py	Thunderbird	Tue, 26 Oct, 18:10	76%	21
Salazar Slughorn.py	Professors	Tue, 26 Oct, 16:03	76%	37
Graham Peverell.py	Horned Serpent	Tue, 26 Oct, 20:04	76%	39
Morfin Whisp.py	Slytherin	Tue, 26 Oct, 20:42	75%	33

Items per page: 25 226-250 of 392

Figure 4.8. *View by submission* page listing the submissions in the current collection.

4.7. Submissions

One of the angles Dolos provides to inspect a collection of source files from, is from the perspective of a single submission. Even though the primary output of the source code plagiarism detection pipeline is pairwise similarities - it is much more intuitive for instructors to inspect single submissions. The Dolos UI provides the *View by submission* page with a searchable table listing all submissions included in the current collection as rows (figure 4.8). This table includes a submission's label, timestamp, total lines of code in the source file, and the highest similarity — the similarity of the nearest-neighbor submission by similarity.

4.7.1. Submission detail

Clicking on a row of the submission list table brings the user to the submission detail page (figure 4.9). This page provides detailed information on the current submission and its relation to other submissions. A card in the top left displays known metadata about this submission: the label, file name, and submission timestamp. A compare table (top right) lists all other submissions, ordered by pairwise similarity with the current submission. This table includes all information included in the *View by submission* table, and adds an indication on the cluster: whether the other submission is in the same cluster (filled icon), another cluster (outlined icon), or not in any cluster (no icon). A link to compare this submission with the other provides quick action to the pairwise comparison page (section 4.6.1).

If the current submission is included in a cluster, this page also features the cluster timeline (center left) and cluster graph (top right) from the corresponding cluster detail page section 4.5.1. A similarity histogram and longest fragment histogram (bottom right) highlight how the metrics of the current submission relate to other submissions. These histograms allow the instructor to see whether the similarity or longest fragment value is abnormally high. The bottom left of the submission detail page features the source file contents of this submission in a code viewer.

4.8. Evolution of the UI

We did not build Dolos UI overnight; it has undergone multiple incremental iterations, each enhancing the current state. To illustrate the UI's evolution, we briefly explore its development here. Dolos began in the summer of 2019 as a script processing data from standard input (`stdin`) and writing hashes to standard output (`stdout`). From this simple script, the project evolved into a full-fledged dashboard for similarity analysis.

This section outlines the most important milestones of the Dolos UI, along with their release versions. For a comprehensive list of releases, including features, bug fixes, and other changes, visit the Dolos release page on GitHub⁴.

⁴github.com/dodona-edu/dolos/releases

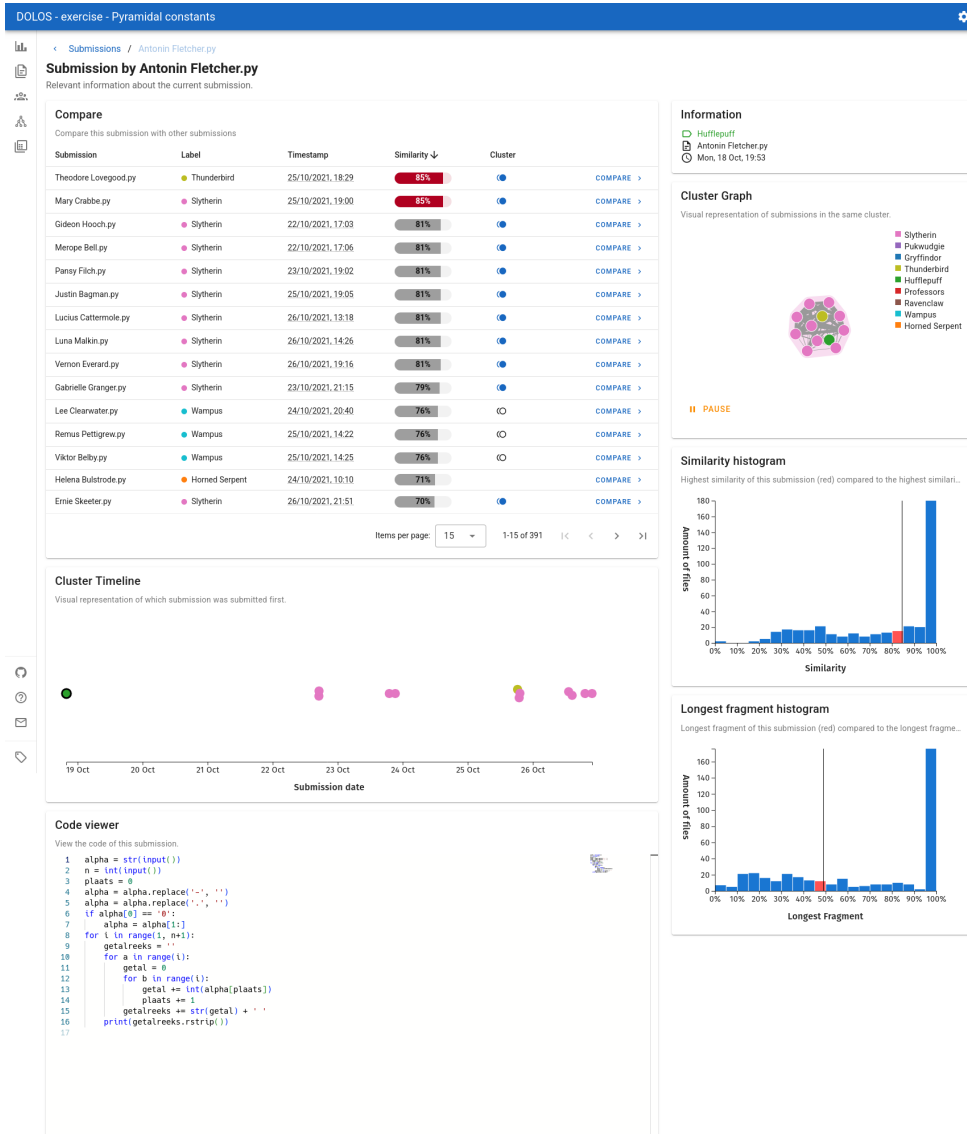


Figure 4.9. Submission detail page featuring a list of similar other submissions in the top left. This page includes the cluster timeline and cluster graph from the corresponding cluster detail page section 4.5.1. Additionally, a similarity histogram and longest fragment histogram in the bottom right visualise how the current submission's metrics compare to other submissions under analysis. The bottom left displays the current submission's code.

4.8.1. TUI before v1.0.0

Development to make Dolos user-facing began in March 2020 with the creation of a CLI. The first proper UI included in Dolos was a terminal user interface (TUI), as shown in figure 4.10. This TUI outputs analysis results to the console as a list of all file pairs, including their similarity metrics. When the input dataset contains a single file pair, or the user explicitly adds the `--compare` CLI flag, the TUI also prints the matching code fragments for each file pair.

Chapter A, which describes the CLI options, details further configurations for the TUI, such as setting a sorting order and limiting the output results. The Dolos CLI still uses the TUI as the default output format, which has changed very little since the v1.0.0 release.

4.8.2. v1.0.0 – A web UI for Dolos

Version v1.0.0, released in the summer of 2021, introduced an actual web UI for Dolos, as shown in figure 4.11. This initial UI featured three pages: the pairs listing (figure 4.11a), the comparison page (figure 4.11b), and the plagiarism graph (figure 4.11b).

This initial UI included a few features that are no longer present. The comparison page featured a toggleable list of shared *blocks* (now called *fragments*) on the right side. This list displayed the fragments and their length in fingerprints, allowing users to iterate over them while highlighting the corresponding code in the editor. Additionally, users could filter out fragments with fingerprints below a threshold count (minimum block length). We removed this functionality because users need a deep understanding of Dolos’s underpinnings to utilise it effectively. Consequently, we noticed that users rarely used this feature, leading to its removal to reduce visual complexity.

Another feature in v1.0.0 allowed users to set a status for each submission pair. This status could be one of four options: *unreviewed* (default), *innocent*, *suspicious*, and *certain plagiarism*. However, the UI did not store this status persistently, meaning users would lose their labelling efforts upon refreshing the page. Moreover, four statuses were too limited to capture the nuances of plagiarism detection. Both limitations led to the elimination of this feature.

```
user@host:/data$ dolos -l javascript *.js
```

File path	File path	Similarity	Cont. overlap	Total overlap
copy_of_sample.js	sample.js	1	37	37
another_copied_function.js	copy_of_sample.js	0.191489	4	4
another_copied_function.js	sample.js	0.191489	4	4
copied_function.js	copy_of_sample.js	0.095238	2	2
copied_function.js	sample.js	0.095238	2	2

(a) CLI output listing all file pairs including their similarity metrics.

```
user@host:/data$ dolos -l javascript sample.js copied_function.js
```

```
copied_function.js          sample.js
```

```
Absolute overlap: 2 kmers
Similarity score: 0.09523809523809523
```

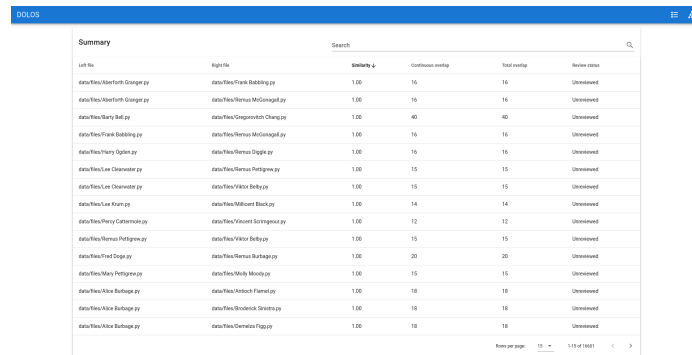
Block 1/1: 2 kmers

```
Tokens: '),(,statement_block,(,lexical_declaration,(,variable_declarator,(,identifier),(,number),(,)),),(,for_in_statement,(,identifier),(,identifier),(,statement_block,(,expression_statement,(,assignment_expression,(,identifier),(,subscript_expression,(,member_expression,(,this),(,property_identifier),(,call_expression,(,member_expression,(,this),(,property_identifier),(,arguments,(,identifier),(,call_expression,(,member_expression,(,identifier),(,property_identifier),(,arguments,(,number),(,)),),(,)),),(,)),),(,property_identifier),(,arguments,(,identifier),(,call_expression,(,member_expression,(,identifier),(,property_identifier),(,arguments,(,number),(,)),),(,)),),(,)),)'
```

```
1  /**                                12      this.combineer = combineer || ((h, v) => (h +
2  * This is a function copied from sample.js    13      }
3  */                                             14
4  function hash(s) {                          15      hash(s) {
5      let h = 0;                               16          let h = 0;
6      for (let c of s) {                       17          for (let c of s) {
7          h = this.tabel[this.combineer(h,      18              h = this.tabel[this.combineer(h,
c.charCodeAt(0))];                             c.charCodeAt(0))];
8      }                                         19      }
9      return h;                               20      return h;
10 }                                           21 }
```

(b) CLI output with a highlighted comparison of the similar code fragments of a single pair. The red list of tokens is the reconstructed list of syntax tokens that the two source files have in common. The side-by-side comparison shows the code snippets corresponding to the similar tokens and highlights the exact code in green.

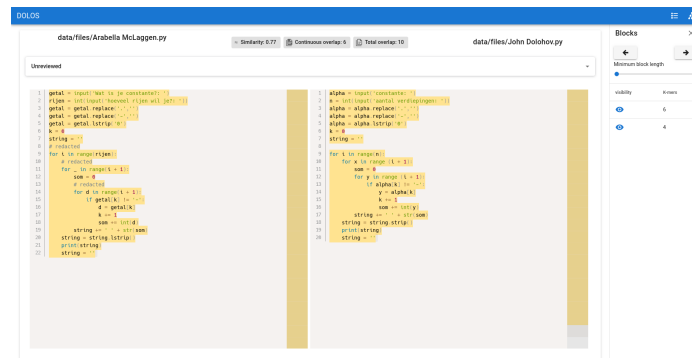
Figure 4.10. Screenshots of TUI provided by the Dolos CLI, using some sample JavaScript files as input. The comparison is shown when two files are analysed, or when the user explicitly adds the `--compare` flag to the CLI evocation. This TUI was the first proper interface included in Dolos.



The screenshot shows the 'Summary' page of the Dolos web UI. It features a table with columns: 'Left file', 'Right file', 'Similarity %', 'Continuous overlap', 'Total overlap', and 'Review status'. The table lists 18 pairs of files, each with a similarity score of 1.00 and a status of 'Unreviewed'. The files are named with paths like 'data/files/Arabella McLaggen.py' and 'data/files/John Dolohov.py'. At the bottom right, there is a pagination control showing 'Items per page: 10' and '1/1 of 1801'.

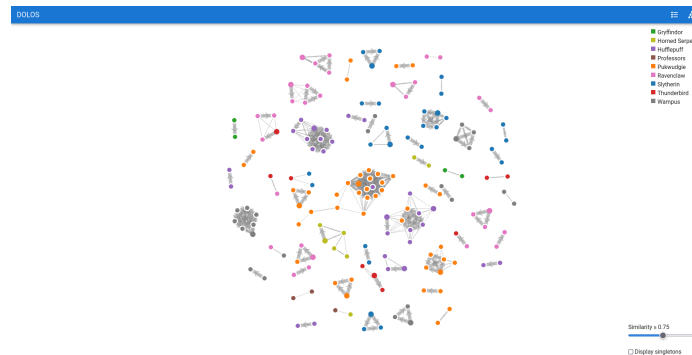
Left file	Right file	Similarity %	Continuous overlap	Total overlap	Review status
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	16	16	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	16	16	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	40	40	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	16	16	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	16	16	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	15	15	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	15	15	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	14	14	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	12	12	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	15	15	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	20	20	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	15	15	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	18	18	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	18	18	Unreviewed
data/files/Arabella McLaggen.py	data/files/John Dolohov.py	1.00	18	18	Unreviewed

(a) Pairs page listing all submission pairs in a table, along with the pairs' similarity metrics.



The screenshot shows the 'Comparison' page of the Dolos web UI. It displays two code submissions side-by-side for comparison. The left submission is 'data/files/Arabella McLaggen.py' and the right is 'data/files/John Dolohov.py'. The code is highlighted in yellow, and the interface includes a 'Blocks' panel on the right with a 'Minimum block length' slider and a 'visibility' section.

(b) Comparison page allowing close inspection of a single submission pair.



(c) Plagiarism graph visualising submissions as nodes, and connecting nodes when their submissions share a similarity above the given threshold.

Figure 4.11. Screenshots of the Dolos web UI version v1.0.0, featuring a pairs page listing all file pairs with their similarity metrics (figure 4.11a), a comparison page to study the differences between one pair in detail (figure 4.11b), and a plagiarism graph (figure 4.11c).

4.8.3. v1.6.0 – Clusters and files

During the academic year 2021–2022, Dolos underwent significant experimentation with the help of two students working on their master’s thesis. This effort led to major improvements in the source code plagiarism detection pipeline, and the addition of new pages and visualisations to the Dolos UI. Version v1.6.0, released in June 2022, incorporated these changes, as shown in figure 4.12.

This release introduced a first version of an overview page (figure 4.12a), featuring a similarity histogram. The overview page also provided information about the current report and offered quick links to the plagiarism graph and the new file analysis page. Additionally, this release added semantic analysis to the Dolos UI comparison page, detailed in section 7.3.2.

File analysis

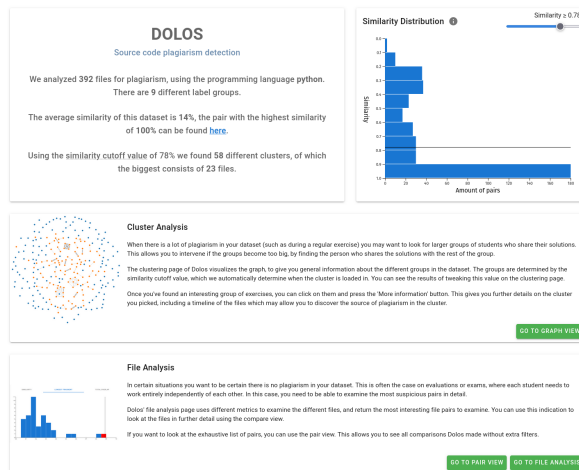
The file analysis page (figure 4.12b) served as the predecessor for both the submission list and submission detail pages described in section 4.7. The concept behind this page was to order all files according to an *interestingness* metric, which combined the three major metrics into one. Each file card displayed a histogram indicating its relative similarity, longest fragment, or total overlap metric, based on the most important metric as deemed by the *interestingness* score. Each card provided basic information about the file and linked to the relevant file pair for further inspection.

However, this page had some limitations. Each file appeared only once, with a link to compare it with the most similar other file. Files with multiple suspicious links to other submissions would thus reveal only one suspicious connection, hiding the others. Additionally, the use of cards instead of a quickly scrollable table or list made searching for suspicious files more challenging. Despite these limitations, this page marked the beginning of a shift in focus from pairs to submissions.

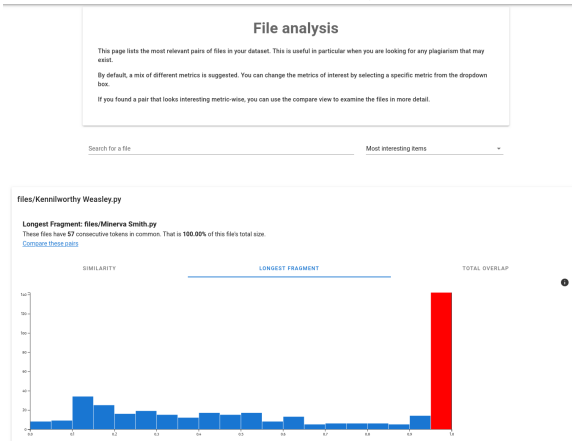
Clusters

Version v1.6.0 formally introduced the concept of clusters in the UI by adding a list of clusters positioned below the plagiarism graph (figure 4.12c). This list provided an alternative, more structured view of the content displayed in the plagiarism graph by sorting the clusters by size. Each cluster in the list was collapsible, and featured three cluster

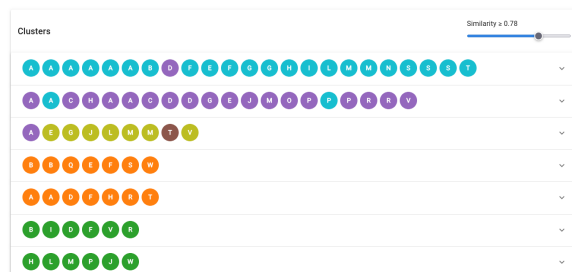
4.8. Evolution of the UI



(a) Overview page featuring a similarity distribution histogram.



(b) File analysis page listing all files in a card with a histogram based on the most significant metric from similarity, longest fragment, or total overlap.



(c) Cluster list shown below the plagiarism graph. Circles including submission author's initials give an indication on the cluster's contents.

Figure 4.12. Screenshots of the new pages in version v1.6.0 of the Dolos UI.

visualisations still present in the current UI: a timeline (figure 4.13a), a graph (figure 4.13b), and a heatmap (figure 4.13c).

However, this page suffered from a discoverability issue, as it was positioned below the plagiarism graph, it required users to scroll down to find it. Although we added an arrow to address this, user studies indicated that the page could still go unnoticed. This ultimately led to the introduction of the separate cluster pages described in section 4.5.

4.8.4. v2.0.0 – Major UI redesign

During the summer of 2022, after organising a summer course on UI/UX design and conducting user studies (described in section 4.1), we redesigned the Dolos UI following proper design fundamentals. These significant changes warranted a major version bump, leading to the release of Dolos v2.0.0 in October 2022.

This release introduced many of the pages currently present in the UI: a redesigned overview page (section 4.3), the submission list and detail page (section 4.7), the cluster list and detail page (section 4.5), and a redesigned comparison page (section 4.6). Additionally, numerous smaller features were added, optimised, or significantly improved.

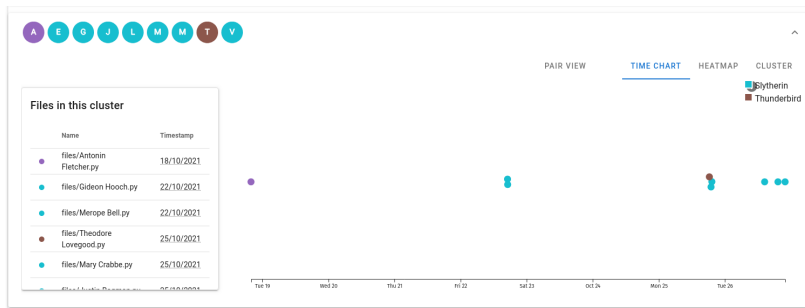
4.8.5. Further development

Since v2.0.0, the Dolos UI has remained relatively stable. With the UI reaching a point of maturity, efforts shifted towards enhancing Dolos's UX, performance and code quality. Although this resulted in only minor visual changes, improved responsiveness and stability deliver a better experience when using the dashboards.

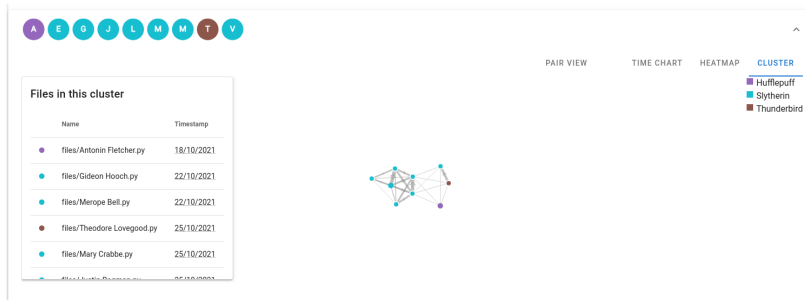
v2.2.0 – Dolos web server

One of the most significant UX issues we observed, was with users executing Dolos through its CLI. Installing and running Dolos is a technical process that requires familiarity with the command line. Not all instructors possess this familiarity, and even those who do sometimes encounter installation errors that prevent them from running Dolos.

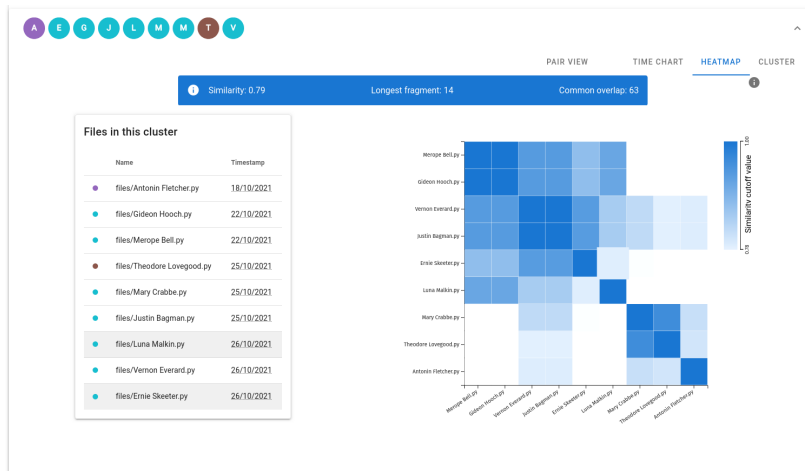
To address this, we aimed to provide an alternative that does not require local installation by enabling Dolos to run from a browser. After



(a) Cluster timeline placing each submission in this cluster on a timeline when submission timestamps are available.



(b) Cluster graph illustrating the interconnected submissions in this cluster as a force-directed graph.



(c) Cluster heatmap visualising pairwise similarities between submissions in a cluster.

Figure 4.13. Cluster visualisations add in Dolos v1.6.0. These visualisations served as predecessors for visualisations described in section 4.5.

iterating through several prototypes, we introduced a publicly available Dolos web server and web Application Programming Interface (API) in May 2023 when releasing Dolos v2.2.0.

v2.6.0 – External integrations

With a public API and web server in place, all the necessary building blocks were available to integrate Dolos as an external service into online learning environments. Release v2.6.0, introduced in April 2024, added the final missing piece to complete integration with the Dodona exercise platform⁵: a loading page to display while reports are being analysed on the Dolos web server.

v2.9.0 – Qualitative improvements

The latest version of Dolos as of writing, v2.9.0 released in January 2025, enhanced the comparison page by highlighting parts of the code that the source code similarity detection pipeline ignores. This includes explicitly provided template code, automatically detected template code, and comments.

⁵dodona.be

Chapter 5.

Implementation

Previous chapters of this dissertation described the theoretical and conceptual foundations underpinning the Dolos source code similarity detection pipeline. The research field of source code similarity detection has a strong focus on building prototypes for studying the algorithms. The review by Novak et al. (2019) notes this focus on prototypes:

In spite of the large production of tools in recent years, most of the tools are not available to the public, they are used only by the authors that developed them and are mentioned in only one article. We note that 65 of these tools are not compared in the literature, so the quality is questionable.

With Dolos, we wanted to develop a tool for broad adoption. Other research projects often focus on algorithms or techniques where implementation is only an afterthought. In this research project, the main result is the implementation of a useful source code similarity detection engine. The result is a complete *system* for similarity detection consisting of different components, each catering to different users: instructors, power-users, developers and other researchers.

This chapter describes the software architecture of Dolos and the implementation of each module. We start by explaining the general structure and software design in section 5.1, listing the different modules and their role within Dolos. The chapter then proceeds by describing the most important modules in this system.

5.1. Software architecture

Dolos is a system comprising of multiple interconnected components. Figure 5.1 illustrates the key components of Dolos. We will first give a

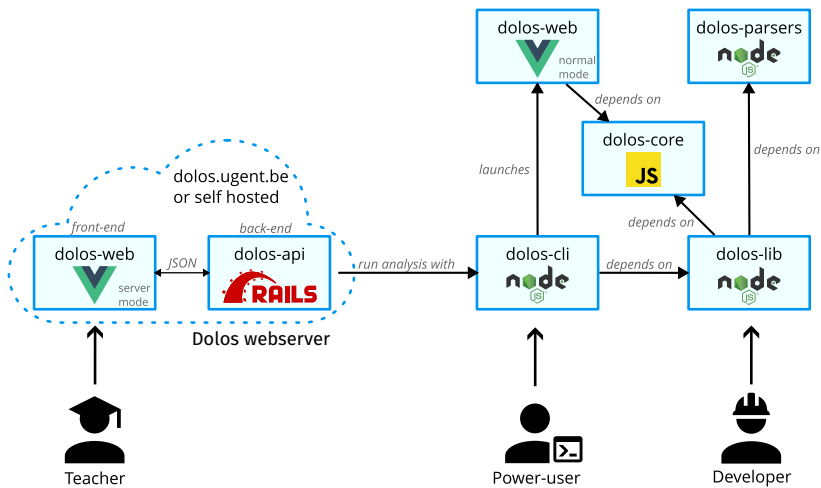


Figure 5.1. Diagram of the main components in the Dolos ecosystem and their relationships. Some components are usable in isolation, as shown by the three users interacting with the components.

high-level overview of these components before exploring their implementation details.

The core of the similarity analysis pipeline is `dolos-cli` (section 5.4), a command-line interface (CLI) that ingests source code files and presents the similarity analysis results. `dolos-cli` leverages `dolos-lib` to perform the similarity analysis (section 5.3.2). `dolos-lib` retrieves source code files from the filesystem, parses them using `dolos-parsers` (section 5.2), subsequently sending their syntax tree to the algorithms encapsulated within `dolos-core` (section 5.3.1). The `dolos-cli` can present the results through an interactive Web user interface (UI) provided by `dolos-web` (section 5.5). This interface relies on `dolos-core` to execute segments of the similarity detection pipeline on demand, thereby avoiding lengthy precomputation during analysis.

`dolos-cli`, `dolos-lib`, and `dolos-parsers` operate within the Node.js JavaScript runtime to interface with the host system, enabling file system access, network connectivity, and binary add-ons. Conversely, `dolos-core` is designed to function in browser environments devoid of the Node.js Application Programming Interface (API), hence it does not depend on the Node.js runtime.

The `dolos-api` module (section 5.6) provides a Hypertext Transfer Protocol (HTTP) JavaScript Object Notation (JSON) API to initiate a

similarity detection analysis and store its results. This API server runs the analysis jobs by invoking the `dolos-cli` in a background job. The `dolos-web` module can be built in *server mode* to interact with the JSON API of `dolos-api`. Together, `dolos-web` and `dolos-api` form the Dolos webserver.

This modular system supports various user needs. Instructors typically use the Dolos webserver for convenience, as it requires no local installation. Advanced users seeking control over the analysis or integration into grading scripts can use the Dolos CLI. Developers can embed Dolos's source code similarity detection capabilities into other applications by directly interfacing with the `dolos-lib` library.

5.1.1. Choice for TypeScript

Dolos uses TypeScript¹ as its main programming language. TypeScript adds static typing to JavaScript to improve resilience against bugs. It transpiles to JavaScript, so there is full compatibility with JavaScript itself.

The choice for TypeScript comes from its portability and popularity. Once transpiled to JavaScript, it can run in a wide range of environments: JavaScript is the only language to natively run within browser contexts, and runs directly on host systems within the Node.js² runtime environment. JavaScript offers a diverse ecosystem of software libraries published on NPM³ and keeps rapidly evolving. Because of its presence in browsers, the JavaScript runtime is heavily optimised, which keeps its runtime reasonably fast even for algorithmically demanding programs.

According to GitHub and Stack Overflow, JavaScript was the most popular programming language for 10 years, with GitHub reporting that Python just overtook JavaScript last year. Since its first release in 2012, TypeScript has gained popularity. In 2024, both Stack Overflow and GitHub place TypeScript in third position when counting programming language popularity (GitHub 2024; Stack Overflow 2024).

Using a more popular programming language increases the chances of developers contributing to the project, as they already know that programming language. Because of its popularity, JavaScript and Node.js have high-quality tooling available to help with development.

Stack Overflow technically reports TypeScript in fifth place, but HTML/Cascading Style Sheets (CSS) and SQL which precede TypeScript are not really programming languages.

¹www.typescriptlang.org

²nodejs.org

³npmjs.org

5.1.2. Repository structure

The repository hosting the Dolos source code can be found on GitHub⁴. All components related to Dolos are centralised in this monorepository using the Git version control system. The main components or modules of Dolos each have their own folder in the root of the repository:

- `parsers` containing the officially supported parsers shipped with Dolos, described in section 5.2.
- `core` provides the core algorithms of the source code similarity analysis pipeline, described in section 5.3.1.
- `lib` re-exports the core algorithms and adds Node.js code for reading from the filesystem and running the parsers, described in section 5.3.2.
- `cli` implements the command-line interface and includes code to serialise reports and launch the Web UI, described in section 5.4.
- `web` contains the graphical user interface as a single-page web application, described in section 5.5
- `api` a straightforward API server to provide the Dolos web server running at dolos.ugent.be/server described in section 5.6.
- `docs` the documentation website hosted at dolos.ugent.be, described in section 5.7.1.
- `samples` a collection of sample source code files for each supported programming language, described in section 5.7.2.

In addition to these components, the repository contains files facilitating the development of Dolos. Configuration files in the root directory ensure a shared, consistent code style between modules. To enhance the process of running, building and developing, Dolos provides files for Docker containers (section 5.7.3) and Nix environments (section 5.7.4).

5.1.3. Continuous Integration and Deployment

As Dolos has grown to a sizeable system of interdependent components that need to be executed in a wide variety of environments, it is crucial to ensure that each component functions as expected. Where relevant,

⁴github.com/dodona-edu/dolos

each component includes unit tests and integration tests to validate that the components are functioning properly.

Each time a new commit is pushed to the GitHub repository, a new job is launched using GitHub Actions to build and test each component. A total of 111 tests in `dolos-lib` and `dolos-core`, executed across the latest three versions of Node.js, helps identify issues within the JavaScript libraries. Additionally, the `dolos-api` test suite includes 16 tests, encompassing a full similarity analysis. This comprehensive test suite enables Dolos developers to swiftly identify bugs and regressions.

There is an additional workflow that checks whether all components can be packaged and released without errors. However, this additional workflow is computationally expensive, easily taking up more than 10 minutes per run, compared to the standard workflow finishing in 3 minutes and 30 seconds on average. Enabling this workflow for every push would delay the development cycle significantly. This validation workflow will thus only run when a new commit is pushed to the master branch. This only occurs when a pull request is merged or before a new release is published.

Version numbering

The NPM packages `dolos-core`, `dolos-lib`, `dolos-parsers` follow the *semantic versioning* scheme. Each package has a version number in the format `MAJOR.MINOR.PATCH`. We increase the version number of a package corresponding to how that package changes by this new release. A change of the MAJOR version number indicates an incompatible API change (a *breaking change*), a MINOR version increase indicates new functionality, and a PATCH version bump indicates a bug fix.

The other packages and container images are primarily user-facing (the CLI, web interface, and web server). As these packages are more user-facing, the concept of a *breaking change* is not really applicable. Hence, we apply a less strict scheme of semantic versioning where we only increase the MAJOR version number when Dolos has received major new features. The latest increase of the major version was after the user interface redesign in October 2022. We use this version number as the primary version of the Dolos repository.

Release flow

We trigger a new release by pushing a commit tagged with a new version number. The continuous integration will automatically execute the following tasks in a GitHub Action workflow:

- Create a new entry on the releases page⁵ on GitHub that includes a short description of the changes.
- Publish `dolos-parsers`, `dolos-core`, `dolos-lib`, `dolos-cli`, and `dolos-web` to the NPM package repository.
- Publish the `dolos-cli`, `dolos-web`, and `dolos-api` container images to the GitHub Container Registry (GHCR) container repository.
- Run the analysis on the demo datasets and publish them to the demo page on the website⁶.

5.1.4. Software Licence

We licensed the code under the Massachusetts Institute of Technology (MIT) licence which permits anyone to use, distribute and modify the source code free of charge when preserving this licence and copyright notice. There is no distinction between private and commercial use, so companies can provide paid services using Dolos without legal ramifications. The licence even permits re-licensing and re-selling the software in this repository when retaining the original licence and copyright notice. This means that Dolos is free and open-source software (FOSS), and it has been from the start.

5.2. Parser module

One of the main features of Dolos is support for many programming languages. We achieve this by building on top of Tree-sitter⁷, a parser generator tool and incremental parsing library (Brunsfeld et al. 2024). There are 24 languages with official Tree-sitter parser implementations, and an ever-growing list of 467 parsers made by the community⁸.

⁵github.com/dodona-edu/dolos/releases

⁶dolos.ugent.be/demo

⁷tree-sitter.github.io/tree-sitter

⁸You can find a list of existing parsers here: github.com/tree-sitter/tree-sitter/wiki/List-of-parsers

```

1 import { Parser } from "tree-sitter";
2 import { JavaScript } from "tree-sitter-javascript";
3
4 const parser = new Parser();
5 parser.setLanguage(JavaScript);
6
7 const sourceCode = 'let x = 1; console.log(x);';
8 const tree = parser.parse(sourceCode);

```

Listing 5.1. JavaScript code snippet that shows how to parse a string of code into a Tree-sitter concrete syntax tree. We first initialise the parser with the desired language object, after which we call the `parse`-method on a string containing source code to receive the syntax tree. To parse a different language, change the imported language object and pass that to the parser upon initialisation. Since Tree-sitter binds with compiled binaries using the Node-addon-API, this code will only work in Node.js environments.

The selection of Tree-sitter is based on its widespread adoption and active maintenance. During development, we briefly explored other alternatives:

- `vscode-textmate`⁹: a tokeniser for TextMate grammars used by the popular integrated development environment (IDE) Visual Studio Code. This parser runs using WebAssembly (WASM).
- `highlight.js`¹⁰: A widely-used syntax highlighter written in pure JavaScript.
- `Prism`¹¹ and its fork `reprism`¹²: both are pure JavaScript highlighting libraries, although neither project is actively maintained.

However, these alternatives primarily focus on syntax highlighting and do not provide ergonomic access to the underlying syntax tree, which is crucial for our needs. In addition, all three of them are regex-based parsers that cannot fully parse context-free grammars, in contrast to the canonical LR parsers generated by Tree-sitter (see section 3.1.1).

Tree-sitter generates parsers in the C programming language that are meant to be compiled on the system they will be executed on. Luckily, it also generates bindings for other programming languages, including Node.js. The provided API is quite simple and is the same for each language, as demonstrated in listing 5.1.

⁹github.com/microsoft/vscode-textmate

¹⁰github.com/highlightjs/highlight.js

¹¹github.com/PrismJS/prism

¹²github.com/tannerlinsley/reprism

This uniform API allows Dolos to be programming language-agnostic. If there is a parser for a programming language, Dolos supports that language.

5.2.1. Vendoring parsers

Tree-sitter provides access to numerous parsers that share a common API. However, each parser is an independent project, maintained and published separately from others. Adding support for a new programming language required including it as a separate dependency via NPM. Unfortunately, not all Tree-sitter parsers share the same development pace, and some are not published at all, particularly the community-maintained ones. As Dolos expanded its collection of officially supported parsers, the frequency of compatibility issues between parsers using different Tree-sitter versions increased.

The `dolos-parsers` module solves this problem by bundling Tree-sitter parsers for major programming languages into a single package. This approach is similar to *vendoring*, where third party software is copied instead of being managed as a dependency by a package manager. Rather than copying the parsers, Dolos includes their repository within its own repository using Git submodules. This process simplifies the process of receiving updates for these parsers. Additionally, if Dolos needs a custom change (e.g. fixing a bug in a parser), it can readily replace the submodule with a forked repository containing the desired changes.

Each parser has a configuration file for `node-gyp`, the Node.js build tool used to compile native add-on modules for multiple platforms. The package configuration specifies the source code files required by a binary add-on, which `node-gyp` then compiles during package installation. To streamline the build process, `dolos-parsers` employs an overarching `node-gyp` configuration file that references the configurations of all vendored parsers. The result is a single JavaScript package with node bindings for each parser.

Having a module that aggregates all parsers avoids conflicts between parsers using different Tree-sitter versions. Dolos is now much less reliant on the maintainers of individual parsers to publish new releases once an update or fix is available. In addition, adding a new parser is also much easier and faster.

However, supporting a new parser comes with a cost. Upon installation, this module builds these parsers for the host system, so `dolos-parsers` needs to include the parser code. Some parsers can take up

a sizeable amount of disk space. Adding a new parser also increases the build time. `dolos-parsers` currently includes 19 parsers. The module uses 237 MB for the code, 19 MB for the build artefacts, and it takes a bit more than a minute to build the package.

5.3. Software libraries

Initially, Dolos started as one NPM package that combined the library and CLI. Later on, when developing the Web UI, there was a need for compartmentalising the different components of Dolos. We split off `dolos-lib` from the original package to provide this encapsulation. This code was still highly dependent on Node.js to perform IO and run Tree-sitter and could not easily run in browser environments. Since the web UI of Dolos needs access to the algorithms of the similarity detection pipeline, we moved the core algorithms to a new package `dolos-core`. These two libraries make it possible for other developers to add similarity detection to custom software packages.

5.3.1. `dolos-core`

The `dolos-core` package includes most of the algorithms described in chapter 3. This package is free from runtime dependencies, ensuring that the JavaScript it provides is compatible with any JavaScript runtime. This flexibility enables the use of algorithms in both the Dolos CLI and `dolos-lib` within the Node.js JavaScript runtime, as well as using the algorithms in the web UI where the Node.js API is unavailable.

We have spent considerable effort to speed up the implemented algorithms, as they make up the core of the matching engine that drives the similarity detection. Chapter 3 discusses most algorithmic optimisations. There are some optimisations specific to our implementation, which we will discuss below:

Efficient hashing with JavaScript's number type

The built-in primitive number data type provided by JavaScript can behave a bit odd compared to other programming languages. A number represents a double-precision 64-bit floating point number according to the IEEE 754 format. Calculations using floating-point numbers can provide unexpected results and are often slower than using integers.

While JavaScript does provide some classes to handle integer numbers such as `BigInt` and `Int8Array`, these are not primitives and lack the optimisation associated with primitives.

Under the hood, JavaScript engines optimise the behaviour of the number primitive and try to use integer arithmetic when possible. The V8 Engine used by Node.js, assigns an *elements kind* to each value or array. There is a whole range of element kinds, but the most important ones are small integers (SMI), double-precision floating-point numbers and regular elements (objects). Each *elements kind* has its own set of optimisations, and the SMI elements will use fast integer arithmetic. But this *elements kind* of values and arrays is not static. Whenever a value changes to something that is no longer representable as the current elements kind, it transitions its elements type. Once an element transitioned to a less efficient kind, it cannot transfer back and will use the less-efficient operations.

To implement efficient hashing algorithms, it is imperative to use integer arithmetic. One goal of a hashing algorithm is to avoid collisions as much as possible. This implicates using as many bits of an integer as possible — in other words: each bit should have an equal probability to be 0 or 1.

Returning to JavaScript's number type, we aim for our hash functions to use as many bits as possible without surpassing the maximum value representable by the significand, which stores the significant digits of the floating-point number as an integer. Exceeding this value would cause the engine to convert the number to a double-precision floating-point format. According to the IEEE standard, 64-bit double-precision floating-point numbers consist of one sign bit, 11 exponent bits and 52 bits dedicated to its significand. Therefore, the maximum representable value as an integer, is $2^{52} - 1$. In order to ensure our hashing function is using integer arithmetic under the hood, we thus need to ensure that a value in our hashing algorithm never exceeds this maximum representable value.

Section 3.2 describes the two polynomial hashing function Dolos leverages: a polynomial rolling hash with an infinite window to hash individual tokens (listing 5.2), and a polynomial rolling hash with a window k to hash k subsequent tokens (listing 5.3). The latter hash function is also used in the Rabin-Karp string search algorithm (Karp and Rabin 1987). Each of these polynomial hash functions have two important constants to pick: the multiplier a and the modulus m . Within Dolos's implementation, we selected these values to ensure numbers remain within the maximum safe integer range:

By using the sign bit, we have $2^{53} - 1$ representable integers. Dolos only uses the positive integers, as one extra bit doesn't justify the added complexity.

```

1  readonly mod: number = 33554393;
2  readonly base: number = 747287;
3  public hashToken(token: string): number {
4      let hash = 0;
5      for (let i = 0; i < token.length; i++) {
6          hash = ((hash + token.charCodeAt(i)) * this.base) %
              ↪ this.mod;
7      }
8      return hash;
9  }

```

Listing 5.2. The code currently used to hash tokens found in the TokenHash-class, implementing a Rabin-Karp rolling hash with an infinite window. The base (multiplier a_T) and mod (m) values are carefully chosen as not to exceed the maximum safe integer number of a double-precision floating-point number to maintain efficient integer arithmetic in JavaScript.

- The modulus (m) determines the largest value of the hashes and thus the numbers used in the hashing function. Both the token hash and the rolling hash use the same constant: the largest prime number with 26 bits. This ensures that multiplying two values within this range will not exceed 52 bits.
- The multiplier (a_T) of the token hash (listing 5.2) aims to distribute the used bits as widely as possible. Each character value is multiplied with this multiplier, and we aim to distribute the used bits as widely as possible within the available range provided by the modulus. Assuming that characters values in token names are less than 128, we chose a_T to be the largest prime such that $127 \cdot a_T < m$.
- The multiplier(a_R) of the rolling hash (listing 5.2) serves a similar function to the token hash, but we can already assume that input values are spread out evenly over the available range dictated by the modulus. This multiplier needs to be different to the multiplier used by the token hash, so we picked the largest prime number with 22 bits.

The rolling hash function also uses modular exponentiation to calculate $-a_R^k \bmod m$ once during its initialisation, so we implemented the algorithm presented by Schneier (1995) that takes care not to overflow the modulus.

```

1  public class RollingHash {
2      readonly mod: number = 33554393;
3      readonly base: number = 419430;
4
5      constructor(
6          public readonly k: number
7      ) {
8          //  $-a_R^k \bmod m$ 
9          this.maxBase = mod - modPow(this.base, this.k, this.mod)
10         ↪ ;
11         this.memory = new Array(this.k).fill(0);
12     }
13
14     public nextHash(token: number): number {
15         this.hash = (
16             this.base * this.hash + \
17             token + \
18             this.maxBase * this.memory[this.i]
19         ) % this.mod;
20         this.memory[this.i] = token;
21         this.i = (this.i + 1) % this.k;
22         return this.hash;
23     }
24 }

```

Listing 5.3. Excerpt from Dolos’s RollingHash-class, implementing a Rabin-Karp rolling hash with a finite window k . This hash function uses the same mod value as the TokenHash implementation from listing 5.2, but a different base that assumes the input hashes are already using all bits. Both values still safeguard that numbers do not to exceed the maximum safe integer number of a double-precision floating-point number to maintain efficient integer arithmetic in JavaScript.

```

1 // Determine the maximum overlap using the spread operator
2 const maximumOverlap1 = Math.max(...pairs.map(s => s.overlap));
3
4 // Determine the maximum overlap using a conventional loop
5 let maximumOverlap2 = 0;
6 for (const pair of pairs) {
7     maximumOverlap2 = Math.max(maximumOverlap2, pair.overlap);
8 }

```

Listing 5.4. Two ways to find the maximum overlap in a list of pairs: the first using the JavaScript spread syntax and the second using a conventional for loop. The spread syntax expands all values as arguments to the function call it is used in, internally pushing the values on the call stack. In a benchmark with 10^5 pairs, the latter is more than twice as fast as the spread syntax. At 10^6 pairs, the spread syntax causes a `RangeError: too many function arguments`.

Avoid the JavaScript spread syntax (. . .)

ECMAScript 6, released in 2015, introduced the spread syntax that allows iterables to be expanded in function calls or array literals. This syntax allows for elegant expressions that are often shorter than using conventional loops, as shown in listing 5.4. Unfortunately, this syntax comes with one major drawback: as it expands the elements in the function calls, it actually pushes them on the call stack. As this stack size is limited, this can result in an error when the number of elements in the iterable exceeds a certain threshold. This syntax can also be slower than other approaches, the execution time of the example shown in listing 5.4 is double that of a conventional loop.

Dolos initially used this spread syntax quite a lot, which resulted in crashes when analysing large datasets with 1000 submissions. By replacing this syntax with other alternatives, the call stack size no longer limited the supported size of datasets. This particular change did, however, not cause a measurable change in runtime of the Dolos similarity detection pipeline.

5.3.2. dolos-lib

Since `dolos-core` is used by `dolos-web` and must operate within browser environments, it cannot have access to platform-specific APIs. Consequently, it lacks functionality to read or parse source code files. `dolos-lib` is a separate library that includes all these capabilities, enabling it to fully execute the source code similarity detection pipeline

```
1 import { Dolos } from "@dodona/dolos-lib";
2
3 const files = [
4   "sample.js",
5   "copied_function.js",
6   "another_copied_function.js",
7   "copy_of_sample.js",
8 ];
9
10 const dolos = new Dolos();
11 const report = await dolos.analysePaths(files);
12
13 for (const pair of report.allPairs()) {
14   for (const fragment of pair.buildFragments()) {
15     const left = fragment.leftSelection;
16     const right = fragment.rightSelection;
17     console.log(`Match between ${left} and ${right}`);
18   }
19 }
```

Listing 5.5. Example of using the Dolos class from the `dolos-lib` package to perform similarity detection on a list of files. The library will automatically detect the programming language of the submissions and select the corresponding parser. The resulting Report object will contain the results of the similarity detection. In this example, the code will print all similar code fragments Dolos has found.

directly files. It depends on `dolos-core` and re-exports all its functionalities. As a result, `dolos-lib` serves as the primary library for developers embedding similarity detection in their applications.

The main purpose of `dolos-lib` is ingesting source code files, run the similarity detection pipeline on them, and then provide the results as JavaScript objects. It does not provide functionality to present these results (either as files, as a website, or printing to the console), as this responsibility belongs to `dolos-cli` (section 5.4). `dolos-lib` itself uses the Node.js runtime and its API to read from the filesystem and run the Tree-sitter parsers provided by `dolos-parsers`.

The `dolos-lib` library offers convenient access to the source code plagiarism detection pipeline through a Dolos class. Power-users and developers wishing to embed the Dolos's capabilities in their scripts and software using the JavaScript API can do so by following the example shown in listing 5.5.

Ingesting datasets

The `dolos-lib` library supports a myriad of different formats in how the collection of source code files can be stored and passed to Dolos for analysis:

- A list of files
- A ZIP archive with a collection of files
- A single comma-separated values (CSV)-file named `info.csv` with at least a column `filename` pointing to the location of the files
- A ZIP archive including an `info.csv` file in addition to a collection of source code files

The last two options that pass the `info.csv` file to Dolos, can optionally provide extra metadata to Dolos. When Dolos detects this metadata, it will include this in the report to enhance the visualisations provided by `dolos-web` (section 5.5). When analysing a ZIP file, Dolos will try to use the `unzip` program present on the system and extract the ZIP archive to a temporary directory to analyse the contents.

Automatic language detection

By looking at the extension of the files submitted for analysis, Dolos is able to detect the programming language automatically. It checks whether the most frequent file extension belongs to a programming language for which it has a parser in `dolos-parsers`. Dolos ignores files that do not have an extension that matches this programming language, and shows a warning indicating that not all files have an extension matching the programming language.

This is a small but convenient quality-of-life feature that allows Dolos to be executed without any extra user input except for a collection of files. In addition, this also allows Dolos to have a convenient developer interface for working around parsers that have a slightly different structure. Some Tree-sitter repositories provide multiple parsers (e.g. `tree-sitter-typescript` provides a parser for `typescript` and `tsx`). Having a `Language` class that knows the correct parser is an ergonomic abstraction that helps developers to find the correct parser. It is still possible to override this behaviour and manually specify which programming language is used in the source code submissions.

Tokenisers

Dolos can support multiple kinds of tokenisers that transform a string of source code into tokens. The most common tokeniser is the `CodeTokenizer` which creates a `tree-sitter` parser to parse the source code. The `Tokenizer` abstract class allows for other kinds of tokenisers.

The `CharTokenizer` is a tokeniser that will make a separate token out of each character in a file. Using this tokeniser, the source code similarity detection pipeline essentially searches for exact matches between the submissions. When submitting files with the extensions `txt` (plain text) or `md` (markdown), Dolos will use this tokeniser by default. One could say that Dolos is able to perform textual plagiarism detection using this tokeniser; however, there are other tools that are far more effective for that purpose.

There has been a draft of a tokeniser for computational notebooks (e.g. Jupyter Notebook¹³). These interactive notebooks are stored as JSON-files with an array of cells which can be either text or code. We developed a prototype that would extract the text and code and run a `CharTokenizer` and `CodeTokenizer` on the corresponding parts, but we discontinued this prototype. As an alternative, it is possible to extract code and text using the `nbconvert`¹⁴ utility.

5.4. Command-line interface

The CLI `dolos-cli` was the first user-facing module that was created for Dolos. It exposes the similarity detection functionalities provided by `dolos-lib`. Dolos can display results from the analysis pipeline in the terminal, export these to CSV-files, or launch a dashboard to inspect the results in the browser (using `dolos-web`).

The CLI provides two commands: `dolos run` executes the similarity detection pipeline and output the results, `dolos serve` displays the results of a previous analysis again. These commands accept command-line options configuring the similarity detection pipeline parameters and output options. Chapter A describes the exact commands and options supported by `dolos-cli`.

¹³: jupyter.org

¹⁴: nbconvert.readthedocs.io

5.4.1. CSV-format

The `dolos-cli` module defines the format of the analysis result files written to the filesystem. The format currently consists of four CSV-files:

metadata.csv The smallest file that contains general information about the analysis: the pipeline parameters used to perform the analysis, warnings (if any), programming language, creation date, and a report title.

files.csv Contains a list of all submitted files used in the analysis with their name, contents, the parsed tokens, and a mapping between each token and its corresponding location in the file.

pairs.csv For each pair of files, it contains the calculated metrics, among them: similarity, longest overlap, and total overlap.

kgrams.csv The (shared) fingerprints occurring in at least two source code files, listing all files that share this fingerprint.

This output format has evolved slightly over time to minimise the report file size. Early versions of Dolos would store the k -grams (fingerprints) in a separate directory with a single file for each k -gram. However, there can easily be thousands of k -grams shared between a collection of submissions. With each of these files taking up at least 4096-bytes¹⁵, this would inflate the disk size used by a report significantly. Combining these files in a single CSV-file significantly reduced the required storage space.

5.4.2. Launching the Web UI

After the analysis has completed, the CLI is able to display the results in an interactive dashboard (section 5.5). Since the Web UI uses the report CSV-files, the CLI will first generate the report and write these CSV-files to the filesystem. Next, the CLI will start a simple web server to host these CSV files and the Web UI. If possible, the CLI will request the operating system (OS) to open a new browser tab pointing to the uniform resource locator (URL) where these files are hosted.

The web server used to rely on Express.js¹⁶ but was replaced by directly launching an HTTP server using the Node.js `http.Server` API. To fully replace the Express server, we added 63 extra lines of code, but

¹⁵The conventional filesystem block size on most operating systems.

¹⁶expressjs.com

this avoids installing 64 extra dependencies. This was one of the more effective attempts reducing the number of dependencies in the Dolos codebase to improve maintainability.

Users who want to inspect a report that was already generated, can use the `dolos serve` subcommand that only launches the web server.

5.5. Web interface

We implemented the visualisations discussed in chapter 4 using the JavaScript frameworks Vue 3¹⁷, Vuetify 3¹⁸, and D3¹⁹. By combining these three frameworks, we built a single-page application (SPA) visualising a similarity analysis report generated by the source code similarity analysis pipeline. This section describes the rationale behind choosing these frameworks and provides specific details about the implementation of the Web UI.

5.5.1. Vue philosophy

Vue, or Vue.js, is a JavaScript framework design for building user interfaces and SPAs. Vue positions itself as *the progressive framework*, as it is designed to scale according to the application and supports deployment across web browsers, desktop applications, mobile applications and other platforms. First released in 2014, it has gained significant popularity due to its simplicity, flexibility, and incremental adoptability.

Vue adheres to a clear philosophy that results in maintainable applications for building user interfaces. Vue follows the model-view-viewmodel (MVVM) design pattern which differs from the model-view-controller (MVC) design pattern. In the MVC design pattern, user interactions are applied to the controller, which manipulates the model and, in turn, influences the view. In the MVVM design pattern, the *viewmodel* acts like a data object, exposing the model data in a way that allows the view to easily access and manipulate it.

To achieve this, Vue provides the following building blocks:

¹⁷v3.vuejs.org

¹⁸v3.vuetifyjs.com

¹⁹d3js.org

Components

Vue 3 encapsulates functionality in single-file components (SFCs) that extend the Hypertext Markup Language (HTML) elements. These SFCs can be compared to classes from the Object-Oriented programming paradigm and are designed to be modular and reusable. Each SFC consist of an HTML template with CSS styling, and JavaScript-code that provides the component's functionality. Some components have input data (called *props*, short for properties) determining how that component behaves. These props can serve as configuration options for that component or as the viewmodel that this component should bind to. Listing 5.6 shows an example of a single file component present in Dolos.

These modular components facilitate their reuse across applications. Consequently, Vue boasts a rich ecosystem of components and libraries that address common challenges in web development. Dolos leverages the Vuetify library of predefined UI components to ensure a consistent user interface. Vuetify provides the building block components to build interactive interfaces, such as buttons, cards, and layouts. The visual design of these components adheres to the Material Design specification²⁰.

Reactivity

Vue offers primitives that support the *reactive programming* paradigm. This paradigm enables programs to respond to changes in data or events, propagating updates throughout the system as needed. Developers declare *reactive values*, also known as *reactive state* or *observables*. When defining these *reactive values*, they can depend on other *reactive values*. This dependency causes the dependent value to automatically subscribe to the original value, reacting to any changes. Instead of explicitly specifying how data changes, developers define how data flows through their system in a declarative manner. This paradigm is well-suited for applications that process events and real-time data, such as user interfaces. These reactive values often serve as the *viewmodel* in the MVVM design pattern.

Stores

In small Vue applications, it is possible use only reactive component props to propagate the model information throughout the applica-

²⁰m3.material.io

tion. However, once the application and its global state become larger and more complex, this approach becomes more challenging. Vue addresses this by introducing the concept of *stores* through a neatly integrated library Pinia²¹. A store acts as a pool of reactive global state, accessible from anywhere in an application using the singleton design pattern. Typically, the *model* from the MVVM pattern is present in or accessed by the stores of a Vue application.

5.5.2. Report ingestion and initialisation

dolos-web visualises similarity detection reports generated by Dolos's source code similarity detection pipeline. It ingests the report data created by the CLI by reading CSV-files as described in section 5.4.1. When a user opens the Web UI, Vue will show a loading animation and begins the initialisation of the central API store (`api.store.ts`). During this phase, the API store initialises the metadata store, the file store, the k -gram store, and pairs store. Each of these stores fetches their corresponding CSV-file and parses it using the Papaparse²² library. Once the report data is loaded in the store, it calculates the similarity threshold using the algorithm described in section 4.3.2. After completion, the loading animation is replaced by the proper UI displaying the report results, starting with the overview page (section 4.3).

5.5.3. D3 Visualisations

The Dolos Web UI offers insight in the source code similarity report by presenting clear visualisations. We constructed these visualisations using the D3²³ JavaScript visualisation library (Bostock et al. 2011). D3 is a low-level toolbox that provides the building blocks for creating interactive data-driven visualisations.

The visualisations that use D3 include:

- The Similarity Histogram (section 4.3.1)
- The Plagiarism Graph (section 4.4)
- The Cluster Timeline (section 4.5.1)
- The Cluster Heatmap (section 4.5.1)

²¹pinia.vuejs.org

²²papaparse.com

²³d3js.org

```

1  <script lang="ts" setup>
2  import { storeToRefs } from "pinia";
3  import { useApiStore } from "@/api/stores";
4
5  interface Props { compact?: boolean; }
6
7  const props = withDefaults(defineProps<Props>(), {});
8  const { cutoff, cutoffDefault } = storeToRefs(useApiStore());
9
10 const resetCutoff = (): void => {
11   cutoff.value = cutoffDefault.value;
12 };
13 </script>
14
15 <template>
16   <div class="similarity-setting">
17     <label v-if="!compact" class="text-medium-emphasis">
18       Threshold ≥ {{ (cutoff * 100).toFixed(0) }}%
19     </label>
20     <div class="similarity-setting-actions">
21       <span v-if="props.compact">
22         {{ (cutoff * 100).toFixed(0) }}%
23       </span>
24       <v-slider v-model.number="cutoff" min="0.25" max="1" />
25       <v-btn icon="mdi-restore" @click="resetCutoff" />
26     </div>
27   </div>
28 </template>
29
30 <style scoped>
31   .similarity-setting label {
32     font-size: 0.9rem;
33     font-weight: normal;
34   }
35
36   .similarity-setting-actions {
37     display: flex;
38     justify-content: space-between;
39     align-items: center;
40   }
41 </style>

```

Listing 5.6. The Vue 3 component `SimilaritySettings.vue` from the `dolos-web` package, demonstrating how Vue 3 works. This component provides a single slider to adjust the similarity cutoff value globally used throughout the application. Since the cutoff value is part of the global state, it is saved in a *store*. This store provides the reactive value `cutoff`. When this cutoff in the store changes, the reactivity will automatically re-render the slider and the similarity percentage shown. Updates in the other direction are also possible: if the user changes the slider value, this change will propagate to the store and update all other components using this value. By using the `v-slider` component provided by the `Vuetify` library, this component will have a style consistent with the rest of the application, coherent with the Material Design specification.

Except for the plagiarism graph, these visualisations use D3 to render an interactive Scalable Vector Graphics (SVG) image. The plagiarism graph uses the HTML Canvas API to render the force-directed graph efficiently, because of the sheer amount of nodes and edges that need to be rendered.

Plagiarism graph

The plagiarism graph, with its design detailed in section 4.4, is the most complex visualisation in Dolos and has undergone several rewrites and optimisations to maintain performance and manageability. Initial implementations of the plagiarism graph used D3's SVG API, resulting in an SVG element for each node and connecting edge. As the threshold decreases, the number of edges increases, leading to a slow and unresponsive page as the browser's HTML parser and renderer struggle to handle the sheer number of elements updating every frame.

To address this performance issue, we refactored the plagiarism graph implementation to leverage the HTML Canvas API. This API allows direct drawing of basic shapes on a fixed-size canvas, bypassing the HTML parsing process and significantly improving performance. The result is a much snappier plagiarism graph capable of rendering large numbers of nodes and edges efficiently.

For the *π -ramidal constants* dataset as mandatory exercise²⁴, the previous implementation would freeze the page for 3 seconds with a similarity threshold of 50% (1 398 edges) and for 55 seconds with a 25% threshold (13 528 edges). When this page loaded, the plagiarism graph was unusably unresponsive. The current implementation using the canvas API remains responsive even at a 25% similarity threshold. Even when loading the *Pluto killer* dataset, which comprises 1162 nodes and 674 541 edges, the plagiarism graph does not freeze before rendering the graph with a 25% similarity threshold. However, the page does begin to struggle as evidenced by a noticeable drop in frame rate.

The current implementation also enhances code quality. Previously the plagiarism graph implementation was concentrated in a single class of 570 lines of code. The refactored version features a clearer structure and is more aligned with Vue's philosophy. The source file serves `useD3ForceGraph.ts` as the entry point for creating the force-directed graph simulation. This source file exports the `useD3ForceGraph composable` function, which encapsulates clearly defined stateful logic for rendering the plagiarism graph. Instead of directly accepting a

²⁴dolos.ugent.be/demo/pyramidal-constants/exercise/

reactive list of nodes and edges, it offers functions to ergonomically accept or update them. This composable delegates most functionality to four dedicated source files:

- `data.ts`: Stores the nodes (submissions), edges (pairs) and groups (clusters).
- `interaction.ts`: Handles user interactions, such as showing tooltips, selecting, and dragging nodes and clusters.
- `simulation.ts`: Calculates new node positions according to interacting forces, and recalculates the convex hull around clusters.
- `rendering.ts`: Draws the nodes, edges and groups on the HTML canvas.

The result is a modular, maintainable and efficient plagiarism graph component, which stands as one the most intuitive visualisations in Dolos.

5.5.4. The Monaco editor

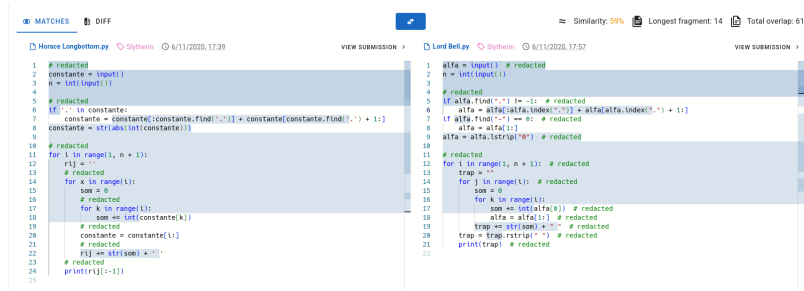
The comparison page (section 4.6) relies on the Monaco editor²⁵ to render a pair of source code files. This editor was initially developed for the highly popular IDE VS Code²⁶. The Monaco editor is feature-rich and provides extensive customisation options.

The comparison page has two modes, indicated by the top left tabs: matches and diff. The diff tab (shown in figure 5.2b) uses Monaco's built-in diffing engine to highlight the differences between two files. The Dolos Web UI automatically selects this tab when the similarity exceeds 80%.

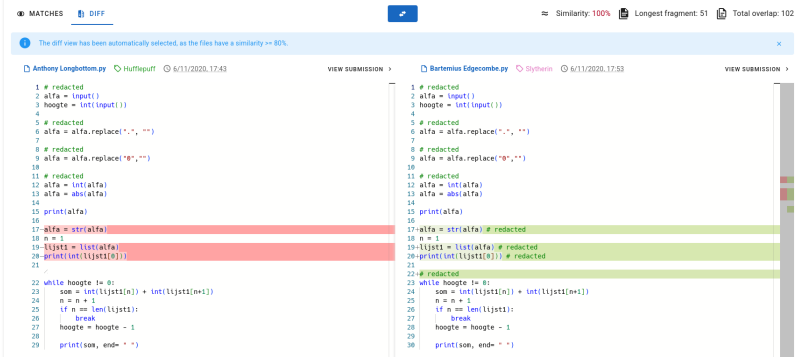
The matches tab (shown in figure 5.2a) highlights similar code fragments discovered by the pipeline. Storing all matching information in the csv-files would use a lot of disk space, so Dolos instead calculates these fragments lazily. When Dolos requires the matching fragments of a pair, it runs the source code similarity detection algorithm again using the `dolos-core` library. All information to re-calculate this data is present in the csv-files. Once we have constructed the small fingerprint index for these two files, we apply the method described in section 3.3.3 to compute the matching code fragments.

²⁵microsoft.github.io/monaco-editor

²⁶code.visualstudio.com



(a) Pairwise matches



(b) Pairwise diff

Figure 5.2. Two pairwise editor modes

This process is quite fast, as it must analyse only two already tokenised submissions. However, we cannot run this on the main thread in the browser, as it would noticeably freeze the web page. We use a web worker to perform this computation asynchronously in the background, and briefly display a loading animation.

Using the locations of these fragments, we use Monaco's decorations API to highlight the areas present in a fragment. When selecting a highlighted area in the editor, we give the corresponding decorations a different to indicate these matching fragments in both submissions.

5.5.5. Server mode

The `dolos-web` package, by default, compiles in *normal mode*. In this configuration, Dolos Web UI visualises a single report, expecting its associated data files to be situated in a predetermined location. This

mode is intended for use in conjunction with the CLI, facilitating a seamless interaction between the two components.

Since introducing the web server (section 5.6), we incorporated a dedicated *server mode* into the `dolos-web` package. This mode modifies UI to communicate with the `dolos-api` API server and introduces capabilities to upload, manage, and share reports. The build tool, Vite²⁷, determines the current mode by detecting the presence of environment variable `VITE_MODE=server`. When in server mode, Vite uses the `VITE_API_URL` variable to locate the Dolos API server.

Combining the components for creating a new analysis and inspecting the analysis report, results in a seamless interface between these two aspects. This design enables reusing components from the normal mode and allows users to transition effortlessly between reports.

Server mode introduces a new upload page (illustrated in figure 5.3), enabling users to submit new datasets for analysis. This page features an upload form where users can submit a ZIP archive containing the submissions they wish to analyse. The Dolos Web UI keeps track of previously analysed reports and displays a list from which users can view, share, or delete these reports.

We opted not to implement user management and authentication to Dolos, as this would introduce unnecessary complexity not necessarily for our use case. Instead, upon creating a new report, the Dolos API generates a URL with a secret identifier that grants access to that report. The Dolos Web UI stores this unique URL in the browser's LocalStorage and ensures this URL remains hidden from the web page's address bar. The UI facilitates sharing a report via this secret URL, and visiting a shared report adds it to the list of known reports.

5.6. API server

The Dolos API server provides access to Dolos's similarity detection pipeline through a JSON API. This API integrates with the Dolos Web UI, built in *server mode* (section 5.5.5), to form the Dolos web server.

We have simplified the installation process for Dolos CLI to minimise potential obstacles. Nevertheless, this task can still be challenging for instructors, especially those less adept with technology. The Dolos Web server addresses this issue by enabling instructors to perform source code similarity detection within their browsers.

Even those adept with technology struggle with installing software from time to time.

²⁷vitejs.dev

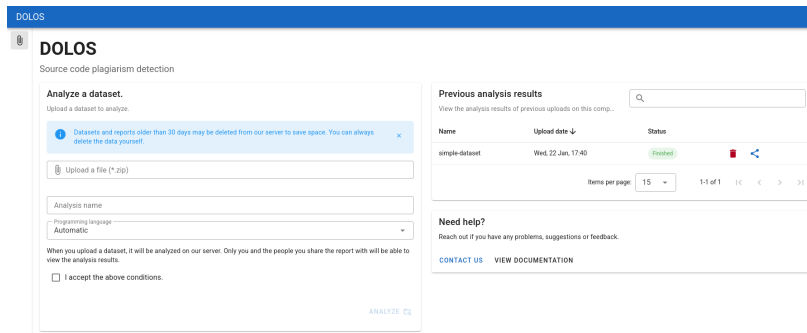


Figure 5.3. Upload page shown when the Dolos Web UI is built in *server mode*, acting as the launchpad of the Dolos web application. The left card shows the upload form where users can submit a new collection of source code files to analyse. The right card shows a searchable table for accessing, deleting and sharing previously submitted reports.

Currently, the API is freely accessible without authentication and no rate-limiting, allowing developers to integrate Dolos as a microservice within other web applications. Notable examples of such external integrations are presented in section 5.6.2.

The Dolos API server is the sole component within the Dolos ecosystem implemented using an alternative programming language. It is driven by Ruby on Rails²⁸ (short: Rails), a full-stack and full-featured framework for the Ruby programming language. The API server harnesses Docker to execute analysis tasks and uses the MariaDB relational database server to persist report data.

5.6.1. API submission flow

Using the Dolos API server is quite straightforward. Creating a new source code similarity analysis goes through a few steps, visualised in figure 5.4.

First, the API consumer (for example, the Dolos Web UI in *server mode*) submits a new dataset (a ZIP archive with source code files) using a HTTP POST request. The API server stores this dataset and links a new Report record to it in the database with a unique, secret identifier. The server enqueues a new analysis in the job queue for this report and sets the report status to queued. The API responds with a JSON object including the Report secret URL.

²⁸rubyonrails.org

In the background, worker processes process the job queue using the `delayed_job`²⁹ asynchronous queue system. Once a worker picks up a pending report from the job queue, it updates its status to `running`. The background worker uses the same system used by Dodona to run the analysis jobs in an isolated, sandboxed environment: a job will run the Dolos CLI using the `dolos-cli` Docker container (section 5.7.3). Once the analysis is complete, the worker collects the resulting CSV-files and stores them with the Report record. Finally, the worker updates the report status to `finished`. If the analysis failed or an error occurred when trying to run the analysis, the report status will be updated to `failed` or `error`.

The code running analysis jobs is in fact a copy-paste of the Dodona code.

Meanwhile, the API consumer can fetch the report status by performing a HTTP GET request on the secret report URL to look up its status. Once the status of the report is `finished`, the CSV-files are available for download. The consumer can fetch each CSV-file using this secret report URL and the file name.

Optionally, the API consumer can send a HTTP DELETE request to the report URL to remove the report. This will remove the stored dataset with the source code files, and the resulting CSV-files. The API will update the report status to `deleted` to indicate this change. The web server can also periodically remove reports in the same manner according to its data retention policy.

5.6.2. External integrations

The Dolos API facilitates the integration of Dolos as a microservice within other platforms. To integrate Dolos with an application, the following two steps are needed:

- Initiate a new analysis by submitting a ZIP archive containing the desired source code files to Dolos.
- Direct the user to the Dolos report URL received in response.

A microservice is an independent and specialised service communicating through a lightweight protocol.

The HTTP POST request, as detailed section 5.6.1, contains all necessary information for Dolos to generate the similarity report. Naturally, users must specify which submissions they wish to analyse. The application then compiles these submissions into a ZIP archive before dispatching them to Dolos. Gathering the source code of the submissions and creating the ZIP archive can be resource-intensive and

²⁹github.com/collectiveidea/delayed_job

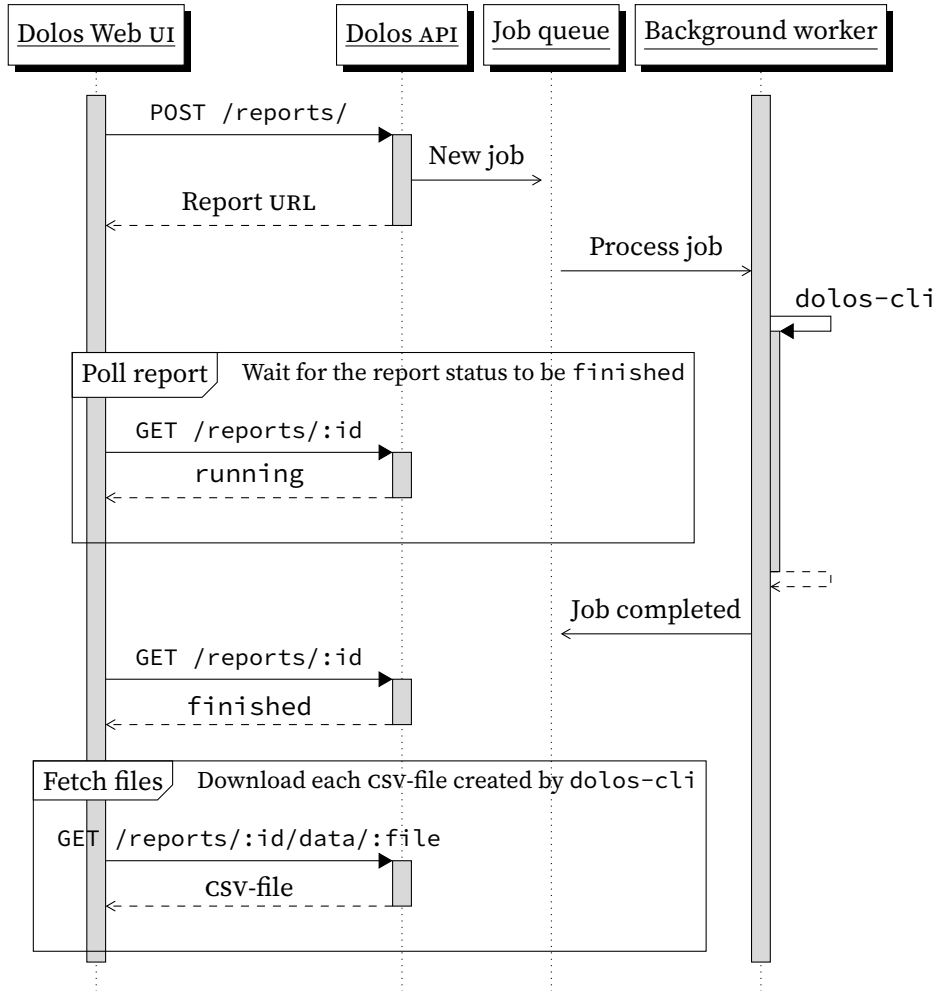


Figure 5.4. Sequence diagram of the analysis flow through the Dolos API server.

time-consuming task, particularly when numerous submissions are involved.

Upon submitting the analysis and redirecting the user to the report URL, the report may not be immediately available. To avoid the external application repeatedly polling the report until the analysis is completed, we enable applications to promptly redirect users to Dolos. The Dolos Web UI, operating in *server mode*, will initially request the report status. If the Dolos API server is still processing the report, a loading page will display until the analysis concludes.

Dodona

Dolos originated and evolved within the team responsible for building Dodona, the programming exercise platform at Ghent University. Integration has always been a key objective for Dolos since its inception, and this integration has progressively advanced during its development. Initially, Dolos could interpret the `info.csv`-file included in Dodona's submission exports. Subsequently, Dolos CLI gained the capability to directly utilise the ZIP archive as input, eliminating the need for users to extract its contents manually. The Dolos web server further streamlined this process by enabling users to upload the ZIP archive immediately after downloading it from Dodona. These efforts culminated with the merging of a pull request on 13 May 2024, which achieved the direct integration of Dolos within Dodona.

This integration introduces a *Detect plagiarism* button on the Dodona pages that list all submissions for an exercise. Activating this button initiates the similarity detection process, as illustrated in figure 5.5, ultimately directing the user to the Dolos similarity report. Following the initial integration, instructors can also trigger the similarity detection process from within the evaluations page, nudging them to perform a quick plagiarism check before commencing the grading process.

A+ and Radar

A+ is a learning management system (LMS) initially launched by the Learning+Technology research group at Aalto University in Helsinki, Finland (Karavirta et al. 2013). CS-IT, the IT team at the Department of Computer Science at Aalto University, is currently leading this project. A+ LMS focuses on integrating multiple online learning services by leveraging protocols that dictate how these services should work together.

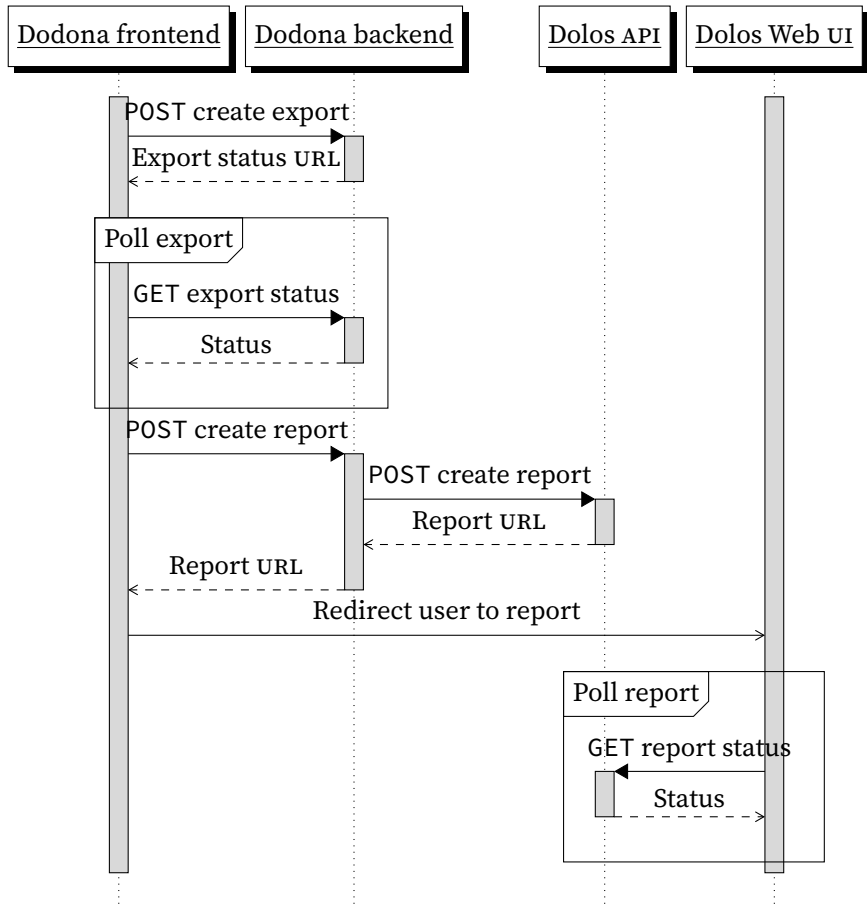


Figure 5.5. Sequence diagram of the flow of running a similarity detection from within the Dodona platform. The user initialises the flow by clicking a *Detect plagiarism* button on a submissions page in Dodona. The Dodona frontend (JavaScript) creates an ZIP archive export including the desired submissions and polls the status of this export. Once the export is ready, the frontend confirms the report creation, after which the Dodona backend will upload the report to the Dolos API, queueing a new report. The Dodona frontend will then redirect the user to the Dolos report URL, where the Dolos Web UI will show a loading screen until the report is ready.

One of these services is Radar³⁰, a web service for analysing source code similarity created in 2015. Radar implements the Running Karp-Rabin Greedy-String Tiling (RKR-GST) matching algorithm to search for similarities between submission source code. Using the Learning Tools Interoperability (LTI) protocol, Radar fetches course, exercise, and submission information to perform its analysis.

The CS-IT team at Aalto University contacted us in April 2024 mentioning they wished to update Radar, a tool they have been using since 2015, with something newer. Dolos fit their purpose, and during a close collaboration in September 2024, the CS-IT team integrated Radar and Dolos in a single service.

This combined service was colloquially dubbed *Rhodos*.

An important part of the original Radar service was managing the similarity detection results within the courses and exercises, while also ensuring only authorised users can see the results. Dolos does not implement course management, nor does it implement user authentication or authorisation. The CS-IT team then integrated Dolos within Radar, similar to the approach of Dodona (section 5.6.2): Radar is responsible for collecting the submission source code files, preparing a ZIP archive, and then sending the archive to the Dolos API.

The Radar integration does not use the publicly available Dolos API. The CS-IT team host their own instance of the API using the Docker compose configuration provided by Dolos (section 5.7.3). This ensures that student submissions never leave their university's servers to perform the similarity detection.

5.7. Additional components

Some components in the Dolos repository are not directly part of the Dolos source code similarity detection pipeline. However, these components are essential for the development and adoption of Dolos.

5.7.1. Documentation

Documenting software is a crucial practice in software development. Writing good software is of little value if knowledge about its inner workings is not preserved. Dolos provides two types of documentation: internal documentation, which describes the workings of each module, and external documentation, which explains how to use Dolos.

³⁰github.com/apluslms/radar

Internal documentation

The internal documentation outlines the current state of the code and architecture underpinning Dolos. Each primary module includes a `README.md` file that provides a high-level overview. This document serves as a starting point to developers interested in contributing to that module. GitHub and NPM prominently display the contents of this README file on the main repository or package page, helping external developers understand the module's main role and determine whether Dolos meets their needs. Additionally, comments interspersed throughout the code offer more detailed insight into the codebase's inner workings.

The Git history, GitHub pull requests, and issues provide a record of all major changes and fixes applied throughout the development of Dolos. When investigating or remediating a bug or lacking feature, it is very helpful to re-read the initial reasoning that went behind the original code. The release notes published with each new release of Dolos contain a short summary about these changes in each release.

External documentation

The module `dolos-docs` contains the external documentation aimed towards users and developers. Presented in the form of a public-facing website hosted at dolos.ugent.be, this documentation aims to explain how to use Dolos in its various facets: the software libraries, the CLI, the web server, the API, ... Because this website is often the first point of contact with users searching for similarity detection tools that fit their purpose, the website highlights the features and functionality of Dolos.

The documentation website is a static website using the Vitepress³¹ static site generator. Vitepress transforms pages from the Markdown format to HTML. This allows developers to focus on writing the plain text documentation, while Vitepress takes care of properly rendering it in a modern website interface. Vitepress uses Vue components and allows extending the documentation with custom components. We leverage this functionality on the page with the API documentation³² to allow visitors to modify the URL of the Dolos API endpoint shown on that page.

The documentation website currently provides the following pages:

³¹vitepress.dev

³²dolos.ugent.be/docs/api.html

- **Introduction:** What is Dolos? Who made it?
- **Use Dolos:** Explaining how to use the Dolos web server.
- **Use case: Dodona:** Showing instructors how to use Dolos to analyse Dodona submissions.
- **Self-host Dolos:** Instructing developers how to set up their own Dolos web server
- **Dolos API:** Listing the API endpoints and demonstrating it with examples.
- **Install Dolos CLI:** Instructing how to install the Dolos CLI.
- **Use Dolos CLI:** Demonstrating how to use the Dolos CLI.
- **Run Dolos CLI using Docker:** Explaining how to run Dolos CLI using the `dolos-cli` Docker container (section 5.7.3)
- **Add metadata:** Describing the `info.csv` format to enhance a dataset with submission metadata.
- **Add new languages:** Listing methods to add support for new programming languages.
- **Use the Dolos library:** Providing an example on how to use the `dolos-lib` software library.
- **How Dolos works:** Describing the Dolos sourcecode similarity detection pipeline algorithms.
- **Supported programming languages:** Explaining how Dolos supports programming languages.
- **Research publications:** Listing publications made by Team Dodona.
- **Contact us:** Providing our contact details.

5.7.2. Samples

The `samples` folder in the root of the Dolos repository contains small source code snippets for programming languages supported by Dolos. These samples are used in tests to validate whether the tokenisers function as expected and produce stable results. We sourced these samples from the Caesar cipher page on the Rosetta Code website³³. Rosetta Code is a wiki that provides code snippets implementing algorithms in as many programming languages as possible. The wiki

³³rosettacode.org/wiki/Caesar_cipher

```
docker run --init --network host -v "$PWD:/dolos"
↳ ghcr.io/dodona-edu/dolos-cli -f web *.js
```

Listing 5.7. Shell command to run the Dolos CLI using its Docker image. A user running this command only needs to have Docker installed for Dolos to work. The `--network host` option is needed for the host system to access the Dolos Web UI in the container. The `-v` option mounts the current directory as the working directory for the Dolos CLI present in the container.

offers implementations in over 150 different programming languages. Most implementations are concise but demonstrate common language features.

5.7.3. Containers

Managing program dependencies can be quite challenging, especially when these components use different programming languages and requires specific versions of a software development kit (SDK). Docker is a system to bundle programs and their dependencies in *containers*. These containers run using lightweight OS-level virtualisation, sharing the host system's kernel. Except for predefined channels like network and file system access, the virtualised containers operate in isolation from each other and the host system. This makes Docker a secure and reliable solution to distribute packages.

`dolos-cli`

The `dolos-cli` Docker image, hosted on the full URL ghcr.io/dodona-edu/dolos-cli, packages the CLI. Listing 5.7 shows the simple oneliner that runs the CLI using this image. This image builds further on the `node:alpine` image: a container image packaging the lightweight Alpine Linux distribution³⁴.

Dolos Docker Compose

Docker Compose is a configuration format and tool to combine multiple containers together in one system. By clearly defining how each container should connect to the network and file storage, it can be used to create complex services. Dolos provides a compose configuration

³⁴alpinelinux.org

to host the Dolos web server, combining the API and Web UI, on your own system.

The Docker Compose configuration for the Dolos Web server publishes two extra container images on GHCR: `dolos-api` bundling the API server, and `dolos-web` bundling the Web UI. The Dolos API image is used in two container services: once as the API server itself and once as a worker processing the analysis jobs. Changing the environment variables in the `docker-compose.yml` file in the repository configures the services in the containers. Because the containers combine to form the Dolos web server, they are not able to run standalone.

Currently, we recommend using our Docker Compose configuration for self-hosting the Dolos web server. This is the most straightforward approach to achieve a secure and working setup. However, it is also possible to run these services outside of Docker directly on the host system. The public Dolos web server uses the latter approach.

5.7.4. Nix flake

Similar to Docker, Nix is a build tool and package management system to reliably run software across diverse systems. Nix, often used within NixOS³⁵, uses declarative configuration files written in the Nix programming language. Unlike Docker, Nix enforces stricter reproducibility by precisely specifying external sources. A Dockerfile, defining the build steps of containers, allows interactions with non-deterministic sources, such as the internet. This can lead to Docker containers producing different results when built at different times, potentially causing unexpected consequences.

In contrast, Nix packages, often referred to as *derivations*, require their inputs to be deterministic. For packages that need internet access (e.g. to download a *tarball* with source code), the accessed URL is stored along with the cryptographic hash of the expected result. When the package is downloaded again, Nix verifies that the hash matches to ensure the inputs are consistent before continuing the build.

We provide a *Nix flake* providing the following *derivations* that can help when developing or using Dolos:

- A **devshell**, a shell environment with all required SDKs set-up.
- The **dolos-cli** package, providing the Dolos CLI currently present in the repository.

³⁵nixos.org

```
nix run
↳ 'git+https://github.com/dodona-edu/dolos.git?submodules=1'
↳ -- run -f web *.js
```

Listing 5.8. Shell command to run the Dolos CLI using the Nix flake. A user running this command only needs to have Nix installed for the above command to work. This command will download the latest version of Dolos and build it locally, so the first execution can take a while. Subsequent executions will use the cached derivation. Compared to the docker command, Dolos executes on the host system and the users' browser can directly access the spawned web server with the UI.

- A **check** to validate whether this `dolos-cli` package can run successfully.

Similar to Docker, this configuration provides a low-dependency approach to running Dolos, illustrated by listing 5.8. Instead of downloading a sizeable Docker container image, Nix fetches the repository source code and builds the package locally.

Especially the *devshell* configurations can help to provide a known-good development environment. The main reason to make these configurations, was for our personal development environments. Since these configuration can be a useful alternative to Docker, we added them to the repository.

Chapter 6.

Evaluation

To evaluate whether we succeeded in our goal of creating a user-friendly qualitative tool for source code similarity detection, we have assessed Dolos through four methods:

- **Usage metrics:** We examine the adoption of Dolos by analysing usage metrics (section 6.1).
- **Quality benchmarks:** We assess the quality of the similarity detection results by performing benchmarks (section 6.2).
- **User Experience Survey:** We evaluate the user-friendliness of Dolos using a user experience questionnaire (UEQ) (section 6.3.1).
- **Case study:** We discuss how Dolos is utilised in practice to prevent and detect plagiarism (section 6.4).

6.1. Usage metrics

An initial indicator of Dolos’s effectiveness, is its frequency of use. Since its first deployment in december 2022, the public instance of Dolos¹ has processed 22 516 reports. The number of submitted analyses has been steadily increasing since this public release, as shown in figure 6.1. To preserve the privacy of its users, Dolos does not track user information. However, analysing the web server logs of 10 days between April 18–28 2025 reveals Internet Protocol (IP) addresses associated with 66 different countries, indicating that Dolos enjoys global adoption. Figure 6.2 illustrates the number of submitted analyses, grouped by country, including requests to the Dolos Application Programming Interface (API).

¹dolos.ugent.be/server

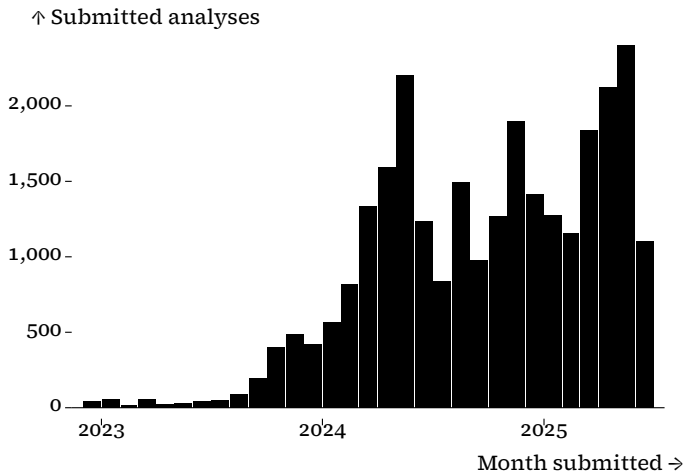


Figure 6.1. Bar chart illustrating the evolution of number of submitted analyses per month for the public instance of the Dolos web server. Since the public release at the end of 2022, this number has been steadily increasing, indicating that more and more teachers are adopting Dolos.

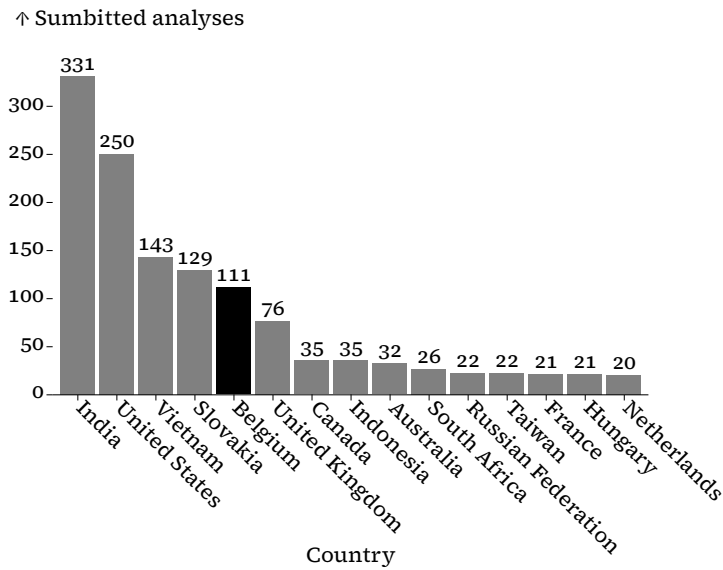


Figure 6.2. Bar chart indicating top 15 countries from which analyses were submitted to the Dolos web server in the 10 days between April 18–28 2025. Belgium, the country where Dolos is developed, is highlighted in black. Note that these numbers might be skewed by temporary fluctuations because of this small time window.

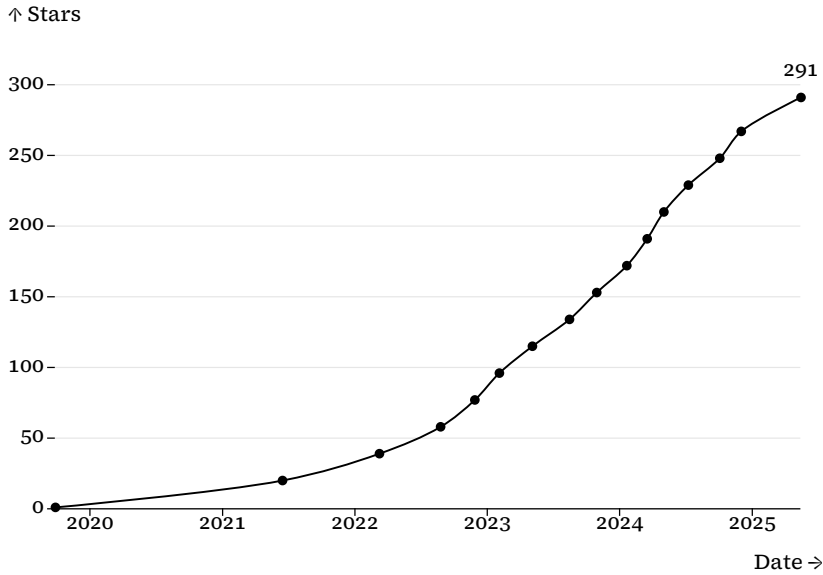


Figure 6.3. Evolution of the number of stars of the Dolos repository on GitHub on May 15th 2025.

These figures apply only to our public instance, and do not account for usage by institutions that have set up their own instance to comply with privacy regulations. We are aware of at least two additional instances: one in Finland and one in the Netherlands. Other institutions hosting their own instance will mostly do so using our containerised images (section 5.7.3), for which we find that the `dolos-api` container has been downloaded a total of 634 times from GitHub Container Registry (GHCR).

Another measure of Dolos's adoption is the number of times the command-line interface (CLI) has been downloaded. The NPM package containing the Dolos CLI has been downloaded 11 655 times since its release in 2020. The container including the Dolos CLI has been downloaded a total of 7 150 times.

These numbers are very likely inflated by automated downloads.

The GitHub repository hosting the Dolos's source code boasts 291 stars, a method for developers to express appreciation and stay updated with new releases (figure 6.3). Our repository also has 42 forks, 11 of which have at least one commit.

6.2. Benchmarks

This section is based on the validation section of our journal article “Dolos: Language-agnostic plagiarism detection in source code” (Maertens, Van Petegem, Strijbol, Baeyens, Jacobs et al. 2022). My contributions to this section in the original article include designing the methodology, collecting data, implementing the benchmark software, performing the benchmark, visualising the results and writing the original draft. While the methodological approach remains largely consistent, we have expanded the scope by incorporating an additional tool (Compare50) and additional datasets evaluating alternative facets of the benchmark. The original benchmark was repeated with the most recent versions of the compared tools, replicating the original results. I have adapted the original text to harmonise with the style and structure of this dissertation.

We conducted a benchmark to assess Dolos’s performance relative to similar tools. The benchmark quantitatively evaluates Dolos’s predictive power for plagiarism detection and compares it against four state-of-the-art tools: Moss, Sherlock Sydney, JPlag, Plaggie and Compare50. For JPlag, we consider both their legacy version JPlag 2, and their latest release JPlag 6.

All tools under investigation compute a similarity value for each pair of source files, but use different similarity measures. Direct comparison of similarity values is therefore not relevant. Instead, we evaluate how well the similarity measure of a tool can separate plagiarised from non-plagiarised code using the optimal similarity threshold for that tool, as a surrogate for its predictive power to detect plagiarism. We consider similarity values above the threshold as positive predictions for plagiarism, and similarity values below the threshold as negative predictions. A ground truth, or an approximation using expert annotations, allow us to determine whether these predictions are true or false. This allows us to determine the recall, precision, and F_1 -measure in order to evaluate each tool’s effectiveness.

6.2.1. Datasets

In the domain of educational source code similarity detection, most tools are validated using either private datasets comprising real student submissions or generated datasets where an original program undergoes a series of modifications to simulate plagiarism (Novak et al. 2019). These datasets are rarely released into the public domain due to the personal information they contain, thereby complicating the

replication and interpretation of results and hindering the evaluation of different tools against one another.

SOCO benchmark

We used the publicly available SOCO benchmark dataset containing 79 C files and 259 Java files (Arwin and Tahaghoghi 2006; Flores et al. 2014). This benchmark dataset features expert annotations of file pairs deemed to be instances of plagiarism. The annotations for the C corpus identify 26 of the 3081 C file pairs (0.84%) as plagiarised, with 37 C files (46.84%) occurring in at least one plagiarism pair. The Java corpus labels 84 out of 33 411 Java file pairs (0.25%) as plagiarised, with 115 Java files (41.97%) involved in at least one plagiarism pair.

Private dataset

We have compiled a dataset containing eight confirmed cases of source code plagiarism between 2020 and 2023, where plagiarism occurred during summative assessments such as exams and evaluations. Each case consists of a corpus of anonymised submissions in the Python programming language and includes exactly one plagiarised pair of submissions. The students involved in each pair confirmed their plagiarism during internal hearings. This dataset serves the purpose of assessing how each similarity detection tool performs on actual cases of plagiarism, in contrast to a synthetic dataset.

Plutokiller

To measure resource usage and performance of each similarity detection tool, we additionally ran our benchmarks on an internal dataset derived from the Plutokiller² exercise on Dodona. This dataset contains 1162 submissions, 80 lines of code on average, submitted to this exercise in the Python programming language. While this dataset likely contains a large amount of plagiarism, the absence of a ground truth indicating which pairs are plagiarised precludes its use for determining predictive accuracy. However, the substantial number of files and the resulting 674 541 pairs make it an excellent stress test for measuring the run time and memory usage of a similarity detection analysis. Using an automatically selected similarity threshold of 86%, Dolos identifies 88 clusters with the largest cluster containing 42 submissions. The resulting plagiarism graph is shown in figure 6.4.

²dodona.be/en/activities/82601015/

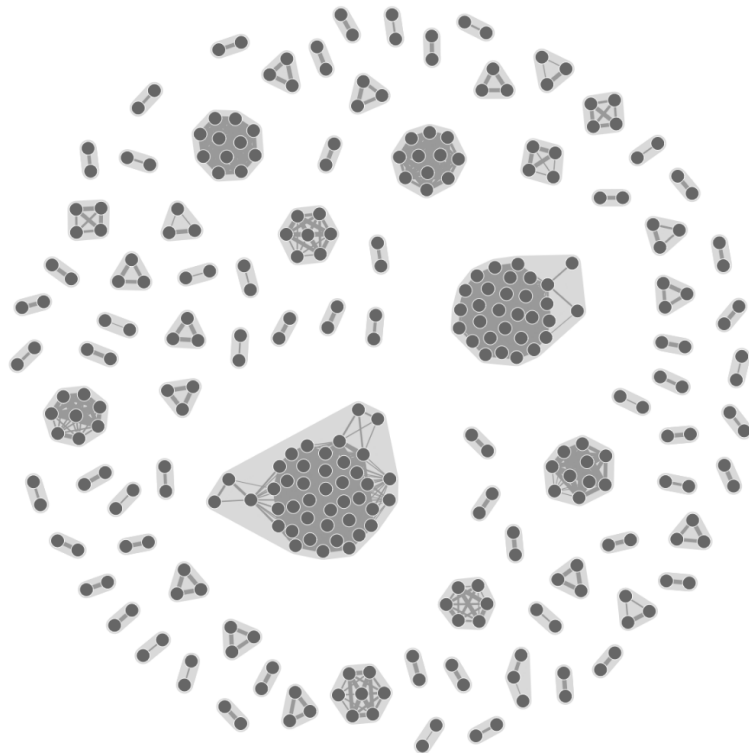


Figure 6.4. Plagiarism graph of the Plutokiller dataset showing all clustered submissions using the automatically selected similarity threshold of 86%. Submissions with all pairwise similarities below this threshold are not included in a cluster and therefore shown in this figure. This dataset includes 1 162 submissions, resulting in 674 541 pairs. Using the similarity threshold, Dolos finds 88 clusters with the largest cluster containing 42 submissions.

6.2.2. Method

We employ the F_1 -score as a global measure of predictive accuracy for similarity detection tools. The F_1 -score is calculated as the harmonic mean of precision and recall, using the formula $F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. Since the F_1 -score is dependent on a similarity threshold, we determine the threshold that maximises the F_1 -score, representing the optimal prediction for a given similarity analysis. The threshold corresponding to the maximum F_1 -score concurrently minimises the number of false positives and false negatives. This approach mirrors how educators use plagiarism detection tools: they sort file pairs by descending similarity and review pairs in that order until the sequence of non-plagiarised pairs is sufficiently long to anticipate no further instances of plagiarism.

Each tool under investigation possesses parameters that influence its computation of similarity values. Rather than merely calculating the maximum F_1 -score for a single configuration of parameter settings, such as the default settings of a tool, we repeat this process across a range of configurations. This parameter sweep provides additional insights into the impact of parameter settings on prediction accuracy, reveals differences between programming languages, and assesses the suitability of default settings. It also prevents the inadvertent selection of suboptimal configurations or the cherry-picking of favorable results for individual tools. The configurations evaluated for each tool are as follows:

- Dolos: k -gram lengths 10, 12, 15, 17, 20, 23 and 25 (option `-k`, default: 23), and window sizes 10, 12, 15, 17, 20, 25, 30, 35 and 40 (option `-w`, default: 17); 63 configurations in total
- Moss: all integers in the range [1, 20] as the maximum number of files in which a code fragment may appear before it is ignored (option `-m`, default: 10); 20 configurations in total
- Sherlock Sydney: all integers in the range [1, 5] as the number of zero bits (option `-z`, default: 3) and the chain length (option `-n`, default: 4); 25 configurations in total
- JPlag 2 and JPlag 6: all integers in the range [5, 20] as the minimum number of matching tokens (option `-t`, default: 9 for Java and 12 for C); smaller values increase sensitivity; 16 configurations in total

- Plaggie: all integers in the range [2, 22] as the minimum number of matching tokens (option `-m`, default: 11); equivalent to the option `-t` of JPlag; 21 configurations in total
- Compare50: as this tool does not provide algorithmic parameters, only one configuration was included in the benchmark

Our benchmark code additionally measures the run time and memory usage using the Unix `getrusage(2)`³ functionality. The benchmark process spawns each tool as a child process and records the run time as the sum of the user and system CPU time, while the memory usage is recorded as the maximum resident set size. We executed these benchmarks on a Unix system running NixOS with an 11th generation Intel i7-1165G7 CPU.

Due to Moss's stringent limitations on the amount of submitted analysis per day and the unchanged nature of its implementation, we opted to reuse Moss's benchmark results from our 2022 study (Maertens, Van Petegem, Strijbol, Baeyens, Jacobs et al. 2022). Additionally, it is not possible to measure the run time and memory usage of Moss, as it is a web service.

6.2.3. Results

Upon plotting the benchmark results for the Java and C datasets, it becomes evident that all tools demonstrate superior performance in identifying plagiarism in Java compared to C (figure 6.5). Both the default configuration (indicated by a vertical black line) and the best configuration (represented by the rightmost circle) of each tool exhibit significantly better predictive power on the Java dataset than on the C dataset. A plausible explanation for this disparity lies in the variation in inter-annotator agreement. This metric, Cohen (1960, 's) κ in this case, measures the agreement between annotators, accounting for the likelihood that they agree by chance. For the SOurce Code re-use (SOCO) datasets, this inter-annotator agreement metric greatly differs between the C ($\kappa = 0.480$, moderate agreement) and Java ($\kappa = 0.668$, substantial agreement) files (Flores et al. 2014). Thus, the quality of expert annotations appears to be a more critical factor than intrinsic differences between programming languages for the SOCO benchmark. In addition, the agreement scores themselves are not perfect, so the ground truth used to compare similarity detection results with is likely not completely correct.

³linux.die.net/man/2/getrusage

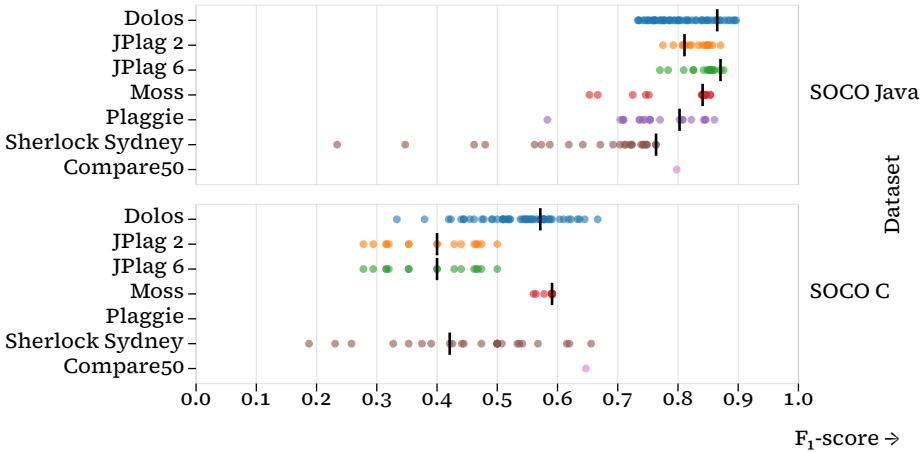


Figure 6.5. Predictive accuracy of plagiarism detection tools on all Java (top) and C (bottom) files in the SOCO benchmark. Circles indicate maximum F_1 -scores obtained with a particular configuration of a tool (parameter sweep). Vertical black lines indicate maximum F_1 -scores obtained with the default configuration of a tool. Higher F_1 -scores correspond to better predictive accuracy.

Figure 6.6 visualises the similarity distribution of plagiarised pairs within the C and Java datasets according to Dolos. With the C dataset there is more overlap between plagiarised and non-plagiarised pairs than with the Java dataset. However, there are a few pairs in the Java dataset classified as plagiarism with very low similarity values, indicating that those will likely only contain small fragments of plagiarised code.

SOCO Java

Examining the performance on the Java dataset, Dolos, Moss, Plaggie, JPlag 2, and JPlag 6 all achieve maximum F_1 -scores ranging between 0.8 and 0.9 for their default configurations. Compare50 scores just below that with an F_1 -score of 0.789. The default configuration of Sherlock Sydney, which is also its optimal configuration for this corpus, trails slightly with a maximum F_1 -score of 0.764. Dolos attains the highest-scoring configuration (0.897) when using a k -gram length of 25 and a window size w of 12, closely followed by JPlag 6 (0.876) when using a minimum of 13 matching tokens, and JPlag 2 (0.871) when using a minimum of 19 matching tokens. However, in terms of default configurations, JPlag 6 outperforms Dolos with an F_1 -score of 0.871

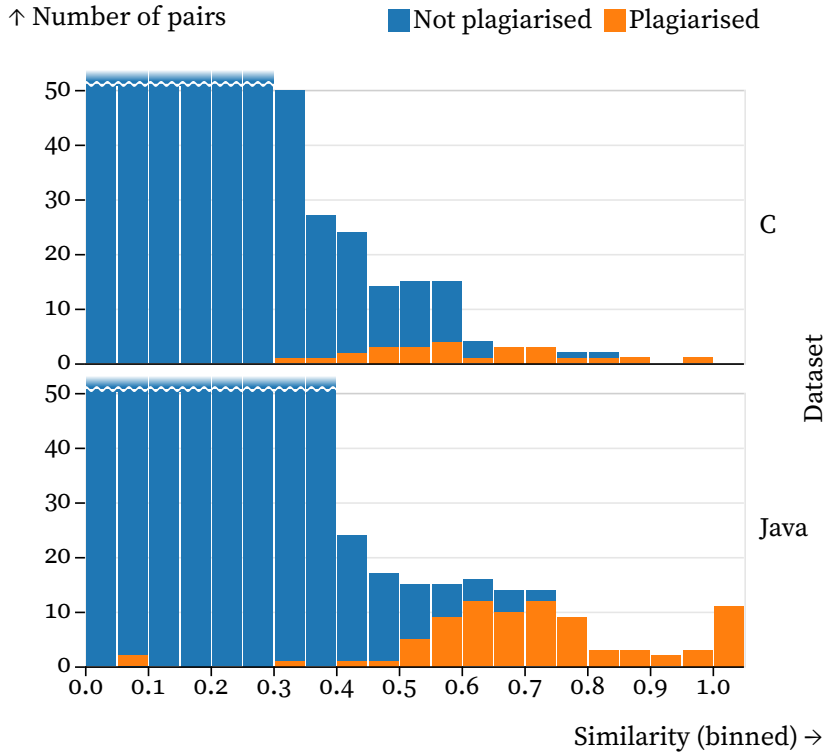


Figure 6.6. Distribution of pairwise similarities computed with default settings in Dolos for all C (top) and Java (bottom) files in the SOCO benchmark. True cases of plagiarism according to the SOCO metadata are shown in orange and false cases in blue. Because of the combinatorial explosion of pairs, the number of false cases with similarity values below 0.40 greatly exceeds the maximum value shown on the graph.

compared to Dolos's 0.865. This marks a notable improvement from JPlag 2's F_1 -score of 0.811 for its default configuration. Since JPlag 6 still uses the same default configuration parameter for Java as JPlag 2, this improvement is likely due to its new token normalisation feature.

Moss's performance remains consistent across configurations, indicating lesser dependence on parameter settings. Conversely, Sherlock Sydney exhibits a wide range of scores between 0.234 and 0.764. Despite JPlag and Plaggie utilising similar algorithms, there is a substantial difference in the performance of their default and optimal configurations.

SOCO C

Analysing the C dataset reveals noticeably lower maximum F_1 -scores. Compare50 scores surprisingly good with its single configuration, reaching an F_1 -score of 0.647. The default configurations for Dolos and Moss achieve similar scores, just under 0.6. The default configurations for Sherlock Sydney, JPlag 2, and JPlag 6 score substantially lower at 0.4 and 0.421, respectively. Once again, Moss' configurations yield consistent results, while Sherlock Sydney's scores vary widely. Dolos achieves the highest F_1 -score (0.667) using k -grams of length 10 and a window w of length 25, followed by Sherlock Sydney (0.656), and Compare50 (0.647).

JPlag 2 and 6 deliver identical results and perform poorly on this dataset, likely due to JPlag's focus on the Java programming language. Plaggie does not support the C programming language and thus has no results for this dataset.

The substantial variation in maximum F_1 -scores across tools, except for Moss, underscores the importance of well-chosen parameter settings for similarity analysis. In practical usage, validating optimal parameter settings is challenging, and users are unlikely to experiment with various configurations. Therefore, it is crucial that the default configuration consistently delivers robust results.

Moss's consistent results likely stem from the nature of its configurable parameter (option `-m`), which adjusts the maximum number of files in which a code fragment may appear before being ignored. This parameter significantly impacts similarity computations only when many files are plagiarised from the same source, as is the case with the Java dataset but not the C dataset (figure 6.7).

Moss and Dolos are the only tools that consistently yield good results for both the Java and C datasets. Dolos demonstrates slightly superior

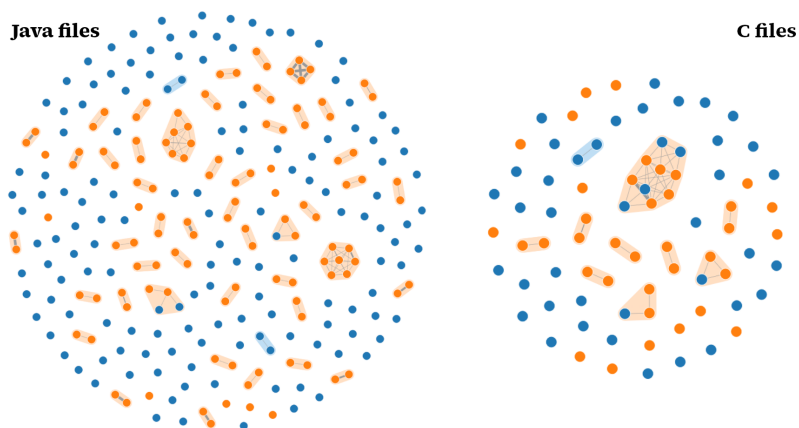


Figure 6.7. Plagiarism graphs by Dolos for all Java (left) and C (right) files in the SOCO benchmark, with similarity threshold 0.54 for Java and 0.58 for C. Orange nodes are involved in at least one case of plagiarism according to the metadata of the SOCO benchmark, whereas blue nodes are not involved in any confirmed cases of plagiarism. Clusters of blue nodes indicate false positives, whereby Dolos considers a pair to be plagiarism and the ground truth does not. Single, unconnected orange nodes indicate false negatives with Dolos missing a pair labeled as plagiarism in the annotations.

	Case #1	Case #2	Case #3	Case #4	Case #5	Case #6	Case #7	Case #8
Dolos -	1	1	1	1	1	1	1	1
JPlag -	1	1	1	1	1	9	8	4
Sherlock Sydney -	1	4	1	161	1	276	3 956	9
Compare50 -	1	1	1	1	2	3	1	1

Figure 6.8. Similarity rank assigned by Dolos, JPlag (version 2 and 6), Sherlock Sydney, and Compare50 to the plagiarised pair in each case of our private dataset. JPlag 2 and 6 are combined into one, as they report identical results. A higher rank indicates that that tool considers other pairs more similar than the pair indicated with plagiarism, therefore instructors using that tool are less likely to discover that case of plagiarism. These cases were initially discovered using Dolos, so these results have an inherent bias towards this tool.

overall predictive accuracy across the entire benchmark. Its optimal configuration achieves the highest scores for both programming languages, and its default configurations rank second for Java and third for the C dataset. Thus, Dolos proves to be highly competitive with current state-of-the-art tools across multiple programming languages.

It is noteworthy that none of the tools achieved an F_1 -score above 0.9 for the SOCO benchmark, indicating that each tool’s predictions contain mismatches with expert annotations. This highlights the limitations of source code similarity detection tools and expert annotations, emphasising that the ultimate decision to classify cases as plagiarism should never be automated and requires human review (Weber-Wulff 2019).

Private dataset

Figure 6.8 illustrates the ability of Dolos, JPlag, and Sherlock Sydney to identify plagiarised pairs within our private dataset. JPlag reports identical results for this dataset between versions 2 and 6, as this dataset uses the Python programming language. Both JPlag, Compare50, and Dolos effectively detect high similarities for the plagiarised pairs. While Dolos accurately reports all known plagiarised pairs as most similar, Compare50 ranks six out of the eight cases first, and JPlag identifies five out of eight cases as the most similar, but both tools remain this pair within the top 10 most similar pairs.

Sherlock Sydney, on the other hand, identifies three out of eight cases as most similar, and a fourth case within the top 10. These cases are almost identical copies with minimal obfuscations. However, the remaining four cases are obscured among other pairs due to the more

extensive obfuscation techniques applied, such as variable renaming and the addition of comments.

It would be erroneous to infer from these results that JPlag or Compare50 are less capable of detecting genuine instances of plagiarism. The reported rank of the plagiarised pair is not necessarily a good criterion for the plagiarism detection capability of a similarity detection tool. Additionally, the cases in our private dataset were initially discovered using Dolos, thereby introducing an inherent bias towards this tool. It is likely that Dolos did not discover certain plagiarism cases that would have been found by other similarity detection tools. Although Dolos did not initially report all plagiarised pairs as the most similar at the time of discovery, these cases were instrumental in refining our similarity detection pipeline. One such enhancement is the integration of a syntax tree preprocessing step, described in section 7.2.2.

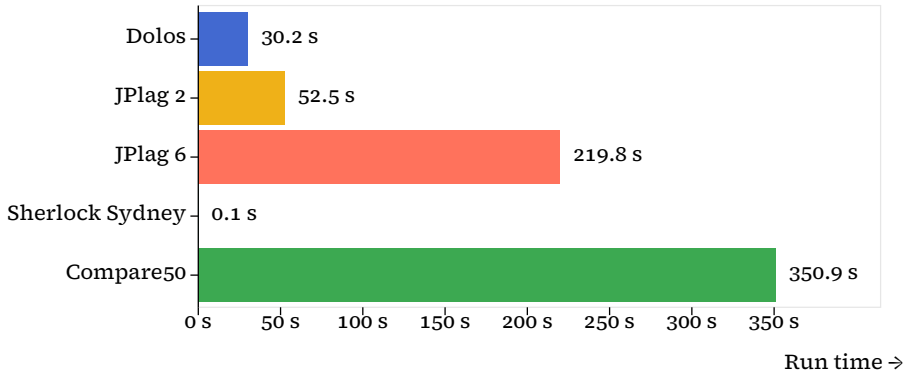
From these observations, we can conclude that for our private dataset, any similarity detection tool leveraging the a parser or lexer for similarity detection is a robust choice. Conversely, tools like Sherlock Sydney, which do not utilise a parser, would have overlooked half of the plagiarised cases.

Plutokiller

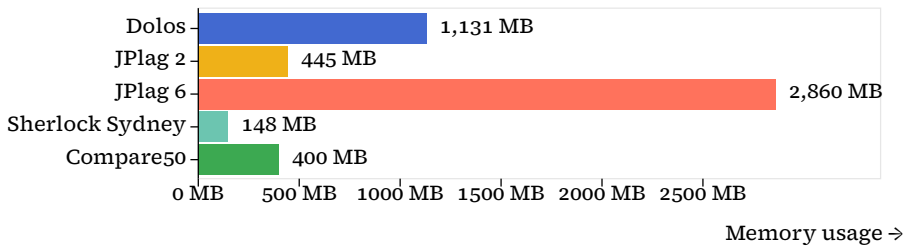
The primary objective of the Plutokiller dataset is to assess the resource usage of each similarity detection tool. For this dataset, we executed the benchmark three times and compared the average of run time and memory usage for each tool. Figure 6.9 visualises the benchmark results for this dataset. Moss is not included, because as a web service, the run time heavily depends on server load and availability, and the memory usage of the local submission script is negligible. Additionally, Plaggie is excluded from this benchmark because it does not support analysing Python files.

Sherlock Sydney demonstrates the lowest resource consumption, with a mean run time of 0.144 seconds and a memory usage of 148 MB, which is substantially lower than the other tools. However, Sherlock Sydney also delivers the lowest quality results, both on the SOCO datasets and our private dataset.

Compare50 has the slowest runtime of all similarity detection tools, requiring almost 6 minutes to process this dataset. However, the runtime of Compare50 is highly dependent on the desired number of reported output pairs. Reducing this number will greatly improve Compare50's runtime, but can cause it to miss some plagiarised pairs. For this



(a) Total run time (in seconds) for each tool to analyse the Plutokiller dataset. Sherlock Sydney is clearly the fastest, only requiring a few milliseconds to complete the analysis. Dolos completes the analysis in 30 seconds. JPlag increased its run time over the years, going from 52 seconds with JPlag 2, to 219 seconds with JPlag 6, more than 4 times slower. Compare50 takes 350.9 seconds to process this dataset.



(b) Total memory usage in MegaBytes (MB) for each tool to process the Plutokiller dataset, measured using the maximum resident set size reported on Unix. Sherlock Sydney requires the least amount of memory, followed by Compare50, then JPlag 2, then Dolos, and then JPlag 6.

Figure 6.9. Run time and memory usage for each similarity detection tool, averaged over three independent runs, using the Plutokiller dataset with 1 162 Python submissions.

benchmark, the number of requested file pairs was set to the number of total files included in the benchmark.

There is a notable increase in JPlag's run time and memory usage between versions 2 and 6. The run time increased more than fourfold, from 52 seconds to 220 seconds. The memory usage surged from 445 MB to 2 860 MB. This substantial increase is likely attributable to the additional features and functionality implemented within JPLag in recent years.

Dolos ranks second in terms of run time (30 seconds), and fourth in memory usage (1 131 MB). The total resource usage of all tools remains within reasonable limits. Even JPlag 6's run time of 220 seconds translates to only 0.189 seconds per submission, and most modern computers possess sufficient memory to handle an analysis of this scale. However, as the number of submission pairs scales quadratically with the number of submissions, JPlag 6 will exhaust available memory before Dolos when dataset size increases.

6.3. Usability and User Experience

The benchmarks outlined in section 6.2 assess the predictive accuracy and resource consumption of similarity detection tools. While these metrics are crucial, they represent only one facet of a tool's overall utility. A primary objective of Dolos is to achieve excellence in user experience (UX) and user interface (UI) design, aspects not captured by the aforementioned benchmarks.

6.3.1. User Experience Questionnaire

To evaluate the usability of Dolos, we conducted a survey using the User Experience Questionnaire (UEQ) (Laugwitz et al. 2008; Schrepp et al. 2017). This questionnaire is designed to be a concise instrument for measuring the user experience of interactive products. It comprises 26 items organised into 6 categories or scales:

- **Attractiveness:** overall impression — do users find the tool appealing?
- **Perspicuity:** Is the tool intuitive and easy to learn?
- **Efficiency:** Can users accomplish their tasks without undue effort?

- **Dependability:** Do users feel in control of the interaction?
- **Stimulation:** Is the tool engaging and motivating to use?
- **Novelty:** Does the design exhibit creativity and capture user interest?

The attractiveness scale provides a general impression of the product. Perspicuity, efficiency, and dependability collectively determine the tool's usability or *pragmatic quality*. The remaining scales, novelty and stimulation, assess the user experience or *hedonic quality*.

UEQ offers a website⁴ and supplementary materials to assist researchers and designers in evaluating the user experience of their interfaces. The questionnaire is available in over 30 languages, accompanied by a handbook and worksheets to facilitate result processing. The worksheet for analysing results automatically compares the tool under evaluation with a benchmark of 452 product assessments involving 20 190 participants (Schrepp et al. 2017). This comparison enables researchers to gauge the relative usability and UX of their interface.

6.3.2. Methodology

We initiated a survey using the UEQ, inquiring whether participants had experience with Dolos, Moss, JPlag, Plaggie, or other source code similarity detection tools. For each tool a user had experience with, we requested them to complete the UEQ. The survey was offered in both English and Dutch, using the official UEQ translations.

Between 17th January and 1st April 2025, the Dolos Web UI featured a banner with a direct link to this survey. Additionally, we emailed all users who had contacted us regarding Dolos, requesting their participation and encouraging them to share the survey with colleagues. Subsequently, we processed the results using the data analysis worksheet provided on the UEQ website.

6.3.3. Results

Table 6.1 and figure 6.10 present the outcomes of the Dolos UEQ survey, compared to the general UEQ benchmark. The results indicate

⁴www.ueq-online.org

Table 6.1. Survey results of the UEQ for Dolos per scale, including confidence intervals ($p=0.05$). Values for each scale range from -3 (terrible) to 3 (excellent).

Scale	Mean	Std. Dev.	N	Confidence	Conf. interval
Attractiveness	1.667	0.687	11	0.406	1.261 – 2.073
Perspicuity	1.614	0.719	11	0.425	1.189 – 2.039
Efficiency	1.591	0.615	11	0.364	1.227 – 1.954
Dependability	1.455	0.714	11	0.422	1.032 – 1.877
Stimulation	1.727	0.493	11	0.291	1.436 – 2.019
Novelty	1.477	0.541	11	0.320	1.157 – 1.797

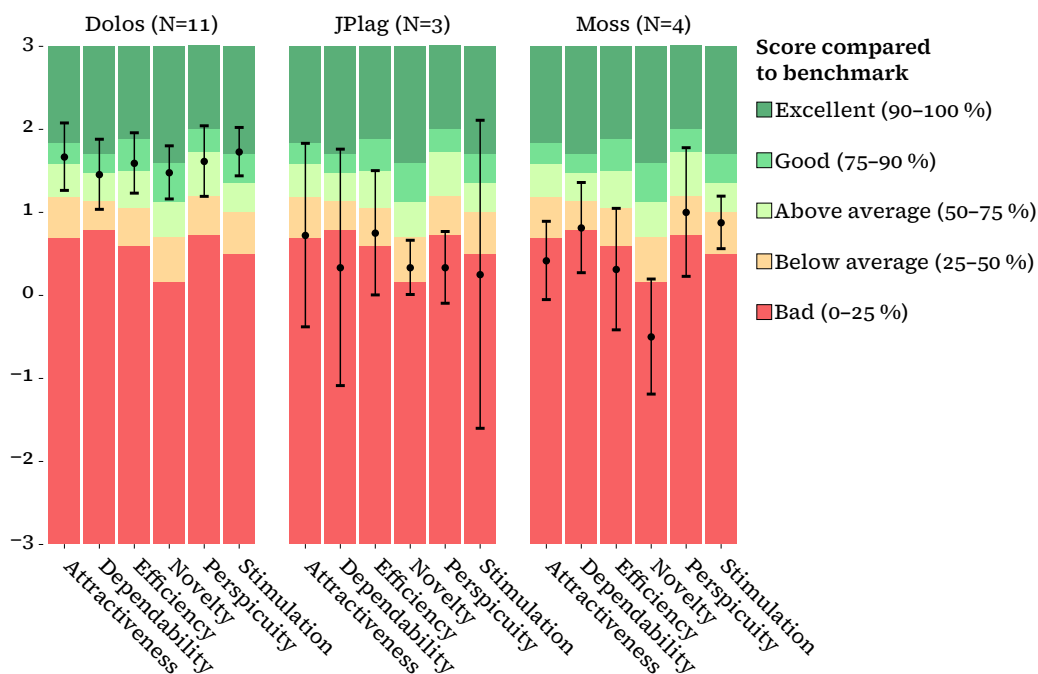


Figure 6.10. Results of the UEQ for Dolos (N=11), JPlag (N=3), and Moss (N=4), compared to the UEQ benchmark. The dot represents the mean scale value for each tool and the tick covers the 95% confidence interval.

a favourable perception, with each scale mean categorised as above average, good or excellent category.

In comparison to the benchmark, Dolos's lowest performance is on the perspicuity scale, though it still ranks in the upper tier of the above-average category. This scale reflects the ease with which users can learn and become familiar with the tool. Given that Dolos must convey complex information, this score is unsurprising. The other two scales within the pragmatic quality group, are efficiency and dependability. For these scales, Dolos's mean score lies at the intersection of above average and good, indicating that Dolos ranks among the top 25% of tools in the general benchmark evaluated on these criteria.

Dolos excels particularly in the stimulation (categorising as excellent) and novelty (categorising as good) scales. Together, these scales measure the hedonic quality, or the UX perception, of Dolos.

Both JPlag and Moss's mean values for each scale categorise as below average or bad. While their confidence intervals are wide, the values of Dolos are noticeably better than the other two tools. Dolos scores substantially better for the novelty scale than JPlag and Moss.

From these results, we cautiously infer that we have achieved our objective of developing a similarity detection tool with commendable UI and UX that is both user-friendly and easy to use.

6.3.4. Limitations

Given that the study primarily targeted Dolos users, the results are likely biased towards our tool. This bias is evident in the responses themselves, with one respondent explicitly praising Dolos.

Another limitation is the modest number of responses to the questionnaire. Only 11 out of 29 respondents fully completed the UEQ questions for Dolos, with even fewer completing the survey for JPlag ($n=3$) and Moss ($n=4$). While we do include their results in figure 6.10 for comparison, their confidence interval is too wide to draw meaningful conclusions.

The UEQ manual stipulates that 20 to 30 responses yield stable results, aiming for a precision (confidence interval width) of 0.5. The data analysis worksheet specifies a precision range for our scales between 0.582 and 0.850.

Therefore, the results of our UEQ study should be interpreted with caution and not considered highly reliable. Nevertheless, they suggest that users hold a positive perception of Dolos's UI and UX.

6.4. Case study

This section is based on the eponymous section of our journal article “Dolos: Language-agnostic plagiarism detection in source code” (Maertens, Van Petegem, Strijbol, Baeyens, Jacobs et al. 2022). Note that this section describes the application of Dolos in introductory programming courses led by professor Peter Dawyndt. For a more comprehensive description of the plagiarism cases discussed herein, please refer to the [supplementary material](#) of the original article. My contribution to this section in the original article was reviewing the original text drafted by Peter Dawyndt. I have modified the original text to align with the style and structure of this dissertation and added a new section about the impact of Generative Artificial Intelligence (GenAI).

This section explores the design and implementation of plagiarism prevention and detection strategies for an introductory programming course at Ghent University, with a particular emphasis on online learning. In particular, we look into the role plagiarism detection tools play in implementing these strategies and illustrate how their features can be practically applied. Initially, Moss was used in the early iterations of this course. However, starting in 2020, Dolos was introduced in programming courses at Ghent University, Belgium.

We also report on our approach to handling plagiarism cases identified at various stages of the course. Although a consistent plagiarism policy has been maintained across multiple editions of the course, there is a notable increase in plagiarism during tests and exams conducted remotely due to the COVID-19 pandemic (2020–2021 edition). Finally, in 2024–2025 there is a drastic impact of GenAI in multiple courses.

6.4.1. Course structure

The introductory programming course at Ghent University is conducted annually over a 13-week semester, spanning from September to December. The course is taken by a diverse cohort of undergraduate, graduate and postgraduate students from various disciplines, predominantly sciences but excluding computer science. For the 2020–2021 edition, 440 students were enrolled.

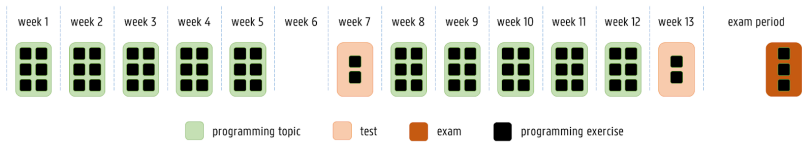


Figure 6.11. Outline of the Python Programming course that runs once per academic year across a 13-week semester. Students submit solutions to Dodona for ten series with six mandatory exercises, two tests with two exercises and an exam with three exercises. Collaboration among small groups of students is expected for the mandatory exercises, but no collaboration is allowed during tests and exams.

Throughout the course, students submit solutions to programming exercises via the online learning environment Dodona (Van Petegem, Maertens et al. 2023), where they receive immediate automated feedback on each submission, including during tests and exams. This feedback enables students to identify and rectify potential errors in their code and submit revised solutions.

Each week, the course focuses on a specific topic of the Python programming language with six programming exercises assigned for completion by the following week’s deadline (figure 6.11). These mandatory exercises are automatically graded through unit tests evaluated in Dodona. Students have the opportunity to work on these exercises during weekly computer labs, where they can collaborate in small groups and seek assistance from teaching assistants. Submissions are also accepted outside of lab sessions.

The course includes two graded tests — one mid-term and one at the semester’s end — where students have two hours to complete two programming exercises. Additionally, a final exam is administered post-semester, allowing students three and a half hours to solve three programming exercises. Tests and exams are conducted on-campus under supervision. Students are permitted to use their personal computers and access the internet “read only”: they can consult documentation, forum posts, exercise solutions, and so on. Communication with others, including artificial intelligence (AI)-backed tools like programming assistants and chatbots, is prohibited.

6.4.2. Plagiarism prevention

Students are permitted to check their solutions for correctness on Dodona as often as they wish, even after the submission deadline, without incurring penalties. This reduces the pressure to plagiarise, as students do not feel limited by the number of allowed submissions, and receive extensive feedback that allows them to continue improving their solution. This approach necessitates a sufficiently large pool of potential solutions for exercises to prevent resolution through guesswork.

Several strategies have been implemented to discourage students from copying and modifying source code from others. Our approach varies depending on the type of assessment; formative assessment (mandatory exercises) or summative assessment (tests or exams). Ghent University considers plagiarism a form of fraud. When invigilators or evaluators suspect a student of plagiarism, teachers must initiate a formal procedure where an examination board determines whether disciplinary measures are warranted. These measures can range from adjusting the student's score to exclusion from the university for up to 10 years. The lecturer strictly adheres to these rules for tests and exams, and address this topic during the first lecture. To ensure the validity and reliability of "open book/open Internet" tests and exams, new exercises are created and assignments are avoided where solutions or parts thereof are readily available online.

The approach to mandatory exercises is less straightforward, as collaboration among small groups of students can be beneficial for learning (Prince 2004). Students are encouraged to collaborate in groups of no more than three students, exchanging ideas and strategies rather than sharing literal code. Each edition of the course uses a new selection of mandatory exercises compiled from previous test and exam exercises, newly created exercises, and exercises last used four or more editions ago. By avoiding the reuse of recent exercises, the possibility of solutions being exchanged between students from one year to another is reduced.

Dolos is used to monitor submitted solutions for mandatory exercises, both before and at the deadline. The limited number of possible solutions for the initial mandatory exercises makes it challenging to link high similarity to plagiarism, as submissions contain only a few lines of code and implementation strategies are limited. As the number of possible solutions increases, so does the number of highly similar solutions which are reliable indicators of code exchange among larger groups of students. Strikingly, this often occurs among students

enrolled in the same study programme (figure 6.12). Typically, this phenomenon emerges in week 3 or 4 of the course, at which the topic of plagiarism is discussed in the next lecture. During this course, the pseudonymised plagiarism graphs is presented as evidence and the lecturer emphasises that the learning effect dramatically decreases when working in groups of four or more students. Usually, in such groups, only one or two students actively learn, while others merely copy solutions. The lecturer addresses these students by highlighting that while they may be proficient in programming and inclined to share their solutions to assist peers, they are actually depriving their fellow students of learning opportunities. Following this lecture, the number of plagiarised solutions typically decreases substantially for mandatory exercises.

The primary goal of plagiarism detection at this stage is prevention rather than penalisation. The goal of this intervention is for students to take responsibility over their learning. The realisation that teachers can easily detect plagiarism, coupled with the impending test that evaluates individual programming skills, usually results in an immediate and sustained reduction in cluster sizes in the plagiarism graphs to a maximum of three students. Simultaneously, it signals that plagiarism detection is one of the tools used to identify fraud during tests and exams. The entire student body is addressed about plagiarism only once, without going into detail about how plagiarism detection itself works, as overemphasising this topic might be ineffective and may encourage students to bypass the detection process - time better spent on learning to code. Every three or four years, a cluster persists of students exchanging code for mandatory exercises over multiple weeks. In such case, these students are individually addressed to remind them of responsibilities, differentiating between those who share solutions and those who receive them.

Tests and exams have a well-delineated rule prohibiting verbal and digital communication. Under normal circumstances, prior to the COVID-19 pandemic, students were restricted in their communication as they took tests and exams on-campus under the supervision of human invigilators. However, they were permitted to use the internet to consult information resources. Following each test and exam, we use Dolos to detect and inspect highly similar code snippets among submitted solutions and to gather convincing evidence of code exchange or other forms of interpersonal communication. If a case is identified as plagiarism beyond reasonable doubt, the examination board is informed.

When exercises created for tests or exams are reused as mandatory

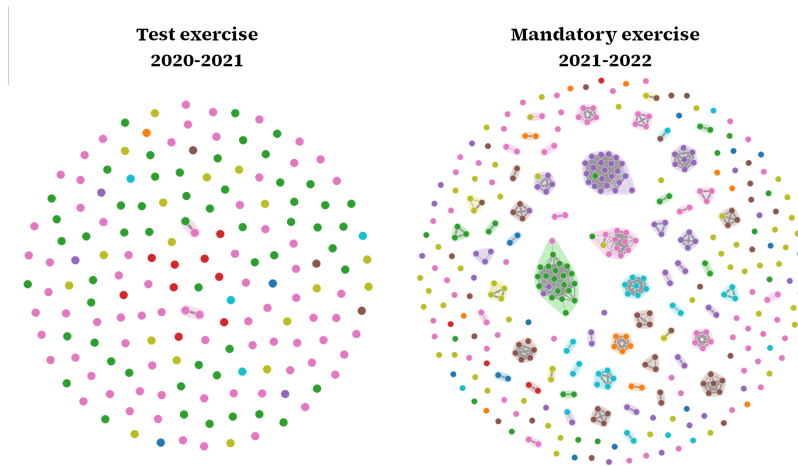


Figure 6.12. Dolos plagiarism graphs for the same Python programming exercise Pyramidal Constants on Dodona, created for a test of the 2020–2021 edition of the course (left) and reused as a mandatory exercise in the 2021–2022 edition (right). Graph constructed for the last submission before the deadline of 169 and 392 students respectively. Node colours indicate study programmes of students. Similarity threshold set to 0.76 (left) and 0.83 (right) respectively. All except two pairs of students submitted unique solutions during the test. Submissions for the mandatory exercise show that most students work either individually or in groups of two or three students. We also observe some clusters of four or more students that exchanged solutions and submitted them with hardly any modifications.

exercises, there is generally a clear distinction: no high-similarity pairs among solutions submitted during the test or exam, but multiple high-similarity pairs found among solutions submitted for the mandatory exercise (figure 6.12). This demonstrates that tracing high-similarity pairs is an effective method for monitoring student collaboration or communication while working on programming exercises.

6.4.3. Impact of COVID-19 pandemic

Where only a single case of suspected plagiarism was filed during all previous editions of the course since its first edition in 2006–2007, four cases of suspected plagiarism have been filed to the examination board during the 2020–2021 edition alone. Two of these cases occurred during tests and two during the exam. The evidence for each case was carefully documented for the examination board and Dolos’s visualisations were extremely useful for this purpose. The plagiarism graph illustrates that there is a high number of possible solutions and helps to convince students and board members that accidental high similarity is extremely unlikely. The compare view helps to disclose that substantial amounts of source code are identical copies or have been modified after copying as a deliberate act to obfuscate plagiarism.

When looking for explanations for this increase of plagiarism in tests and exams, the only fundamental difference in the organisation of the course is that all students took the 2020–2021 edition remotely due to the COVID-19 pandemic, including tests and exams. The lectures switched from on-campus colleges to live Zoom sessions. Students could ask online help from teaching assistants during lab sessions, primarily using the Dodona Q&A module for questions on specific solutions or using MS Teams for general assistance. MS Teams was recommended as an online collaboration tool, in combination with collaborative coding and pair programming services provided by modern Integrated Development Environments. Throughout the 2020–2021 edition of the course, there were no substantial differences in the occurrence of high-similarity pairs among solutions for the mandatory exercises.

Lack of direct human supervision during tests and exams seems the only reason for increased plagiarism. Apart from taking tests and exams remotely, the same rules applied and they had the same online nature as in all previous editions of the course. The lecturer deliberately refrained from using an online proctoring tool to remotely monitor the student’s behaviour and detect irregularities during tests

Sworn declaration

Today, I take part in the exam of the Programming course.

I hereby declare that I will fully comply with the regulations laid down by the **Education and Examination Code**, and that with specific reference to this exam I will not exhibit behaviour that might be considered fraud, such as communication with, offering assistance to, asking help from and/or cooperation with fellow students or third parties, or act in any other way that might be qualified as an irregularity and/or fraud.

I fully realize that by committing fraud of any kind during the exam I render myself liable to sanctions as laid down by article 78 of the **Education and Examination Code**, ranging from an adjustment of the obtained examination mark to a maximum exclusion of ten academic years.

[< PREVIOUS ACTIVITY](#)[MARK AS READ](#)[NEXT ACTIVITY >](#)

Figure 6.13. Sworn declaration that students have to digitally sign in Dodona at the start of each test or exam, together with a document describing the agreements for online exams. The contents of both documents are shared with students at the start of the course. The sworn declaration was newly introduced with the organisation of remote tests and exams during the COVID-19 pandemic.

and exams. Mainly to avoid extra stress that students experience while following proctoring protocols for the first time, and because it is believed that current proctoring tools only create a false sense of security and are too invasive on student privacy. A sworn declaration was introduced that students had to digitally sign at the start of each test and exam (figure 6.13), following a best practice recommended by Ghent University when taking remote exams. The document itself is not legally binding, but rather serves as a reminder of regulations in the Education and Examination Code that students accept when enrolling at the university. Having such an institutional honour code, actively informing students of this code, and signing these honour pledges to remember them about the code, has been observed to reduce the amount of cheating (LoSchiavo and Shatz 2011; McCabe, Trevino et al. 2001).

During this period, the lecturer remained convinced that “trust, but verify” is a viable strategy for organising trustworthy assessments in open and online learning environments. Even if high-stake tests are taken remotely. On the one hand, the strategy builds on educating students about their learning behaviour and making them aware of the importance of academic integrity. The instructor believes that four suspected cases in two tests and an exam for 440 students is acceptable. But only if cheaters get caught and appropriate disciplinary measures are imposed for proven cases of plagiarism beyond reasonable doubt.

6.4.4. Impact of GenAI

This subsection is a novel addition to this section, not included in our 2022 article. It was composed following an interview regarding professor Peter Dawyndt’s observations, combined with my own teaching experiences and those of other instructors.

Another significant event that profoundly impacted our programming courses was the advent of Generative Artificial Intelligence (GenAI) in the form of chatbots such as ChatGPT and programming assistants such as GitHub Copilot and Gemini Code Assist. Although OpenAI popularised this technology with ChatGPT at the end of 2022, it is only in the academic year 2024–2025 that a steep increase is observed in the adoption of this technology by students, accompanied by concerning side effects.

Course policy on GenAI

Following the public release of ChatGPT in 2022, instructors considered the effects of GenAI tools on their courses. They observed that students in introductory programming courses do not yet possess enough skill in reading and understanding code to critically evaluate the code snippets generated by GenAI. Consequently, students tend to accept almost everything suggested by AI-driven programming assistants (Prather, Reeves et al. 2024). While using GenAI is successful in most cases, there are instances where the programming assistant fails to suggest code solving the problem at hand. The students' over-reliance on the agent renders them unable to complete the task independently.

The instructors strongly believe that students need to learn the basics of programming first, learning to read and understand code. Only then will students be able to critically evaluate whether the suggestions of GenAI are appropriate and continue where the ability of GenAI ends.

To safeguard student learning and evaluation validity, the use of GenAI has been explicitly prohibited during tests and evaluations for these courses. The only change made to the course itself, is adding the following line in the rules of summative assessments:

The following actions are not allowed during the exam: [...] communicate with chatbots and programmer assistants that use artificial intelligence.

The lecturer reiterates this rule at the start of every summative assessment (evaluations and exams), both during an announcement and in the written rules referred to by the sworn declaration (figure 6.13). Supervisors actively check for violations by closely monitoring student screens during the assessment. Since its initial implementation, this rule and examples of violations have been expanded to account for the increased integration of GenAI in everyday services like search engines, office applications, and integrated development environments (IDES).

Observations during formative assessment

The rule prohibiting the use of AI only applies to summative assessment (evaluations and exams), and not to formative assessment (mandatory programming exercises) as it is completely unfeasible to enforce such a restriction. During the subsequent year 2023–2024, teaching assistants noticed some students occasionally using GenAI, but observed no other noteworthy changes in student behavior.

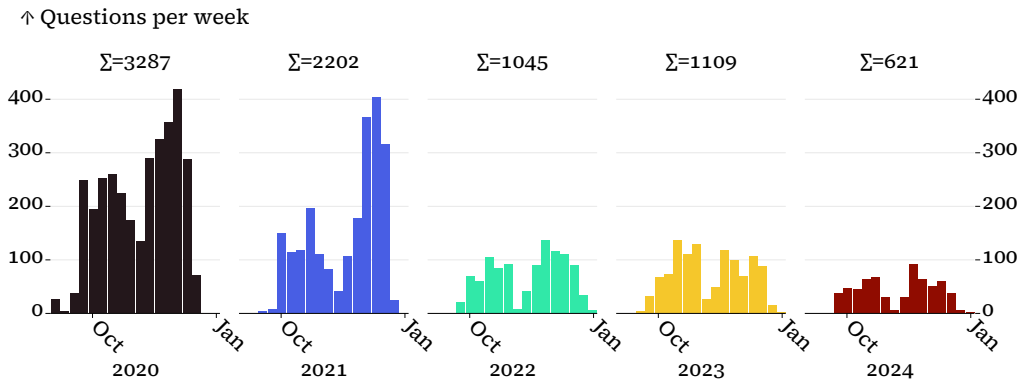


Figure 6.14. Number of online questions asked through Dodona in the last five offerings of the introductory programming course. Each year, we notice a drop halfway in the semester caused by the mid-term evaluation when there are no new programming exercises and students instead solve exercises from previous years (where questions are disabled). Due to the COVID-19 pandemic, the 2020 offering was completely online and 2021 went online during the second half of the semester, explaining the elevated number of questions during these offerings. This number stabilises to 1100 in 2022 and 2023 when the course returned back to on-campus practical lab sessions. A substantial decrease in the number of questions is noticeable in 2024, likely due to the increased adoption of GenAI.

However, during the current academic year 2024–2025, there is a drastic decrease in practical lab attendance and student questions, as shown in figure 6.14. This observation holds true in multiple programming courses, taught to both computer science (CS) and other curricula. Teaching assistants observe that instead, students often resort more quickly to GenAI tools to ask their questions, as the GenAI will not only respond to their prompts but also suggest code likely to solve the entire programming exercise — something that teaching assistants would rarely do.

Despite GenAI often completely solving programming exercises, there is no noticeable change in the amount of inter-student plagiarism during formative assessment. The pattern remains the same as in previous years, with the amount of collaboration slowly increasing during the first few weeks of the semester, peaking at week 4 with large clusters of students submitting similar or equal solutions. After an intervention by the instructor the following week, the collaboration decreases again to acceptable levels.

Observations during summative assessment

Despite the rules prohibiting the use of GenAI during open internet evaluations and exams, more students have been caught committing plagiarism using GenAI than inter-student plagiarism during the past five years combined. At the end of 2023–2024, the first student was caught using GenAI in the second exam period. In the current academic cycle of 2024–2025, a total of eight students have already been prosecuted for cheating using GenAI, with an equal amount of cases in computer science (CS) curricula, than in non-CS curricula. The first exam period of the second semester still needs to occur, as well the second exam period for both semesters, so the number of plagiarism cases involving GenAI is expected to increase. In contrast, there has not been a single case of inter-student plagiarism this academic year.

This drastic increase is possibly caused by the fact that the opportunity to plagiarise is now much greater. Whereas previously a student required someone capable of solving the exercises and willing to take the risk to participate in plagiarism, and to set up a covert communication channel with that student during the closely supervised assessment, this is now no longer necessary. More and more web services and applications integrate capable GenAI agents in their offering and enable them by default, such as search engines, office tools, and the IDEs. Even though the rules additionally mention that these agents are also not allowed, the border between them continuously fades, additionally making it harder to detect use of GenAI.

Additionally, students might start feeling a false sense of confidence, because with the use of GenAI they were able to solve most programming exercises (Prather, Reeves et al. 2024). Illustrating this, an instructor recalls a student approaching them after a mid-term test to confess their over-reliance on GenAI tools. Because the student understood everything suggested by the agent, they believed themselves to be able to reproduce similar code. It was only during the test itself where the student was left to their own devices, that they realised they had no idea how to solve the exercise without external help.

The phenomenon of over-reliance is not unique to GenAI. Students pre-GenAI would experience a similar confrontation with their over-confidence after relying too much on help from their friends during the summative assessment periods. This is one of the reasons for closely monitoring group sizes during formative assessment, the learning effect greatly diminishes when students collaborate in groups of four or more.

Consequences for instructors

Noticing this dramatic rise in fraudulent behaviour, instructors are seeking ways to ensure the validity of assessments. One definitive approach to prevent the use of GenAI by students, is to transition towards written examinations that do not permit computer access. However, this assessment format substantially diverges from the typical programming environment familiar to students. Instructors are justifiably aim to align the examination setting as closely as possible with the students' practice environment. For programming exercises, this necessitates continued internet access. Over the past years, universities have increasingly shifted to a bring-your-own-device (BYOD) policy, with fewer communal computer systems available to execute these assessments in a controlled environment. The latest developments with GenAI have partially halted this shift and in their quest for methods ensure valid assessments, tools have emerged to lock down unauthorised programs and web services on student devices, such as Safe Exam Browser⁵ and Schoology⁶.

Whereas during the mandatory remote assessments required by COVID-19, instructors placed their trust in the honesty of students and the capability to detect any violations, GenAI has caused this mentality to change. In addition to traditional similarity detection tools, instructors are considering invasive sandboxing and monitoring tools. This indicates a concerning erosion in their trust towards students (Luo 2025). The reduced attendance and number of questions come across as a lack of interest and eagerness to learn, profoundly affecting instructor motivation as well.

Given the widespread agreement that GenAI is effectively unbannable, instructors are actively considering how to incorporate this technology in their courses, even permitting its use during summative assessments. The ACM/IEEE/AAAI report on Computer Science Curricula in 2023 dedicates a section to integrating GenAI within CS curricula (Kumar et al. 2024, section 4.3). The report suggest that while students still need to learn how to write programs, the focus will shift from writing code themselves to prompting AI agents to generate code. This shift necessitates greater emphasis on teaching students how to design, comprehend, verify, and modify code. However, as many existing programming assignments for introductory programming courses can now be readily solved by GenAI, new assignments must be crafted to assess these additional competencies. The question of how educators

⁵safeexambrowser.org

⁶schoolyear.com

should adapt their assignments to the rapidly advancing capabilities of GenAI remains open.

Chapter 7.

Experimental prototypes

While establishing the foundational principles of Dolos and demonstrating its efficacy across diverse scenarios, we performed a series of exploratory experiments to identify potential enhancements for Dolos and related work. These experimental prototypes were developed as part of master's theses in collaboration with the Dodona team. While the majority of these experiments did not directly translate into features in the current iteration of Dolos, each contributed valuable insights that indirectly facilitated its enhancement. Some experimental outcomes offered promising avenues for further research; however resource constraints have thus far precluded us from advancing these prototypes.

We have categorised these experiments into three domains: evaluation (section 7.1), matching algorithms (section 7.2), and visualisations (section 7.3).

7.1. Evaluation

To objectively assess the efficacy of tools designed to aid in source code plagiarism detection, robust evaluation methods are essential. However, as Novak et al. (2019) highlighted, this area of research currently suffers from a notable gap:

What was observed while answering our research questions, is that there are missing standard datasets, metrics for evaluation, and objective comparisons of plagiarism detection tools.

This deficiency complicates the determination of optimal techniques and approaches for detecting plagiarism.

7.1.1. Challenges

Currently, there are no standardised, publicly available datasets accompanied by a “ground truth” that identifies plagiarised source files within the dataset. Furthermore, the absence of standardised metrics and methods for benchmarking tools that support plagiarism detection exacerbates this issue.

Datasets

Novak et al. (2019) recognise the SOurce Code re-use (SOCO) dataset as the most widely used resource in this domain. We also employed this dataset for the benchmarks described in section 6.2 of this dissertation. Refined by Flores et al. (2014) from a corpus published by Arwin and Tahaghoghi (2006), the SOCO dataset comprises a collection of Java and C source files. The dataset’s labels were derived by gathering suspicious pairs using JPlag (section 2.4.2) and manually inspecting the selected pairs. However, using a similarity detection tool to filter pairs for constructing a ground truth introduces a bias, as it discourages the detection of cases not initially identified by the similarity detector.

Additionally, the dataset’s quality is called into question due to the imperfect inter-annotator agreement scores: 0.480 (moderate agreement) for the C dataset and 0.668 (substantial agreement) for the Java dataset. Flores et al. (2014) do not specify whether the union or intersection of the annotator’s labels was selected, further complicating the interpretation of results achieved using this dataset.

We believe that the challenge of constructing a benchmark dataset for plagiarism is intrinsic to its domain. First, establishing a ground truth from actual student submissions is fraught with uncertainty, as it is impossible to definitely ascertain whether a student has plagiarised without their admission. Students who successfully evade detection have no incentive to confess, thereby concealing effective plagiarism strategies within datasets reliant on detected and prosecuted cases. Second, the definition of what constitutes plagiarism varies among instructors, necessitating a clear and precise delineation of the ground truth. Third, student submissions are personal data that cannot be shared publicly without the correct form of informed consent and anonymisation, complicating its construction.

Methods and metrics

A second challenge in evaluation source code similarity detection tools lies in the choice of methods and metrics employed. These tools produce pairwise comparisons of submissions, each associated with a similarity value. However, this value varies across different similarity detection tools, rendering direct comparisons of reported values impossible. Consequently, most studies include similarity detection tools as binary classifiers, categorising pairs as either plagiarised or non-plagiarised.

Nonetheless, converting similarity values into binary classifications can be approached in several ways. Similarity tools often report pairs exceeding a user-adjustable similarity threshold. While this threshold could serve as a basis for binary classification—where reported pairs deemed plagiarised and others are not—the relative similarity values are highly dependent on the programming assignments under examination. An alternative method involves selecting the top 20% of pairs, ranked by similarity, as plagiarised percentage of pairs sorted according to their similarity as plagiarised (Misc et al. 2016). However, the effectiveness of this approach is heavily influenced by the prevalence of plagiarism within the dataset.

Moreover, when evaluating similarity detection tools as binary classifiers, the selection of an appropriate metric remains a critical consideration. Novak et al. (2019) observed that various studies utilise different metrics, including precision, recall, sensitivity, F_1 , and F_β , further complicating the comparative analysis of tool performance.

7.1.2. Dataset annotation and benchmark standardisation

The contributions in this subsection are described in more detail in chapter 2 of the master’s dissertation under my supervision by Arne Jacobs (2022).

To address the aforementioned challenges, Jacobs (2022) developed a prototype benchmark suite, Apate, designed to evaluate similarity detection tools. The objectives of this benchmark were threefold:

- **Standardisation:** Facilitate the publication of the dataset and evaluation code, thereby establishing a standardised benchmark suite.
- **Obfuscation detection:** Assess the capability of similarity detection tools to identify various categories of code obfuscations.

In Greek mythology, Apate is the goddess of deceit.

- **Granular ground truth:** Provide a more detailed ground truth that specifies which parts of source files were plagiarised, enabling a finer-grained evaluation.

The Apaté prototype consisted of three primary components: an annotation format and demonstrative dataset, an evaluation metric, and a benchmark implementation.

Dataset and ground truth annotations

The annotations offered by the SOCO dataset indicate which pairs of source files exhibit code reuse, offering a binary classification: either a pair is deemed plagiarised or not. In contrast, within Apaté, we developed an annotation format that specifies the type of plagiarism present in the source files and pinpoints the location of the reused code regions. The format consists of a single JavaScript Object Notation (JSON) file for a given corpus that lists all instances of plagiarism, the corresponding source files, the precise locations of matching content within those files, and the obfuscation methods employed to transform one source file into another. This granular approach addresses the subjectivity of plagiarism by providing contextual information that justifies the classification.

The Apaté benchmark suite includes a manually crafted and annotated corpus of Python source files to demonstrate this format. However, due to the labour-intensive nature of creating and annotating such a corpus, the dataset is limited to 18 source files: two original submissions and several plagiarised derivatives utilising eight different obfuscation methods.

Evaluation metric

Given the granularity of the annotation format, we can employ a metric that evaluates a similarity detection tool's ability to identify matching segments of source code. The evaluation metric used in the Apaté benchmark calculates the true positive rate (TPR) and true negative rate (TNR) of detected lines of code, providing a more nuanced assessment of tool performance.

File	tool_name	TP	TN	FP	FN	TPR	TNR	obfuscation_techniques
main_B.py (None)	Erato()	0	459	19	0	nan	0.9602510460251	
main_B.py (None)	Erato+()	0	459	19	0	nan	0.9602510460251	
exact_copy_A.py (None)	Erato()	472	38	0	14	0.97119341563	1.0	
exact_copy_A.py (None)	Erato+()	486	0	38	0	1.0	0.0	
function_one_integrated.py (OM_16_LG)	Erato()	16	483	19	10	0.61538461538	0.962151394422	ObfuscationMethod.OM_16_LG
function_one_integrated.py (OM_16_LG)	Erato+()	24	483	19	2	0.92307692307	0.962151394422	ObfuscationMethod.OM_16_LG
reformat_A.py (OM_01_L)	Erato()	571	39	0	16	0.97274275979	1.0	ObfuscationMethod.OM_01_L
reformat_A.py (OM_01_L)	Erato+()	587	0	39	0	1.0	0.0	ObfuscationMethod.OM_01_L
added_pydoc.py (OM_02_L)	Erato()	252	39	0	372	0.40384615384	1.0	ObfuscationMethod.OM_02_L
added_pydoc.py (OM_02_L)	Erato+()	410	39	0	214	0.65705128205	1.0	ObfuscationMethod.OM_02_L
fulldoc.py (OM_02_L)	Erato()	115	38	0	546	0.17397881996	1.0	ObfuscationMethod.OM_02_L
fulldoc.py (OM_02_L)	Erato+()	214	38	0	447	0.32375189107	1.0	ObfuscationMethod.OM_02_L
comments.py (OM_02_L)	Erato()	149	39	0	374	0.28489483747	1.0	ObfuscationMethod.OM_02_L
comments.py (OM_02_L)	Erato+()	212	39	0	311	0.40535372848	1.0	ObfuscationMethod.OM_02_L
full_rename.py (OM_05_L)	Erato()	472	39	0	14	0.97119341563	1.0	ObfuscationMethod.OM_05_L
full_rename.py (OM_05_L)	Erato+()	486	0	39	0	1.0	0.0	ObfuscationMethod.OM_05_L
function_rename.py (OM_05_L)	Erato()	472	38	0	14	0.97119341563	1.0	ObfuscationMethod.OM_05_L
function_rename.py (OM_05_L)	Erato+()	486	0	38	0	1.0	0.0	ObfuscationMethod.OM_05_L
shuffle_class_definitions_01.py (OM_07_S)	Erato()	472	40	0	16	0.96721311475	1.0	ObfuscationMethod.OM_07_S
shuffle_class_definitions_01.py (OM_07_S)	Erato+()	488	0	40	0	1.0	0.0	ObfuscationMethod.OM_07_S
shuffle_function_definitions_01.py (OM_07_S)	Erato()	471	40	0	16	0.96714579055	1.0	ObfuscationMethod.OM_07_S
shuffle_function_definitions_01.py (OM_07_S)	Erato+()	487	0	40	0	1.0	0.0	ObfuscationMethod.OM_07_S
shuffle_statements.py (OM_07_S)	Erato()	473	39	0	13	0.97325102880	1.0	ObfuscationMethod.OM_07_S
shuffle_statements.py (OM_07_S)	Erato+()	486	0	39	0	1.0	0.0	ObfuscationMethod.OM_07_S
shuffle_function_arguments.py (OM_07_S)	Erato()	473	39	0	14	0.97125256673	1.0	ObfuscationMethod.OM_07_S
shuffle_function_arguments.py (OM_07_S)	Erato+()	487	0	39	0	1.0	0.0	ObfuscationMethod.OM_07_S
dead_code_01.py (OM_08_S)	Erato()	81	89	2	407	0.16598360655	0.978021978021	ObfuscationMethod.OM_08_S
dead_code_01.py (OM_08_S)	Erato+()	143	67	24	345	0.29303278688	0.736263736263	ObfuscationMethod.OM_08_S
dead_code_02.py (OM_08_S)	Erato()	56	100	0	432	0.11475409836	1.0	ObfuscationMethod.OM_08_S
dead_code_02.py (OM_08_S)	Erato+()	114	78	22	374	0.23360655737	0.78	ObfuscationMethod.OM_08_S
inlining.py (OM_10b_S)	Erato()	259	87	25	169	0.60514018691	0.776785714285	ObfuscationMethod.OM_10b_S
inlining.py (OM_10b_S)	Erato+()	302	85	27	126	0.70560747663	0.758928571428	ObfuscationMethod.OM_10b_S
outlining.py (OM_09_S)	Erato()	283	100	0	185	0.60470085470	1.0	ObfuscationMethod.OM_09_S
outlining.py (OM_09_S)	Erato+()	306	97	3	162	0.65384615384	0.97	ObfuscationMethod.OM_09_S

Figure 7.1. Screenshot of the Apat benchmark user interface (UI) presenting the results of a benchmark.

Benchmark implementation

The benchmark includes an implementation to execute these evaluations on source code similarity detection tools. It utilises the annotation format to measure each tool's capability to detect obfuscations and offers a UI to present the evaluation results (figure 7.1).

This benchmark includes source code similarity detection tools through an adapter interface: a single Python class featuring methods to run a similarity detection tool and collect the similarity detection results in a standardised format. Adding a new tool to the evaluation is straightforward, requiring only the implementation of these two methods for the tool in question, before running the benchmark.

7.1.3. Simulated plagiarism dataset

The contributions in this subsection are described in more detail in the master dissertation under my supervision by Raymond Bultynck (2023).

Manually creating an annotated corpus of plagiarised source files is laborious and typically results in a corpus limited to a single programming language. The review by Novak et al. (2019) highlights that the

literature predominantly focuses on Java and C++, with a notable lack of tools and datasets for other programming languages.

In the master’s dissertation of Bultynck (2023), we developed a prototype for a plagiarism dataset generator named Seth. This generator aims to apply obfuscations to a collection of source files, creating modified copies that mimic plagiarism.

Seth is the ancient Egyptian deity of deserts, storms, violence and foreigners.

Processing source files

Seth achieves this by parsing a source file into its concrete syntax tree (CST), applying obfuscations and transformations to the CST, and then reconstructing the source code from this modified CST. By leveraging the tree-sitter¹ parser generator, also employed by Dolos, Seth can utilise parsers for numerous programming languages through a unified Application Programming Interface (API) for the resulting CST.

However, syntax token types are not standardised across tree-sitter grammars. For instance, import statements in Python are labelled as `import_statement`, while in Java, they are labelled as `import_declaration`. To address this, Seth requires a configuration file for each programming language that maps the semantic meaning of some syntax tokens to the generator. The proof-of-concept developed during this thesis provides a configuration for Python and includes instructions for creating configurations for other programming languages.

Implementing obfuscations

Seth offers a generic interface for implementing various obfuscations. Each obfuscation modifies the underlying CST and records specific details about the exact transformations applied. The generator then summarises the alterations in a machine-readable annotation format.

The prototype does not implement all obfuscations described in section 1.3.2, as it was exploratory and focussed on viability rather than completeness. Additionally, some obfuscations are too language-specific to be applied successfully across different programming languages. For example, the advanced structural obfuscation method “changing statement specification” (OM_11_AS) requires specific knowledge about equivalent operations in particular programming languages.

However, the obfuscations identified as impossible by Bultynck (2023) are primarily advanced structural and logical obfuscations, which

¹tree-sitter.github.io/tree-sitter/

are also more challenging for students to apply. Therefore, there is a lesser need to simulate these obfuscations in generated datasets. Nonetheless, it remains possible to apply these obfuscations manually to the generated dataset.

7.1.4. Lessons learned

The experiments conducted by Bultynck (2023) and Jacobs (2022) underscore the challenges inherent in creating a source code plagiarism benchmark. Constructing a benchmark from actual student submissions is problematic due to the impossibility of distilling a perfect ground truth.

While we demonstrated the feasibility of creating synthetic datasets, either through manual curation, as with Apaté (section 7.1.2), or through automated generation, as with Seth (section 7.1.3), each approach presents its own limitations. Manually creating and annotating a plagiarism dataset is labour-intensive, whereas automatically generating plagiarism, while efficient, cannot simulate all forms of plagiarism.

A potential solution to these challenges lies in the development of a hybrid corpus. This corpus would combine confirmed instances of actual plagiarism with synthetic plagiarism, encompassing both automatically generated simple plagiarism, and manually recreated complex plagiarism. Such an approach might more closely approximate an adequate source code plagiarism benchmark.

7.2. Matching algorithms

A second area of experimentation focuses on Dolos's similarity detection pipeline. The Winnowing algorithm employed by Dolos, as described in chapter 3, selects representative fingerprints from each source file. These fingerprints are then used to calculate a similarity value and reconstruct matching code fragments. One known vulnerability of this algorithm is its susceptibility to automated attacks, where numerous small series of syntax tokens, such as redundant lines of code, are added to the original source file (Devore-McDonald and Berger 2020).

The current matching algorithms in Dolos can handle a large number of submissions but are still limited in scope. Dolos offers minimal

support for analysing multiple files per submission or multiple submissions per student, primarily due to the challenges posed by the sheer number of source file pairs that would need to be processed. A more optimised matching algorithm could mitigate these issues, enabling more advanced analyses.

7.2.1. Tree-matching

The contributions in this subsection are described in more detail in the master’s dissertation under my supervision by Arne Jacobs (2022).

To enhance Dolos’s robustness against obfuscations that introduce redundant lines of code, we explored the feasibility of performing similarity detection directly on the syntax tree instead of the serialised list of syntax tokens. This approach involves searching for “unordered subtree isomorphism” between the syntax tree of source files (Valiente 2021). The advantage of this method is that it allows matches to skip branches in the syntax tree, potentially improving detection accuracy.

Erato is one of the ancient Greek muses, goddess of literature, science and the arts.

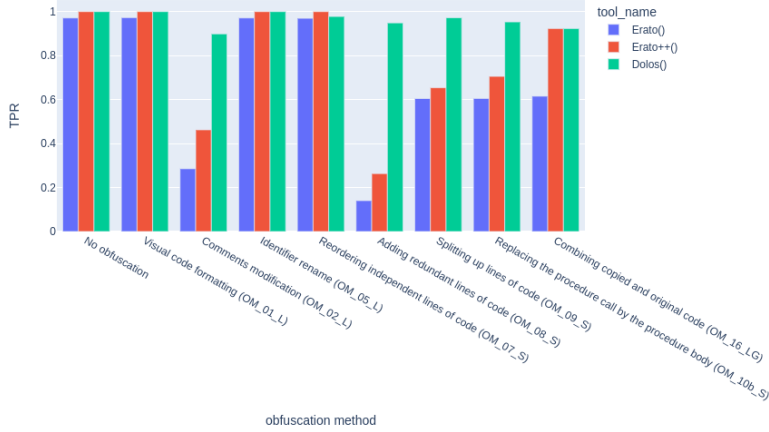
Jacobs (2022) developed the prototypes Erato and Erato++, which implement a maximum common subtree isomorphism algorithm (Dinitz et al. 1999; Valiente 2021). However, the results, evaluated using the Apat benchmark (section 7.1.2), were subpar, as shown in figure 7.2. While Erato and Erato++ reported fewer false positives on the specialised Apat benchmark, they yielded lower-quality results on the SOCO benchmark and operated significantly slower. Consequently, we discontinued the development Erato and Erato++.

Sağlam, Hahner et al. (2024) later successfully applied an alternative approach to achieve resilience against automated obfuscation attacks by normalising the syntax tree. Dolos could potentially integrate this approach, as the normalisation step should be applied before serialising the syntax tree and is independent of the matching algorithm.

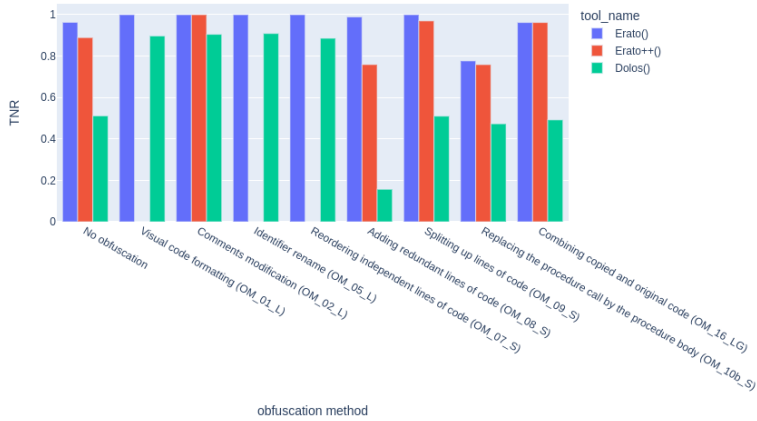
7.2.2. Syntax tree preprocessing

The contributions in this subsection are described in more detail in the master’s dissertation under my supervision by Michiel Lachaert (2025).

Teachers frequently observed that comments significantly influenced the similarity scores reported by Dolos. This is because tree-sitter includes comments as separate syntax tokens in the parse tree. In one



(a) True positive rate (TPR)



(b) True negative rate (TNR)

Figure 7.2. Results of Erato, Erato++ and Dolos on the Apaté benchmark. Erato and Erato++ report fewer matches, resulting in a higher true negative rate (TNR), but a lower true positive rate (TPR) than Dolos.

notable case, a plagiarised pair of submissions differing only by comments yielded a similarity percentage of 80%. While this was sufficient for detection, it also led to the discovery of another plagiarised pair by the same students for a different programming assignment that had a similarity percentage of 50% that differed in some comments and simple obfuscations. Furthermore, by only adding comments to identical submissions, we were able to reduce the similarity reported by Dolos to 32%.

In response to these observations, Lachaert (2025) introduced a pre-processing step to filter out comment tokens from the serialised syntax tree. This feature was publicly released in version v2.9.0 of Dolos. With this functionality enabled, Dolos now reports a similarity of 98% for the plagiarised pair differing only by comments, and 80% for the secondary plagiarism case with additional obfuscations. This enhancement now immediately highlights the second plagiarism pair as suspicious, where it was previously hidden.

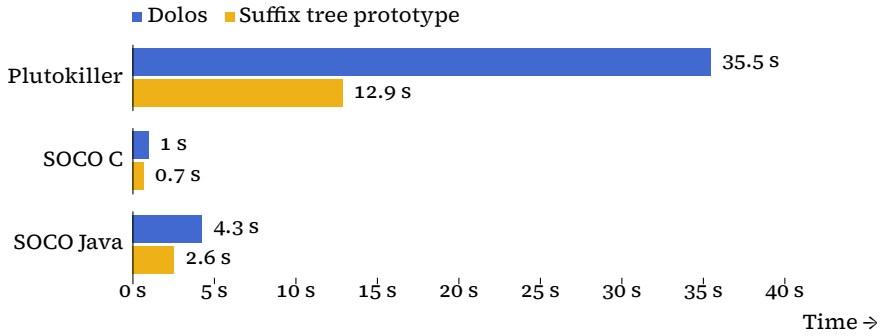
We have noted that Dolos now increasingly reports higher similarities for short submissions, which are often primitive attempts at solving an exercise with only initial code present. However, we accept this minor trade-off, as teachers are able to quickly identify these pairs as false positives.

7.2.3. Suffix trees

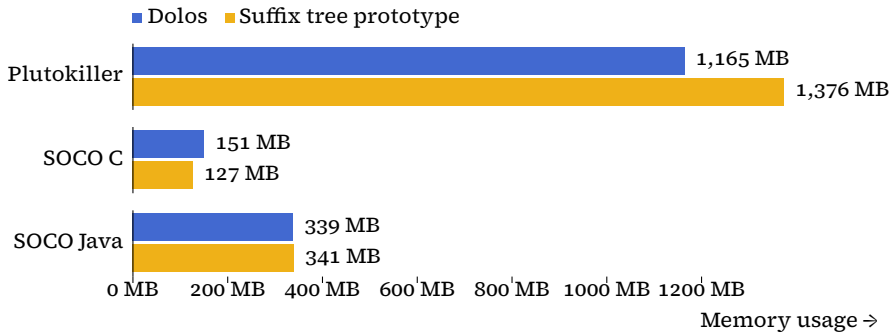
The contributions in this subsection are described in more detail in the master's dissertation under my supervision by Michiel Lachaert (2025).

A problem with the current similarity detection pipeline is the computational expense of calculating all submission pairs. When analysing n submissions, the number of submission pairs is $\frac{n(n-1)}{2}$, resulting in an inherent $\Theta(n^2)$ process. Additionally, calculating the longest common substring (LCS) for a single pair using a $\Theta(m^2)$ dynamic programming algorithm, where m is the length of the longest token sequence, further exacerbates the computational load.

A suffix tree is a trie data structure that stores all suffixes of a string, and can be constructed in $\Theta(n)$ time for a string of length n (Ukkonen 1995). Suffix trees enable efficient algorithms on the stored strings, such as finding the LCS in $\Theta(n)$ time for a string of length n . It is also possible to build a generalised suffix tree that stores multiple strings.



(a) Total run time (in seconds) of Dolos compared to the suffix tree prototype.



(b) Total memory usage of Dolos compared to the suffix tree prototype.

Figure 7.3. Run time and memory usage of Dolos compared to the suffix tree prototype, averaged over 3 independent runs. The prototype uses slightly more memory, 20% more for the Plutokiller dataset, but offers a 275% speedup on this dataset.

Lachaert (2025) prototyped a similarity detection pipeline for Dolos using a generalised suffix tree as index structure for storing the serialised syntax tokens from submissions. This prototype stores all tokens instead of a Winnowed set of fingerprints and enables faster computation of submission pairs with similarity and longest fragment metrics. Another advantage of the suffix tree is its intuitive nature as a data structure for storing match information, opening avenues for new efficient techniques and algorithms to enhance similarity detection. One such improvement would be the ability to store not only the final submission before the deadline but all submissions, enabling historical analysis to support plagiarism detection.

Figure 7.3 visualises the preliminary benchmark results comparing the current version of Dolos with the prototype using suffix trees. We notice a slightly increased memory footprint, up to 20% more when indexing the Plutokiller dataset with 1 162 files. However, this prototype offers a substantial run time improvement of 275%, reducing the analysis from 35 seconds to 12 seconds. While these results are promising, integrating this technique into Dolos's similarity detection pipeline and connecting it to the Dolos UI still requires some effort.

7.2.4. Lessons learned

While the evaluation in chapter 6 demonstrated that Dolos is a capable similarity detection system for aiding plagiarism detection, we acknowledge that our current implementation has certain limitations.

A tree-based approach, such as that employed by Erato and Erato++ (section 7.2.1), did not yield superior results compared to the current algorithm. The effectiveness of detecting matches using serialised syntax tokens in submissions was confirmed by achieving comparable similarity detection results with an algorithm using suffix trees (section 7.2.3).

We also discovered that small preprocessing steps, such as filtering out comments (section 7.2.2), can have a beneficial impact on the similarity detection results. While algorithmic improvements remain valid approaches, we should also explore similar preprocessing transformations, such as normalising the syntax tree to safeguard against more complex obfuscations (Sağlam, Brödel et al. 2024).

7.3. Visualisations

The contributions in this section are described in more detail in the master's dissertation under my supervision by Maxiëm Geldhof (2022).

The interactive dashboards provided by the Dolos UI have been a significant area of exploration. A primary goal of Dolos is to present the similarity detection results in a manner that allows instructors to efficiently and effectively check for plagiarism. The master's thesis by Geldhof (2022) explored various enhancements to the dashboards, with major contributions including the addition of the overview and cluster dashboards.

Two explored features, although not ultimately integrated into the Dolos UI, offered intriguing ideas: an interestingness metric (section 7.3.1) and semantic analysis (section 7.3.2).

7.3.1. Interestingness metric

From its early development stages, Dolos provided three metrics: similarity, total overlap, and the longest fragment. Each metric offers a unique perspective on the extent of plagiarism between a pair of submissions. However, this approach necessitates inspecting the same list of pairs multiple times, each time ranked according to a different metric.

To address this, Geldhof (2022) devised an *interestingness* metric, which combines these three metrics into a single value. This metric is calculated for each submission, rather than for a submission pair, and is determined as a weighted sum of the maximum similarity, total overlap and the longest fragment with any other submission under analysis. A dedicated page would list all submissions sorted by this interestingness metric, placing the most interesting submissions at the top.

7.3.2. Semantic analysis

Another experiment conducted during this thesis involved enhancing matched code fragments between submission pairs with semantic information. The rationale behind this enhancement, was that certain matching code fragments are more indicative of plagiarism than others.

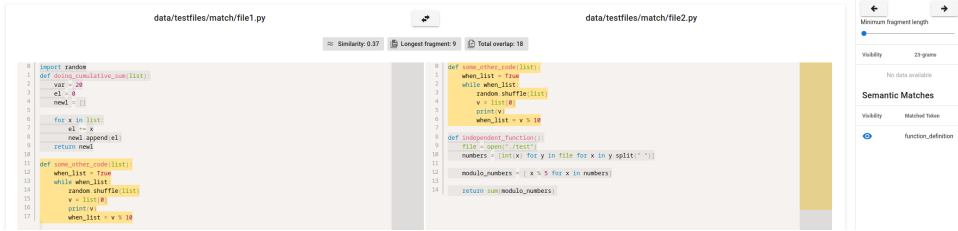


Figure 7.4. Screenshot of the comparison page indicating a matching function using the semantic analysis functionality.

For instance, a matching code fragment consisting solely of import statements is not indicative of plagiarism. The Dolos similarity detection pipeline often matches these import statements, even when the imports themselves differ, as identifiers are masked to protect against obfuscations involving identifier renaming. In programming languages like Java, which frequently employ lengthy import statement prefaces, these matches could obscure genuine instances of plagiarism.

To mitigate this issue, Geldhof (2022) introduced a semantic analysis step to identify the program components included in a match. This prototype could detect whether a matching code fragment encompassed a class, function, or loop and display this information in the Dolos UI on the comparison page (figure 7.4).

However, this prototype encountered performance issues, significantly slowing down the analysis. We concluded that the performance impact outweighed the benefits, leading to the decision not to include this functionality in the Dolos UI.

7.3.3. Lessons learned

The experiments conducted by Geldhof (2022) led to a major redesign of the Dolos UI that eventually culminated in Dolos v2.0 (section 4.8.4). We further expanded upon the overview and cluster dashboards initially prototyped during this master thesis, integrating these pages into the Dolos UI. These additions are extensively described in section 4.3 and section 4.5.

Another outcome of the interestingness and semantic experiments was a shift towards a submission-centered approach, rather than a

pairs-centered one. The submission dashboard (section 4.7) evolved from these ideas, and has been positively received by Dolos users.

Chapter 8.

Conclusions

In section 1.5 we identified substantial shortcomings in addressing source code plagiarism in educational contexts. In response, we formulated the research goal of developing a new source code similarity detection tool that adheres to modern standards with the following features:

- Proven, high-quality algorithms for similarity detection.
- Language-agnostic with broad programming language support and easy extensibility.
- Excellent user interface (UI) and user experience (UX)
- Visualisations that assist educators in preventing and detecting plagiarism in various settings (formative and summative assessment).
- Easy to use and install.
- Open-source and flexible to accommodate diverse use cases.
- Good software design, well-maintained and up to date dependencies.
- Respecting teacher and student privacy and compliant with privacy regulations.
- Conforming to modern security standards.

In this chapter, we conclude this doctoral research by summarising our results, discussing the impact of our work, and exploring avenues for further improvements. Finally, we conclude with a critical examination of Generative Artificial Intelligence (GenAI) and the impact on programming courses and computer science curricula.

8.1. Results

The primary outcome of this dissertation is Dolos, a new software ecosystem for source code similarity detection. Dolos includes a self-hostable web service and Application Programming Interface (API), a command-line interface (CLI), and software libraries designed to aid in detecting plagiarism in source code. We can summarise the main features and contributions to the state of the art as follows:

Dolos offers **high-quality algorithms** for similarity detection. Its similarity detection pipeline, discussed in detail in chapter 3, builds upon the Winnowing-algorithm championed by the popular similarity detection tool Moss (section 2.4.1). Using benchmarks, we demonstrated that Dolos performs on par with other similarity detection tools in terms of its predictive power (section 6.2).

Powerful and innovative visualisations complement the similarity detection pipeline by presenting its results in interactive dashboards. These dashboards allow users to quickly assess the amount of plagiarism and zoom in on the details when necessary. Where other tools present an overwhelming list of pairwise comparisons, Dolos offers a more structured view with its plagiarism graph visualisation, and focuses on clusters and individual submissions.

Dolos additionally serves as a **plagiarism prevention** tool. Its anonymisation feature allows showcasing the capabilities of Dolos to students, serving as an effective deterrence against plagiarism. This feature might be even more important than its plagiarism detection capabilities, as the end goal of instructors is to avoid plagiarism from happening.

We offer Dolos as an **open-source** and free-to-use ecosystem of modules providing **flexibility** for its different users. These modules cater to the needs of a diverse set of users. Instructors can perform similarity detection straight from their browser using the accessible Dolos web service (section 5.6). While power-users, researchers and developers can freely integrate, build, and improve upon Dolos. This flexibility has incubated its application outside educational contexts (section 8.2). The open-source Massachusetts Institute of Technology (MIT) licence ensures that this technology remains accessible (section 5.1.4).

The similarity detection pipeline of Dolos is **language-agnostic**, supporting a theoretically endless number of programming languages (section 5.2). The loose coupling between the similarity detection algorithms and the programming language parsers allow for rapidly

integrating new programming languages without reducing its quality.

Dolos provides a **secure** and **privacy-friendly** alternative to other similarity detection tools. The free-to-use instance of our web service does not track users and is designed with security in mind. We support and encourage our users to self-host this service using containers (section 5.7.3), allowing them to use a user-friendly web service without compromising their and their students' privacy.

All these features are developed while focusing on **excellent UX and UI** design, to lower the barrier for instructors to check for plagiarism. Using user studies (section 6.3) and a survey (section 6.3.1), we confirmed that Dolos is accessible and user-friendly. We specifically removed the hurdle of installing software by offering a free-to-use instance of our web service. This allows Dolos to be integrated directly into learning environments, enabling instructors to seamlessly perform source code similarity detection analysis within their browser (section 5.6.2).

8.1.1. Research contributions

While Dolos as a similarity detection system is a concrete and practical contribution to the state-of-the-art, we additionally describe our secondary research contributions.

Techniques for similarity detection tools

We demonstrated the capability of generalised parser generators in similarity detection tools. Existing similarity detection tools employed either parser modules specific for each programming language or use less-powerful lexers. We have built Dolos's similarity detection pipeline on top of the Tree-sitter parser generator framework that parses code into a syntax tree and are able to deliver detection results that are on-par with the state-of-the art.

Evaluation of similarity detection tools

We provided a novel methodology to benchmark similarity detection tools, described in section 6.3.2. Other articles evaluated the effectiveness of their tool by comparing a fixed similarity threshold, or only comparing the number of reported similarity pairs. Our benchmark methodology picks the optimal similarity threshold for each tool to classify data points, allows comparing different similarity metrics

fairly. Other researchers have started applying our methodology in their research (Slobodkin and Sadovnikov 2023).

Programming course design

Chapter 1 on educational source code plagiarism and the case study in section 6.4 provide an evidence-based approach to combat plagiarism in programming courses. The case study provides inspiration for ways to structure courses preventing plagiarism. Additionally, the case study illustrates the effects of major events like a global pandemic warranting remote examination and the introduction of GenAI as an easily accessible way to cheat.

Similarity detection visualisations

Dolos reimagines how similarity detection results with the visualisations offered by its UI. The interactive plagiarism graph (section 4.4) offers a quick and flexible method to explore similarity detection results. The plagiarism graph including automatic threshold estimation (section 4.3.2), cluster silhouette highlighting, and advanced navigation capabilities, offers far more functionality than similar visualisations present in the state of the art. The similarity histogram (section 4.3.1) innovates displaying the similarity distribution by only using the maximum similarity for each submission as a data point instead of using all pairwise similarities. Finally, Dolos pioneers a submission-first approach that, combined with clusters, reduces the information overload present in other tools when listing the quadratic explosion of pairwise comparisons.

8.2. Impact

Since its inception, Dolos has garnered attention from external instructors and researchers. Users frequently reach out with questions, bug reports, feedback and expressions of gratitude. By aggregating these testimonies, we have been able to chart the diverse applications of Dolos. Predictably, its primary impact lies in plagiarism detection for programming exercises. However, we were pleasantly surprised to discover that Dolos is also employed in non-educational contexts, underscoring its versatility.

8.2.1. Plagiarism detection in education contexts

Dolos was initially developed for detecting plagiarism in programming exercises, and this remains its most prevalent use case. Among its users, we observe a broad spectrum of applications. We have received communications from individual educators utilising Dolos, as well as from developers integrating it into their learning environments.

Integrations with other platforms

Various programming platforms have seamlessly integrated Dolos as their plagiarism detection service. We have previously described its integration with our own platform, **Dodona**, and that with Aalto University’s learning management system (LMS), **A+**, in section 5.6.2. However, we are aware of several other platforms that have adopted Dolos.

One of the first external platforms to integrate Dolos was **Codio**¹, a commercial platform for programming exercises based in the United States. Codio offers a code similarity checker powered by Dolos, in addition to other plagiarism detection methods like code playback and keystroke logging².

The Media Research Lab at the Technical University of Ostrava, Czech Republic, has integrated Dolos in their open-source LMS for programming courses, **Kelvin**³. Initially, they used Moss for their plagiarism detection needs but, dissatisfied with its sluggishness and unpredictable availability, they supplemented it with Dolos.

The University of Groningen, the Netherlands, has an in-house LMS **Themis**⁴, to support their computer science courses. They host their own instance of Dolos’s web service and have integrated this with Themis to meet their similarity detection needs.

8.2.2. Detecting plagiarism by LLMs

An unexpected application of Dolos’s similarity detection capabilities is in identifying plagiarism generated by Large Language Models (LLMs). These models have demonstrated exceptional proficiency in solving

¹codio.com

²codio.com/plagiarism-detection-for-source-code

³github.com/mrlvsb/kelvin

⁴gitlab.com/rug-digitallab/products/themis

programming tasks and programmers increasingly adopt GenAI with tools such as GitHub Copilot⁵ and Gemini Code Assist⁶. However, LLMs require vast datasets for training, primarily sourced from publicly available information on the internet, regardless of intellectual property rights prohibiting such use. This can lead to LLMs reproducing code fragments from their training data, causing programmers using GenAI to inadvertently commit copyright infringement as well.

Another problem arises when assessing LLM capabilities by checking their performance on publicly available tests or challenges. Since participants often publish their solutions to those tests and challenges, there is a risk that an LLM's training corpus might be *contaminated* with those solutions. This contamination may cause the LLM to reproduce these solutions instead of demonstrating the ability to generalise to unseen tasks (Carlini et al. 2023; Dekoninck et al. 2024).

Detecting those problems is, in essence, equivalent to detecting source code plagiarism. Consequently, multiple research articles refer to Dolos to check for dataset contamination or intellectual property violations (Abad et al. 2024; Chan et al. 2025; Riddell et al. 2024; Wang, Shao, Bhandari et al. 2025; Wang, Shao, Nabeel et al. 2025; Z. Yu et al. 2023). We highlight two of these articles below.

OpenAI's MLE-Bench

OpenAI⁷, the company behind the popular LLM GPT-3, its successors (GPT-4, OpenAI o1, etc.), and the widely used web service ChatGPT, has developed a benchmark for LLM agents performing machine learning engineering (MLE) tasks (Chan et al. 2025). This benchmark, named MLE-bench⁸, evaluates artificial intelligence (AI) agents on their capability to solve MLE tasks by challenging them to solve competitive problems published on the Kaggle⁹ data science competition platform. The AI agents must perform all steps required to train a new model for each task, including preparing datasets, training the model, and running experiments against their model. The benchmark compares each agent's model performance to that of the publicly available Kaggle leaderboard to assess the agent performance for MLE tasks.

⁵github.com/features/copilot

⁶codeassist.google

⁷openai.com

⁸openai.com/index/mle-bench/

⁹kaggle.com

One of the risks identified by OpenAI's researchers is that the LLM used by the agent might be contaminated with existing successful submissions for these challenges, as Kaggle competition participants tend to publish their solutions on public blogs or source code repositories. To address this risk, the benchmark uses Dolos to perform a similarity check of the agent's code against the top 50 associated solutions for the relevant Kaggle challenge. The benchmark disqualifies agent solutions when Dolos reports a similarity above 60% with any of the existing solutions.

CodeIPrompt

The Computer Security and Privacy Laboratory from Washington University in St. Louis, United States, conducted research to test whether LLMs violate intellectual property by producing code protected by a restrictive licence. To this end, Z. Yu et al. (2023) developed CodeIPrompt¹⁰, a tool designed to elicit LLMs to generate licenced code by prompting them with function signatures of such code fragments. The prompt response is checked for its similarity against the original code by using Dolos and JPlag (section 2.4.2).

All LLMs investigated, including GPT-4 and Copilot, were found to violate intellectual property by generating code under a restrictive licence. This study further revealed that publicly available datasets intended for training or fine-tuning LLMs contain code under such licences.

The article continues by suggesting approaches to avoid these models from infringing intellectual property rights. Building further on the work of Z. Yu et al. (2023), Abad et al. (2024) presents such an approach, also using Dolos and JPlag to validate whether LLMs generate fewer protected code fragments when using their techniques.

8.2.3. Malware classification

In a software supply chain attack, malicious code is inserted into a software product, often indirectly by targeting one of the often thousands of dependencies of that software project (Ohm et al. 2020). Upon discovering such malware, security researchers try to identify other software infected by the same malicious code. They achieve this by searching for an indicator of compromise (IoC), such as a hash, code fragment,

¹⁰github.com/zh1yu4nyu/CodeIPrompt

or other byte string embedded in the malicious code, which may reveal potential infections. In response, malware authors frequently obfuscate their malicious code to diminish the likelihood of finding a successful IoC, while simultaneously rendering their malicious code more difficult to comprehend, concealing its true purpose. Although these obfuscations are often more extensive than those employed by students to hide plagiarism (section 1.3.2), a similarity detection tool using the syntax tree could aid in uncovering infected code fragments by penetrating these obfuscations.

A security researcher¹¹ contacted us about their use of Dolos to analyse a corpus of source files from a large scale software supply-chain attack. The researcher investigated a coordinated attack infecting over a thousand software packages within a large package repository. Employing Dolos, they analysed the similarity between the malicious source files and discovered that these files could be grouped into a handful of clusters. All malware files shared a similar function of downloading and extracting a second-stage malware program to further infect the current system. However, the exact payload containing this second-stage malware differed per cluster, providing opportunities for further analysis. Dolos successfully reduced the large corpus of malware to a smaller set of clusters, substantially reducing the workload of the security researcher.

Since this occurrence, we have no further knowledge of Dolos being applied in similar circumstances. We do think that this avenue is potentially underexposed and that a collaboration between the field of similarity detection and malware analysis might yield new and exiting insights.

8.3. Future work

While one should be mindful of feature creep, Dolos has several areas for improvement that would greatly benefit its users.

8.3.1. Multi-file and multi-submission analysis

Dolos currently considers each file as the sole submission of a student, and compares that file with all other files under analysis. However,

¹¹Upon request of the researcher, we have left out details that might identify them, their employer, or their research.

this approach does not support all types of programming assignments and may overlook potentially valuable indicators of plagiarism.

Multiple files per submission

Advanced programming courses often task students with large projects, whose submissions comprise multiple source files, sometimes in different programming languages. Analysing multiple files per submission is currently not directly supported by Dolos. Submitting multiple files per student will compare files of the same student with each other, complicating the UI with many superfluous file pair comparisons and slowing down the analysis.

While it is possible to concatenate all files belonging to a single student submission into one source file, the resulting analysis results can be confusing. The similarity detection pipeline can handle concatenated files without problems, as long as they use the same programming language. Parsers are flexible enough to construct a single syntax tree from these files that does not differ from single-file submissions.

This is how we use Dolos currently with larger projects.

A first issue with this approach is that this potentially creates matches crossing file boundaries. When the k -gram length is sufficiently small, a `class_end` followed by multiple `import` statements and a `class_declaration` might be large enough to match between all transitions across these file barriers. This artificially adds noise to the similarity value, reducing the effectiveness of the similarity score. This can be solved by not allowing matches to occur across file boundaries, but does require some additional bookkeeping in the similarity detection pipeline.

However, a bigger challenge arises when visualising a comparison between two submission pairs. We now notice that the side-by-side diff comparing two files (section 4.6.1) can become confusing because a code fragment from one file can match with multiple other files from the other submission. This comparison page will additionally need to support a nested directory structure and visualise similarities between files of both submissions. Deciding how to visualise this is a challenging endeavour that we have not found the resources for yet.

Multiple submissions per student

Many programming platforms allow students to update and resubmit a previous solution. Especially when the platform provides automated

feedback on each submission, students will often hand in a large number of submissions to test and improve upon their solution.

Dolos considers each student to have a single submission. When instructors use Dolos for plagiarism detection, they often analyse the final submission of each student. However, previous submissions might contain additional clues of plagiarism that are now missed. When students hand in the submission from another student, they often apply their obfuscation in multiple passes, submitting between each pass to check whether their obfuscations did not break the functionality of the code. A student sharing their code might also share a previous solution or improve upon their solution after sharing it. In these situations, an earlier pair of submissions exist with a higher similarity. Analysing all submissions for each student would allow detecting those historical submissions.

However, students will often hand in early versions of their solution to receive automated feedback. These short, early versions will likely be highly similar between students, causing high similarity values between submissions that are not indicative of plagiarism and should be ignored. Additionally, as each student can easily have dozens of submissions, the resulting pairwise comparisons increase quadratically, requiring extensive computational resources, especially if we want to support multi-file submissions. Trying out alternative index structures, like a generalised suffix tree (section 7.2.3), is a first step towards taming the resulting computational complexity of such an analysis.

This feature would also introduce additional challenges to visualising the results. Different submissions of a student might contain separate clues, and bringing those to the attention of the user without confusing them will be challenging.

However, having access to all submissions does open up avenues for new visualisations analysing the student submission behaviour, as some types of behaviour might be more indicative of plagiarism than others. For example, a student making a dozen submissions with only small changes that are incorrect, followed by a full rewrite of their code that is suddenly correct, could be caused by that student copying the code from another student.

8.3.2. Improving the similarity threshold estimate

The current estimation of the similarity threshold, described in section 4.3.2, can be unreliable or yield unexpected results. This estimate

is based on the assumption that the optimal threshold is located in a local minimum of the similarity histogram. However, some reports do not have a local minimum, or only have local minima at similarity values that do not make sense. As this similarity threshold implicitly creates a certain classification between possible plagiarism (above this threshold), and possibly innocent (below this threshold), finding a better estimation method would be beneficial.

One possible approach could be to use the clusters as indicators of a well-chosen similarity threshold. We currently use single-linkage clustering, connecting submissions if their similarity exceeds the similarity threshold. Evaluation methods and metrics exist to determine whether clusters are well-chosen (D. Xu and Tian 2015), and we could pick the similarity threshold such that the resulting clusters are optimal for a given clustering metric.

Another possibility, as an alternative to the similarity threshold, is using a different clustering algorithm. Many clustering algorithms exist that try to automatically determine an optimal clustering based on the distance, or similarity, between data points (D. Xu and Tian 2015). This approach requires finding or modifying an appropriate clustering algorithm that does not cluster all data points, and can handle the absence of any clusters at all, as we do expect the majority of submissions not to be plagiarised. It would also require easy modification by instructors, as the clustering algorithm will likely not produce ideal results in all situations, and we would preferably want to keep giving instructors the flexibility to adjust the clustering.

8.3.3. Supporting instructors to report plagiarism

While Dolos offers an array of visualisations allowing instructors to inspect the analysis results in detail, teachers still have to extract that information and create a report when bringing a case of plagiarism in front of a plagiarism council. This requires a lot of work and might be one of the reasons why instructors avoid actively checking for plagiarism (Coren 2011).

Obfuscation distance

One of the most intensive tasks of inspecting a similarity detection report is comparing a suspicious pair of source files to determine whether the differences are accidental or caused by intentional obfuscations hiding plagiarism. Similar to how the edit distance between strings

describes the number of changes needed to go from one string to another, an interesting avenue for further research would be to describe the difference between two submissions in terms of an obfuscation distance.

Diffastic¹², a novel code diffing tool, uses a similar approach to compute the shortest path in terms of concrete syntax tree (CST) modifications. By considering each modification starting from the current CST as a weighted edge leading to the modified CST, diffastic applies the A* algorithm to compute the shortest path between the two CSTs in the resulting directed acyclic graph (Hart et al. 1968).

The CST modifications corresponding to the shortest path between the two submissions is the most likely sequence of obfuscations applied by the student. This path can be used to construct an explanation of the modifications between similar code fragments.

Report distillation

Once an instructor has collected enough signals indicating plagiarism, the time-intensive task starts of collecting this evidence in a case report. In an ideal scenario, instructors would be able to indicate these signals in a report-building tool, additionally tagging suspicious submissions or file pairs. Using these tags and indicated signals, the tools can prepare a report draft containing the necessary information.

As the information required in those reports can differ between institutions, the functionality of building the draft report is probably best not included in Dolos itself. However, Dolos could support tools to build those reports by allowing to add tags to a similarity detection report and offering an API to collect them.

8.4. Generative AI

While the aforementioned future work focuses on enhancing Dolos within the established paradigm of source code similarity detection, the field itself is currently grappling with a transformative shift driven by GenAI. Despite its many flaws, this technology produces text and other documents almost indistinguishable from those created by humans, and it is publicly available and free of charge. Since then, the

¹²diffastic.wilfred.me.uk

LLMs have improved with each iteration, as more resources are invested in their development.

One of the tasks at which GenAI excels, and continues to improve, is writing small code snippets to solve clearly defined tasks (M. Chen et al. 2021; H. Yu et al. 2024). This is exactly the format of assignments given during introductory programming classes, and LLMs have started performing better on these tasks than the average student (Svetkin 2024). This makes GenAI the perfect plagiarism machine (Watters 2025), and we have observed a drastic increase of this technology's adoption by students, as described in section 6.4.4.

GenAI poses a substantial risk to education, particularly in programming courses that evaluate students on their own devices using open internet exams. Similarity detection tools require the source of the plagiarism to be present in its analysed dataset, rendering plagiarism from GenAI undetectable by these tools. This landslide shift towards plagiarising from GenAI has rendered the field of educational plagiarism detection scrambling for new technologies, as this form of plagiarism seems virtually undetectable.

Early attempts to detect GenAI-generated code exist, but these contain fundamental flaws and have a high rate of false positives making them unreliable (Pan et al. 2024). The general academic consensus seems to be that this technology is unbannable (Prather, Leinonen et al. 2025). Additionally, Prather, Reeves et al. (2024) observe that students struggling with learning to program may be hindered by GenAI as it leaves them with an illusion of competence.

However, alongside its risks, there are also opportunities presented by this breakthrough technology. In an ironic twist of faith, source code similarity detection tools find their application in improving LLMs, reducing blatant plagiarism in their output. This opens up interesting new avenues for research using source code similarity tools in this rapidly evolving field.

GenAI is becoming increasingly proficient at tasks such as giving hints for programming assignments (Phung et al. 2023). AI-powered teaching assistants are emerging to help students without providing full answers (Denny et al. 2024; Liffiton et al. 2024). This technology could allow computer science education to scale dramatically, offering an enhanced learning process for more students with the same number of human teaching assistants as before.

As this technology becomes more widespread, we cannot ignore it. It is the obligation of instructors to teach about and with GenAI: its strengths and weaknesses, its risks and opportunities. In addition to

teaching students to become expert programmers, we will also need them to become expert code reviewers capable of critically evaluating generated code. This is especially important as these models grow more capable but also become more unreliable (Zhou et al. 2024). Otherwise, if new programmers solely start outputting AI-generated code, we can expect a future full of poisoned slop (Shumailov et al. 2024).

8.5. Concluding remarks

This dissertation embarked on addressing significant shortcomings in source code plagiarism detection within educational settings. The primary outcome, the Dolos software ecosystem, has successfully met the research goals outlined at the outset, delivering a proven, open-source, language-agnostic, privacy-focused, and user-friendly tool with powerful visualisations. As demonstrated by its adoption and diverse applications (section 8.2), Dolos has made a tangible contribution to the state of the art, not only in educational plagiarism detection but also in emerging areas such as LLM output analysis. The journey of developing Dolos has underscored the enduring need for robust tools that support academic integrity. While the future work outlined (section 8.3) presents exciting avenues for enhancing Dolos's capabilities, the rise of GenAI (section 8.4) undeniably reshapes the landscape. This development does not diminish the foundational principles of source code analysis, but rather calls for their evolution and adaptation. The ability to understand code similarity, provenance, and originality remains paramount, perhaps more so than ever, as educators and developers navigate this new technological frontier. Ultimately, this research provides a robust platform for similarity detection in Dolos and contributes to the ongoing dialogue on how to foster genuine learning and maintain academic integrity in computer science education. The challenge of GenAI is substantial, but it also presents an opportunity to re-emphasise critical thinking, ethical considerations, and the development of deeper programming understanding—goals that tools like Dolos can continue to support.

Appendix A.

Dolos CLI commands and options

This appendix lists the commands and command-line interface (CLI) options supported by `dolos-cli`. The Dolos CLI will output this information when using the `help` command or `-h`, `--help` flag.

A.1. `dolos` command-line options

Plagiarism detection for programming exercises

Options

- `-v --version` Output the current version.
- `-V, --verbose` Enable verbose logging.
- `-h, --help` Display help for this command.

Commands

- `run [options] <paths...>` Run an analysis and show the results.
- `serve [options] <path>` Serve the contents of an analysis.
- `help [command]` Display help for command.

A.2. `dolos serve` command-line options

Serve the contents of an analysis.

Required arguments

<path> — Path to the result of an analysis.

Options:

- no-open** Do not open the web page in your browser once it is ready.
- p --port <port>** Specifies on which port the webserver should be served.
- H --host <host>** Specifies on which host the webserver should be served.
- h, --help** Display help for this command.

A.3. dolos run command-line options

Run an analysis and show the results.

Required arguments

<paths...> — Input file(s) for the analysis. Can be a list of source code files, a CSV-file, or a zip-file with a top level info.csv file.

Options

- n, --name <name>** Resulting name of the report. Dolos tries to pick a sensible name if not given.
- l, --language <language>** Programming language used in the submitted files. Or char to do a character by character comparison. Detect automatically if not given.
- m, --max-fingerprint-count <integer>** The **-m** option sets the maximum number of times a given fingerprint may appear before it is ignored. A code fragment that appears in many programs is probably legitimate sharing and not the result of plagiarism. With **-m N** any fingerprint appearing in more than *N* programs is filtered out. This option has precedence over the **-M** option, which is set to 0.9 by default.

- M --max-fingerprint-percentage <fraction>** The **-M** option sets how many percent of the files the fingerprint may appear in before it is ignored. A fingerprint that appears in many programs is probably a legitimate fingerprint and not the result of plagiarism. With **-M N** any fingerprint appearing in more than *N* percent of the files is filtered out. Must be a value between 0 and 1. This option is ignored when comparing only two files, because each match appear in 100% of the files
- i, --ignore <path>** Path of a file with template/boilerplate code. Code fragments matching with this file will be ignored.
- L, --limit-results <integer>** Specifies how many matching file pairs are shown in the result. All pairs are shown when this option is omitted.
- s, --min-fragment-length <integer>** The minimum amount of kgrams a fragment should contain. Every fragment with less kgrams then the specified amount is filtered out. (default: 0)
- c --compare** Print a comparison of the matching fragments even if analysing more than two files. Only valid when the output is set to `terminal` or `console`.
- S, --min-similarity <fraction>** The minimum similarity between two files. Must be a value between 0 and 1
- f, --output-format <format>** Specifies what format the output should be in, current options are: `terminal/console`, `csv`, `html/web`. (default: `terminal`)
- p, --port <port>** Specifies on which port the webserver should be served.
- H, --host <host>** Specifies on which host `--output-format=web` should be served.
- o, --output-destination <path>** Path where to write the output report to. This has no effect when the output format is set to `terminal`.
- no-open** Do not open the web page in your browser once it is ready.
- sort-by <field>** Which field to sort the pairs by. Options are: `similarity`, `total overlap`, and `longest fragment` (default: `total overlap`)

- b, --fragment-sort-by <sort>** How to sort the fragments by the amount of matches, only applicable in terminal comparison output. The options are: kgrams ascending, kgrams descending and file order (default: file order)
- k, --kgram-length <integer>** The length of each kgram fragment. (default: 23)
- w, --kgrams-in-window <integer>** The size of the window that will be used (in kgrams). (default: 17)
- C, --include-comments** Include the comments during the tokenization process.
- h, --help** Display help for this command.

Bibliography

- Abad, J., K. Donhauser, F. Pinto and F. Yang (Dec. 2024). *Copyright-Protected Language Generation via Adaptive Model Fusion*. doi: [10.48550/arXiv.2412.06619](https://doi.org/10.48550/arXiv.2412.06619). arXiv: [2412.06619 \[cs\]](https://arxiv.org/abs/2412.06619).
- Acampora, G. and G. Cosma (Aug. 2015). “A Fuzzy-based Approach to Programming Language Independent Source-Code Plagiarism Detection”. In: *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pp. 1–8. doi: [10.1109/FUZZ-IEEE.2015.7337935](https://doi.org/10.1109/FUZZ-IEEE.2015.7337935).
- Aho, A., M. Lam, R. Sethi and J. Ullman (Aug. 2006). *Compilers: Principles, Techniques, and Tools*. 2nd edition. Boston: Addison Wesley. isbn: 978-0-321-48681-3.
- Ahtiainen, A., S. Surakka and M. Rahikainen (2006). “Plaggie: GNU-licensed Source Code Plagiarism Detection Engine for Java Exercises”. In: *Proceedings of the 6th Baltic Sea Conference on Computing Education Research Koli Calling 2006 - Baltic Sea '06*. Uppsala, Sweden: ACM Press, p. 141. doi: [10.1145/1315803.1315831](https://doi.org/10.1145/1315803.1315831).
- Alam, L. S. (2004). “Is Plagiarism More Prevalent in Some Forms of Assessment than Others?” In: *Beyond the Comfort Zone: Proceedings of the 21st ASCILITE Conference*. Perth, Australia, pp. 48–57. isbn: 0-9751702-4-4.
- Albluwi, I. (Dec. 2019). “Plagiarism in Programming Assessments: A Systematic Review”. In: *ACM Transactions on Computing Education* 20.1, 6:1–6:28. doi: [10.1145/3371156](https://doi.org/10.1145/3371156).
- Alexandron, G., J. A. Ruipérez-Valiente, Z. Chen, P. J. Muñoz-Merino and D. E. Pritchard (May 2017). “Copying@Scale: Using Harvesting Accounts for Collecting Correct Answers in a MOOC”. In: *Computers & Education* 108, pp. 96–114. issn: 0360-1315. doi: [10.1016/j.compedu.2017.01.015](https://doi.org/10.1016/j.compedu.2017.01.015).
- Arwin, C. and S. M. M. Tahaghoghi (Jan. 2006). “Plagiarism Detection across Programming Languages”. In: *Proceedings of the 29th Australasian Computer Science Conference*. Vol. 48. ACSC '06. AUS: Australian Computer Society, Inc., pp. 277–286. isbn: 978-1-920682-30-9.
- Bogomolov, E., V. Kovalenko, Y. Rebryk, A. Bacchelli and T. Bryksin (June 2021). *Authorship Attribution of Source Code: A Language-Agnostic Approach and Applicability in Software Engineering*. doi: [10.48550/arXiv.2001.11593](https://doi.org/10.48550/arXiv.2001.11593). arXiv: [2001.11593 \[cs\]](https://arxiv.org/abs/2001.11593).
- Bostock, M., V. Ogievetsky and J. Heer (Dec. 2011). “D³ Data-Driven Documents”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.12, pp. 2301–2309. issn: 1941-0506. doi: [10.1109/TVCG.2011.185](https://doi.org/10.1109/TVCG.2011.185).
- Brimble, M. and P. Stevenson-Clarke (Dec. 2005). “Perceptions of the Prevalence and Seriousness of Academic Dishonesty in Australian Universities”. In: *The Australian Educational Researcher* 32.3, pp. 19–44. issn: 2210-5328. doi: [10.1007/BF03216825](https://doi.org/10.1007/BF03216825).
- Brin, S., J. Davis and H. García-Molina (May 1995). “Copy Detection Mechanisms for Digital Documents”. In: *Proceedings of the 1995 ACM SIGMOD International Conference*

- on Management of Data. SIGMOD '95. New York, NY, USA: Association for Computing Machinery, pp. 398–409. isbn: 978-0-89791-731-5. doi: [10.1145/223784.223855](https://doi.org/10.1145/223784.223855).
- Brunsfeld, M. et al. (Jan. 2024). *Tree-Sitter/Tree-Sitter: Vo.20.9*. Zenodo. doi: [10.5281/ZENODO.4619183](https://doi.org/10.5281/ZENODO.4619183).
- Bultynck, R. (2023). “Seth: Simulatie van Plagiaatobfuscatie Door Studenten in Bron-code.” MA thesis. Ghent University.
- Bulut, N. and M. H. Halstead (June 1973). “Invariant Properties of Algorithms”. In: *SIGPLAN Not.* 8.6, pp. 12–13. issn: 0362-1340. doi: [10.1145/986953.986959](https://doi.org/10.1145/986953.986959).
- Carlini, N., D. Ippolito, M. Jagielski, K. Lee, F. Tramèr and C. Zhang (Mar. 2023). *Quantifying Memorization Across Neural Language Models*. doi: [10.48550/arXiv.2202.07646](https://doi.org/10.48550/arXiv.2202.07646). arXiv: [2202.07646 \[cs\]](https://arxiv.org/abs/2202.07646).
- Chae, D.-K., J. Ha, S.-W. Kim, B. Kang and E. G. Im (Oct. 2013). “Software Plagiarism Detection: A Graph-Based Approach”. In: *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*. CIKM '13. New York, NY, USA: Association for Computing Machinery, pp. 1577–1580. isbn: 978-1-4503-2263-8. doi: [10.1145/2505515.2507848](https://doi.org/10.1145/2505515.2507848).
- Chan, J. S., N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, L. Weng and A. Mądry (Feb. 2025). *MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering*. doi: [10.48550/arXiv.2410.07095](https://doi.org/10.48550/arXiv.2410.07095). arXiv: [2410.07095 \[cs\]](https://arxiv.org/abs/2410.07095).
- Cheers, H., Y. Lin and S. P. Smith (Oct. 2019). “A Novel Approach for Detecting Logic Similarity in Plagiarised Source Code”. In: *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, pp. 1–6. doi: [10.1109/ICSESS47205.2019.9040752](https://doi.org/10.1109/ICSESS47205.2019.9040752).
- Cheers, H., Y. Lin and S. P. Smith (2021). “Academic Source Code Plagiarism Detection by Measuring Program Behavioral Similarity”. In: *IEEE Access* 9, pp. 50391–50412. issn: 2169-3536. doi: [10.1109/ACCESS.2021.3069367](https://doi.org/10.1109/ACCESS.2021.3069367).
- Chen, B., C. M. Lewis, M. West and C. Zilles (July 2024). “Plagiarism in the Age of Generative AI: Cheating Method Change and Learning Loss in an Intro to CS Course”. In: *Proceedings of the Eleventh ACM Conference on Learning @ Scale*. Atlanta GA USA: ACM, pp. 75–85. isbn: 979-8-4007-0633-2. doi: [10.1145/3657604.3662046](https://doi.org/10.1145/3657604.3662046).
- Chen, G., Y. Zhang and X. Wang (Sept. 2011). “Analysis on Identification Technologies of Program Code Similarity”. In: *2011 International Conference of Information Technology, Computer Engineering and Management Sciences*. Vol. 1, pp. 188–191. doi: [10.1109/ICM.2011.240](https://doi.org/10.1109/ICM.2011.240).
- Chen, M. et al. (July 2021). *Evaluating Large Language Models Trained on Code*. doi: [10.48550/arXiv.2107.03374](https://doi.org/10.48550/arXiv.2107.03374). arXiv: [2107.03374 \[cs\]](https://arxiv.org/abs/2107.03374).
- Chomsky, N. and M. P. Schützenberger (Jan. 1959). “The Algebraic Theory of Context-Free Languages*”. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by P. Braffort and D. Hirschberg. Vol. 26. Computer Programming and Formal Systems. Elsevier, pp. 118–161. doi: [10.1016/S0049-237X\(09\)70104-1](https://doi.org/10.1016/S0049-237X(09)70104-1).
- Chuda, D., P. Navrat, B. Kovacova and P. Humay (Feb. 2012). “The Issue of (Software) Plagiarism: A Student View”. In: *IEEE Transactions on Education* 55.1, pp. 22–28. issn: 1557-9638. doi: [10.1109/TE.2011.2112768](https://doi.org/10.1109/TE.2011.2112768).
- Clarke, R. and T. Lancaster (June 2006). “Eliminating the Successor to Plagiarism? Identifying the Usage of Contract Cheating Sites.” In: *Proceedings Of 2nd Plagiarism: Prevention, Practice and Policy Conference 2006*. Newcastle, UK.

- Clarke, R. and T. Lancaster (July 2013). "Commercial Aspects of Contract Cheating". In: *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '13. New York, NY, USA: Association for Computing Machinery, pp. 219–224. isbn: 978-1-4503-2078-8. doi: [10.1145/2462476.2462497](https://doi.org/10.1145/2462476.2462497).
- Cohen, J. (Apr. 1960). "A Coefficient of Agreement for Nominal Scales". In: *Educational and Psychological Measurement* 20.1, pp. 37–46. issn: 0013-1644. doi: [10.1177/001316446002000104](https://doi.org/10.1177/001316446002000104).
- Collberg, C. and S. Kobourov (Apr. 2005). "Self-Plagiarism in Computer Science". In: *Communications of the ACM* 48.4, pp. 88–94. issn: 0001-0782. doi: [10.1145/1053291.1053293](https://doi.org/10.1145/1053291.1053293).
- Cook, S. A. (May 1971). "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. New York, NY, USA: Association for Computing Machinery, pp. 151–158. isbn: 978-1-4503-7464-4. doi: [10.1145/800157.805047](https://doi.org/10.1145/800157.805047).
- Coren, A. (Dec. 2011). "Turning a Blind Eye: Faculty Who Ignore Student Cheating". In: *Journal of Academic Ethics* 9.4, pp. 291–305. issn: 1572-8544. doi: [10.1007/s10805-011-9147-y](https://doi.org/10.1007/s10805-011-9147-y).
- Cosma, G. and M. Joy (May 2008). "Towards a Definition of Source-Code Plagiarism". In: *IEEE Transactions on Education* 51.2, pp. 195–200. issn: 1557-9638. doi: [10.1109/TE.2007.906776](https://doi.org/10.1109/TE.2007.906776).
- Cosma, G. and M. Joy (Mar. 2012). "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis". In: *IEEE Transactions on Computers* 61.3, pp. 379–394. issn: 0018-9340. doi: [10.1109/TC.2011.223](https://doi.org/10.1109/TC.2011.223).
- Cressey, D. R. (1953). *Other People's Money; a Study in the Social Psychology of Embezzlement*. Wadsworth Publishing Company, Belmont, California. isbn: 978-0-534-00142-1.
- Culwin, F., A. MacLeod and T. Lancaster (2001). "Source Code Plagiarism in UK HE Computing Schools". In: *Proceedings of the 2nd Annual LTSN-ICS Conference*. London, United Kingdom: LTSN Centre for Information and Computer Sciences, pp. 1–7.
- D'Souza, D., M. Hamilton and M. C. Harris (Jan. 2007). "Software Development Marketplaces: Implications for Plagiarism". In: *Proceedings of the Ninth Australasian Conference on Computing Education - Volume 66*. Vol. 66. ACE '07. AUS: Australian Computer Society, Inc., pp. 27–33. isbn: 978-1-920682-46-0.
- Dekoninck, J., M. N. Müller and M. Vechev (May 2024). *ConStat: Performance-Based Contamination Detection in Large Language Models*. doi: [10.48550/arXiv.2405.16281](https://doi.org/10.48550/arXiv.2405.16281). arXiv: 2405.16281 [cs].
- Denny, P., J. Leinonen, J. Prather, A. Luxton-Reilly, T. Amarouche, B. A. Becker and B. N. Reeves (Mar. 2024). "Prompt Problems: A New Programming Exercise for the Generative AI Era". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. New York, NY, USA: Association for Computing Machinery, pp. 296–302. isbn: 979-8-4007-0423-9. doi: [10.1145/3626252.3630909](https://doi.org/10.1145/3626252.3630909).
- Devore-McDonald, B. and E. D. Berger (Nov. 2020). "Mossad: Defeating Software Plagiarism Detection". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–28. issn: 2475-1421. doi: [10.1145/3428206](https://doi.org/10.1145/3428206).
- Dick, M., J. Sheard, C. Bareiss, J. Carter, D. Joyce, T. Harding and C. Laxer (June 2002). "Addressing Student Cheating: Definitions and Solutions". In: *SIGCSE Bull.* 35.2, pp. 172–184. issn: 0097-8418. doi: [10.1145/782941.783000](https://doi.org/10.1145/782941.783000).

- Dinitz, Y., A. Itai and M. Rodeh (May 1999). "On an Algorithm of Zemlyachenko for Subtree Isomorphism". In: *Information Processing Letters* 70.3, pp. 141–146. issn: 0020-0190. doi: [10.1016/S0020-0190\(99\)00054-X](https://doi.org/10.1016/S0020-0190(99)00054-X).
- European Commission (Jan. 2021). Q&A: COVID-19 Vaccination in the EU. https://ec.europa.eu/commission/presscorner/detail/en/qanda_20_2467. Press Release.
- Faidhi, J. A. W. and S. K. Robinson (Jan. 1987). "An Empirical Approach for Detecting Program Similarity and Plagiarism within a University Programming Environment". In: *Computers & Education* 11.1, pp. 11–19. issn: 0360-1315. doi: [10.1016/0360-1315\(87\)90042-X](https://doi.org/10.1016/0360-1315(87)90042-X).
- Flores, E., P. Rosso, L. Moreno and E. Villatoro-Tello (Dec. 2014). "On the Detection of Source Code Re-use". In: *Proceedings of the Forum for Information Retrieval Evaluation. FIRE '14*. New York, NY, USA: Association for Computing Machinery, pp. 21–30. isbn: 978-1-4503-3755-7. doi: [10.1145/2824864.2824878](https://doi.org/10.1145/2824864.2824878).
- Gehringer, E. (Oct. 2004). "Reuse of Homework and Test Questions: When, Why, and How to Maintain Security?" In: *34th Annual Frontiers in Education, 2004. FIE 2004. S1F/24-S1F/29 Vol. 3*. doi: [10.1109/FIE.2004.1408702](https://doi.org/10.1109/FIE.2004.1408702).
- Geldhof, M. (2022). "Dolos: Gebruiksvriendelijke En Toegankelijke Plagiaatdetectie Voor Dodona". MA thesis. Ghent University.
- Gibson, J. P. (July 2009). "Software Reuse and Plagiarism: A Code of Practice". In: *SIGCSE Bull.* 41.3, pp. 55–59. issn: 0097-8418. doi: [10.1145/1595496.1562900](https://doi.org/10.1145/1595496.1562900).
- GitHub (Oct. 2024). *Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges*. <https://github.blog/news-insights/octoverse/octoverse-2024/>.
- Gusfield, D. (May 1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. 1st edition. Cambridge England ; New York: Cambridge University Press. isbn: 978-0-521-58519-4.
- Hart, P. E., N. J. Nilsson and B. Raphael (July 1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. issn: 2168-2887. doi: [10.1109/TSSC.1968.300136](https://doi.org/10.1109/TSSC.1968.300136).
- Hellas, A., J. Leinonen and P. Ihanola (June 2017). "Plagiarism in Take-home Exams: Help-seeking, Collaboration, and Systematic Cheating". In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education. ITiCSE '17*. New York, NY, USA: Association for Computing Machinery, pp. 238–243. isbn: 978-1-4503-4704-4. doi: [10.1145/3059009.3059065](https://doi.org/10.1145/3059009.3059065).
- Husain, F. M., G. K. S. Al-Shaibani and O. H. A. Mahfoodh (June 2017). "Perceptions of and Attitudes toward Plagiarism and Factors Contributing to Plagiarism: A Review of Studies". In: *Journal of Academic Ethics* 15.2, pp. 167–195. issn: 1572-8544. doi: [10.1007/s10805-017-9274-1](https://doi.org/10.1007/s10805-017-9274-1).
- Jaccard, P. (1901). "Étude Comparative de La Distribution Florale Dans Une Portion Des Alpes et Du Jura". In: *Bulletin de la Société Vaudoise des Sciences Naturelles* 37.142, p. 547. issn: 0037-9603. doi: [10.5169/seals-266450](https://doi.org/10.5169/seals-266450).
- Jacobs, A. C. (2022). "Erato En Apate: Similariteitsanalyse Met Deelboomisomorfie En Een Kwalitatieve Benchmark Voor Plagiaatdetectie Op Broncode". MA thesis. Ghent University.
- Jones, K. O., J. Reid and R. Bartlett (Dec. 2008). "Cyber Cheating in an Information Technology Age". In: *Digithum* 10. issn: 1575-2275.

- Joy, M. and M. Luck (May 1999). "Plagiarism in Programming Assignments". In: *IEEE Transactions on Education* 42.2, pp. 129–133. issn: 1557-9638. doi: [10.1109/13.762946](https://doi.org/10.1109/13.762946).
- Joy, M., G. Cosma, J. Y.-K. Yau and J. Sinclair (Feb. 2011). "Source Code Plagiarism—A Student Perspective". In: *IEEE Transactions on Education* 54.1, pp. 125–132. issn: 1557-9638. doi: [10.1109/TE.2010.2046664](https://doi.org/10.1109/TE.2010.2046664).
- Karavirta, V., P. Ihanntola and T. Koskinen (July 2013). "Service-Oriented Approach to Improve Interoperability of E-Learning Systems". In: *2013 IEEE 13th International Conference on Advanced Learning Technologies*, pp. 341–345. doi: [10.1109/ICALT.2013.105](https://doi.org/10.1109/ICALT.2013.105).
- Karp, R. M. and M. O. Rabin (Mar. 1987). "Efficient Randomized Pattern-Matching Algorithms". In: *IBM Journal of Research and Development* 31.2, pp. 249–260. issn: 0018-8646. doi: [10.1147/rd.312.0249](https://doi.org/10.1147/rd.312.0249).
- Kumar, A. N., R. K. Raj, S. G. Aly, M. D. Anderson, B. A. Becker, R. L. Blumenthal, E. Eaton, S. L. Epstein, M. Goldweber, P. Jalote, D. Lea, M. Oudshoorn, M. Pias, S. Reiser, C. Servin, R. Simha, T. Winters and Q. Xiang (2024). *Computer Science Curricula 2023*. New York, NY, USA: Association for Computing Machinery. isbn: 979-8-4007-1033-9.
- Kyrilov, A. and D. C. Noelle (Nov. 2015). "Binary Instant Feedback on Programming Exercises Can Reduce Student Engagement and Promote Cheating". In: *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. Koli Calling '15. New York, NY, USA: Association for Computing Machinery, pp. 122–126. isbn: 978-1-4503-4020-5. doi: [10.1145/2828959.2828968](https://doi.org/10.1145/2828959.2828968).
- Kyrilov, A. and D. C. Noelle (Apr. 2016). "Do Students Need Detailed Feedback on Programming Exercises and Can Automated Assessment Systems Provide It?" In: *J. Comput. Sci. Coll.* 31.4, pp. 115–121. issn: 1937-4771.
- Lachaert, M. (2025). "Optimaliseren van Dolos Met Behulp van Suffixbomen Voor Plagiaatdetectie in Code". MA thesis. Ghent University.
- Lancaster, T. and F. Culwin (June 2004). "A Comparison of Source Code Plagiarism Detection Engines". In: *Computer Science Education* 14.2, pp. 101–112. issn: 0899-3408, 1744-5175. doi: [10.1080/08993400412331363843](https://doi.org/10.1080/08993400412331363843).
- Laugwitz, B., T. Held and M. Schrepp (2008). "Construction and Evaluation of a User Experience Questionnaire". In: *HCI and Usability for Education and Work*. Ed. by A. Holzinger. Berlin, Heidelberg: Springer, pp. 63–76. isbn: 978-3-540-89350-9. doi: [10.1007/978-3-540-89350-9_6](https://doi.org/10.1007/978-3-540-89350-9_6).
- Li, X. and X. J. Zhong (Oct. 2010). "The Source Code Plagiarism Detection Using AST". In: *2010 International Symposium on Intelligence Information Processing and Trusted Computing*, pp. 406–408. doi: [10.1109/IPTC.2010.90](https://doi.org/10.1109/IPTC.2010.90).
- Liffiton, M., B. E. Sheese, J. Savelka and P. Denny (Feb. 2024). "CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes". In: *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. Koli Calling '23. New York, NY, USA: Association for Computing Machinery, pp. 1–11. isbn: 979-8-4007-1653-9. doi: [10.1145/3631802.3631830](https://doi.org/10.1145/3631802.3631830).
- Liu, C., C. Chen, J. Han and P. S. Yu (Aug. 2006). "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis". In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '06. New York, NY, USA: Association for Computing Machinery, pp. 872–881. isbn: 978-1-59593-339-3. doi: [10.1145/1150402.1150522](https://doi.org/10.1145/1150402.1150522).

- LoSchiavo, F. M. and M. A. Shatz (2011). "The Impact of an Honor Code on Cheating in Online Courses". In: *Journal of Online Learning and Teaching* 7.2, p. 6.
- Luce, R. D. and H. Raiffa (1957). *Games and Decisions: Introduction and Critical Survey*. New York: John Wiley & Sons, Inc. isbn: 978-0-486-13483-3.
- Luo, J. (May 2025). "How Does GenAI Affect Trust in Teacher-Student Relationships? Insights from Students' Assessment Experiences". In: *Teaching in Higher Education* 30.4, pp. 991–1006. issn: 1356-2517. doi: [10.1080/13562517.2024.2341005](https://doi.org/10.1080/13562517.2024.2341005).
- Lupton, R. A., K. J. Chapman and J. E. Weiss (Mar. 2000). "International Perspective: A Cross-National Exploration of Business Students' Attitudes, Perceptions, and Tendencies Toward Academic Dishonesty". In: *Journal of Education for Business* 75.4, pp. 231–235. issn: 0883-2323. doi: [10.1080/08832320009599020](https://doi.org/10.1080/08832320009599020).
- Luxton-Reilly, A. and A. Petersen (Jan. 2017). "The Compound Nature of Novice Programming Assessments". In: *Proceedings of the Nineteenth Australasian Computing Education Conference*. ACE '17. New York, NY, USA: Association for Computing Machinery, pp. 26–35. isbn: 978-1-4503-4823-2. doi: [10.1145/3013499.3013500](https://doi.org/10.1145/3013499.3013500).
- Maertens, R., P. Dawyndt and B. Mesuere (June 2023). "Dolos 2.0: Towards Seamless Source Code Plagiarism Detection in Online Learning Environments". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2*. ITiCSE 2023. New York, NY, USA: Association for Computing Machinery, p. 632. isbn: 979-8-4007-0139-9. doi: [10.1145/3587103.3594166](https://doi.org/10.1145/3587103.3594166).
- Maertens, R., P. Dawyndt and B. Mesuere (June 2025). "Source Code Plagiarism Detection as a Service with Dolos". In: *Proceedings of the 30th ACM Conference on Innovation and Technology in Computer Science Education V. 2*. ITiCSE 2025. New York, NY, USA: Association for Computing Machinery, pp. 729–730. isbn: 979-8-4007-1569-3. doi: [10.1145/3724389.3731274](https://doi.org/10.1145/3724389.3731274).
- Maertens, R., M. Van Neyghem, M. Geldhof, C. Van Petegem, N. Strijbol, P. Dawyndt and B. Mesuere (May 2024). "Discovering and Exploring Cases of Educational Source Code Plagiarism with Dolos". In: *SoftwareX* 26, p. 101755. issn: 2352-7110. doi: [10.1016/j.softx.2024.101755](https://doi.org/10.1016/j.softx.2024.101755).
- Maertens, R., C. Van Petegem, N. Strijbol, T. Baeyens, A. C. Jacobs, P. Dawyndt and B. Mesuere (Mar. 2022). "Dolos: Language-agnostic Plagiarism Detection in Source Code". In: *Journal of Computer Assisted Learning* 38.4, pp. 1046–1061. issn: 1365-2729. doi: [10.1111/jcal.12662](https://doi.org/10.1111/jcal.12662).
- Maertens, R., C. Van Petegem, N. Strijbol, T. Baeyens, M. Van Neyghem, M. Geldhof, A. C. Jacobs, P. Dawyndt and B. Mesuere (Oct. 2024). *Dolos*. Zenodo. doi: [10.5281/zenodo.7966722](https://doi.org/10.5281/zenodo.7966722).
- McCabe, D. L., L. K. Trevino and K. D. Butterfield (1999). "Academic Integrity in Honor Code and Non-Honor Code Environments: A Qualitative Investigation". In: *The Journal of Higher Education* 70.2, pp. 211–234. issn: 0022-1546. doi: [10.2307/2649128](https://doi.org/10.2307/2649128). JSTOR: 2649128.
- McCabe, D. L., L. K. Trevino and K. D. Butterfield (July 2001). "Cheating in Academic Institutions: A Decade of Research". In: *Ethics & Behavior* 11.3, pp. 219–232. issn: 1050-8422, 1532-7019. doi: [10.1207/S15327019EB1103_2](https://doi.org/10.1207/S15327019EB1103_2).
- McCabe, D. L., L. K. Treviño and K. D. Butterfield (June 2002). "Honor Codes and Other Contextual Influences on Academic Integrity: A Replication and Extension to Modified Honor Code Settings". In: *Research in Higher Education* 43.3, pp. 357–378. issn: 1573-188X. doi: [10.1023/A:1014893102151](https://doi.org/10.1023/A:1014893102151).

- McGrath, J. (May 2024). *An 'artificial Sun' Achieved a Record-Breaking Fusion Experiment, Bringing Us Closer to Clean, Limitless Energy*. <https://www.businessinsider.com/west-tungsten-tokamak-fusion-record-plasma-2024-5>.
- Misc, M., Z. Sustran and J. Protic (2016). "A Comparison of Software Tools for Plagiarism Detection in Programming Assignments". In: *The International journal of engineering education* 32.2, pp. 738–748. issn: 0949-149X.
- Nguyen, P. T., J. Di Rocco, C. Di Sipio, R. Rubel, D. Di Ruscio and M. Di Penta (Aug. 2024). "GPTSniffer: A CodeBERT-based Classifier to Detect Source Code Written by ChatGPT". In: *Journal of Systems and Software* 214, p. 112059. issn: 0164-1212. doi: [10.1016/j.jss.2024.112059](https://doi.org/10.1016/j.jss.2024.112059).
- Novak, M., M. Joy and D. Kermek (June 2019). "Source-Code Similarity Detection and Detection Tools Used in Academia: A Systematic Review". In: *ACM Transactions on Computing Education* 19.3, pp. 1–37. issn: 1946-6226. doi: [10.1145/3313290](https://doi.org/10.1145/3313290).
- Ohm, M., H. Plate, A. Sykosch and M. Meier (2020). "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by C. Maurice, L. Bilge, G. Stringhini and N. Neves. Cham: Springer International Publishing, pp. 23–43. isbn: 978-3-030-52683-2. doi: [10.1007/978-3-030-52683-2_2](https://doi.org/10.1007/978-3-030-52683-2_2).
- OpenAI (Mar. 2024). *Video Generation Models as World Simulators*. <https://openai.com/index/video-generation-models-as-world-simulators/>.
- Ottenstein, K. J. (Dec. 1976). "An Algorithmic Approach to the Detection and Prevention of Plagiarism". In: *ACM SIGCSE Bulletin* 8.4, pp. 30–41. issn: 0097-8418. doi: [10.1145/382222.382462](https://doi.org/10.1145/382222.382462).
- Pan, W. H., M. J. Chok, J. L. S. Wong, Y. X. Shin, Y. S. Poon, Z. Yang, C. Y. Chong, D. Lo and M. K. Lim (May 2024). "Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education". In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET '24. New York, NY, USA: Association for Computing Machinery, pp. 1–11. isbn: 979-8-4007-0498-7. doi: [10.1145/3639474.3640068](https://doi.org/10.1145/3639474.3640068).
- Perkins, D. and F. Martin (Oct. 1985). *Fragile Knowledge and Neglected Strategies in Novice Programmers*. IR85-22. Tech. rep.
- Phung, T., V.-A. Pădurean, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla and G. Soares (Sept. 2023). "Generative AI for Programming Education: Benchmarking ChatGPT, GPT-4, and Human Tutors". In: *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 2*. Vol. 2. ICER '23. New York, NY, USA: Association for Computing Machinery, pp. 41–42. isbn: 978-1-4503-9975-3. doi: [10.1145/3568812.3603476](https://doi.org/10.1145/3568812.3603476).
- Prather, J., P. Denny, J. Leinonen, B. A. Becker, I. Albluwi, M. Craig, H. Keuning, N. Kiesler, T. Kohn, A. Luxton-Reilly, S. MacNeil, A. Petersen, R. Pettit, B. N. Reeves and J. Savelka (Dec. 2023). "The Robots Are Here: Navigating the Generative AI Revolution in Computing Education". In: *Proceedings of the 2023 Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '23. New York, NY, USA: Association for Computing Machinery, pp. 108–159. isbn: 979-8-4007-0405-5. doi: [10.1145/3623762.3633499](https://doi.org/10.1145/3623762.3633499).
- Prather, J., J. Leinonen, N. Kiesler, J. Gorson Benario, S. Lau, S. MacNeil, N. Norouzi, S. Opel, V. Pettit, L. Porter, B. N. Reeves, J. Savelka, D. H. Smith, S. Strickroth and D. Zingaro (Jan. 2025). "Beyond the Hype: A Comprehensive Review of Current Trends

- in Generative AI Research, Teaching Practices, and Tools”. In: *2024 Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE 2024. New York, NY, USA: Association for Computing Machinery, pp. 300–338. isbn: 979-8-4007-1208-1. doi: [10.1145/3689187.3709614](https://doi.org/10.1145/3689187.3709614).
- Prather, J., B. N. Reeves, J. Leinonen, S. MacNeil, A. S. Randrianasolo, B. A. Becker, B. Kimmel, J. Wright and B. Briggs (Aug. 2024). “The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers”. In: *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1*. Vol. 1. ICER ’24. New York, NY, USA: Association for Computing Machinery, pp. 469–486. isbn: 979-8-4007-0475-8. doi: [10.1145/3632620.3671116](https://doi.org/10.1145/3632620.3671116).
- Prechelt, L., G. Malpohl and M. Philippsen (Nov. 2002). “Finding Plagiarisms among a Set of Programs with JPlag”. In: *Journal of Universal Computer Science* 8.11, pp. 1016–1038. doi: [10.3217/jucs-008-11-1016](https://doi.org/10.3217/jucs-008-11-1016).
- Prince, M. (2004). “Does Active Learning Work? A Review of the Research”. In: *Journal of Engineering Education* 93.3, pp. 223–231. issn: 2168-9830. doi: [10.1002/j.2168-9830.2004.tb00809.x](https://doi.org/10.1002/j.2168-9830.2004.tb00809.x).
- Riddell, M., A. Ni and A. Cohan (Mar. 2024). *Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models*. doi: [10.48550/arXiv.2403.04811](https://doi.org/10.48550/arXiv.2403.04811). arXiv: [2403.04811 \[cs\]](https://arxiv.org/abs/2403.04811).
- Roberts, E. (Nov. 2002). “Strategies for Promoting Academic Integrity in CS Courses”. In: *32nd Annual Frontiers in Education*. Vol. 2, F3G–F3G. doi: [10.1109/FIE.2002.1158209](https://doi.org/10.1109/FIE.2002.1158209).
- Robins, A. (Mar. 2010). “Learning Edge Momentum: A New Account of Outcomes in CS1”. In: *Computer Science Education* 20.1, pp. 37–71. issn: 0899-3408. doi: [10.1080/08993401003612167](https://doi.org/10.1080/08993401003612167).
- Robins, A., J. Rountree and N. Rountree (June 2003). “Learning and Teaching Programming: A Review and Discussion”. In: *Computer Science Education* 13.2, pp. 137–172. issn: 0899-3408. doi: [10.1076/cs.ed.13.2.137.14200](https://doi.org/10.1076/cs.ed.13.2.137.14200).
- Roy, C. K., J. R. Cordy and R. Koschke (May 2009). “Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach”. In: *Science of Computer Programming* 74.7, pp. 470–495. issn: 0167-6423. doi: [10.1016/j.scico.2009.02.007](https://doi.org/10.1016/j.scico.2009.02.007).
- Sağlam, T. (2025). *Mitigating Automated Obfuscation Attacks on Software Plagiarism Detection Systems*. <https://publikationen.bibliothek.kit.edu/1000179018>. doi: [10.5445/IR/1000179018](https://doi.org/10.5445/IR/1000179018).
- Sağlam, T., M. Brödel, L. Schmid and S. Hahner (Apr. 2024). “Detecting Automatic Software Plagiarism via Token Sequence Normalization”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. New York, NY, USA: Association for Computing Machinery, pp. 1–13. isbn: 979-8-4007-0217-4. doi: [10.1145/3597503.3639192](https://doi.org/10.1145/3597503.3639192).
- Sağlam, T., S. Hahner, L. Schmid and E. Burger (May 2024). “Obfuscation-Resilient Software Plagiarism Detection with JPlag”. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ICSE-Companion ’24. New York, NY, USA: Association for Computing Machinery, pp. 264–265. isbn: 979-8-4007-0502-1. doi: [10.1145/3639478.3643074](https://doi.org/10.1145/3639478.3643074).
- Schleimer, S., D. S. Wilkerson and A. Aiken (June 2003). “Winnowing: Local Algorithms for Document Fingerprinting”. In: *Proceedings of the 2003 ACM SIGMOD International*

- Conference on Management of Data*. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, pp. 76–85. isbn: 978-1-58113-634-0. doi: [10.1145/872757.872770](https://doi.org/10.1145/872757.872770).
- Schneier, B. (Oct. 1995). *Applied Cryptography (2nd Edition): Protocols, Algorithms, and Source Code in C*. USA: John Wiley & Sons, Inc. isbn: 978-0-471-11709-4.
- Schrepp, M., A. Hinderks and J. Thomaschewski (2017). “Construction of a Benchmark for the User Experience Questionnaire (UEQ)”. In: *International Journal of Interactive Multimedia and Artificial Intelligence* 4.4, pp. 40–44. issn: 1989-1660. doi: [10.25968/opus-3397](https://doi.org/10.25968/opus-3397).
- Sheahen, D. and D. Joyner (Apr. 2016). “TAPS: A MOSS Extension for Detecting Software Plagiarism at Scale”. In: *Proceedings of the Third (2016) ACM Conference on Learning @ Scale*. L@S '16. New York, NY, USA: Association for Computing Machinery, pp. 285–288. isbn: 978-1-4503-3726-7. doi: [10.1145/2876034.2893435](https://doi.org/10.1145/2876034.2893435).
- Sheard, J., A. Carbone and M. Dick (Jan. 2003). “Determination of Factors Which Impact on IT Students’ Propensity to Cheat”. In: *Proceedings of the Fifth Australasian Conference on Computing Education - Volume 20*. Vol. 20. ACE '03. AUS: Australian Computer Society, Inc., pp. 119–126. isbn: 978-0-909925-98-7.
- Sheard, J. and M. Dick (June 2011). “Computing Student Practices of Cheating and Plagiarism: A Decade of Change”. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE '11. New York, NY, USA: Association for Computing Machinery, pp. 233–237. isbn: 978-1-4503-0697-3. doi: [10.1145/1999747.1999813](https://doi.org/10.1145/1999747.1999813).
- Sheard, J. and M. Dick (Jan. 2012). “Directions and Dimensions in Managing Cheating and Plagiarism of IT Students”. In: *Proceedings of the Fourteenth Australasian Computing Education Conference - Volume 123*. ACE '12. AUS: Australian Computer Society, Inc., pp. 177–186. isbn: 978-1-921770-04-3.
- Sheard, J., M. Dick, S. Markham, I. Macdonald and M. Walsh (June 2002). “Cheating and Plagiarism: Perceptions and Practices of First Year IT Students”. In: *ACM SIGCSE Bulletin* 34.3, pp. 183–187. issn: 0097-8418. doi: [10.1145/637610.544468](https://doi.org/10.1145/637610.544468).
- Sheard, J., Simon, M. Butler, K. Falkner, M. Morgan and A. Weerasinghe (June 2017). “Strategies for Maintaining Academic Integrity in First-Year Computing Courses”. In: *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. Bologna Italy: ACM, pp. 244–249. isbn: 978-1-4503-4704-4. doi: [10.1145/3059009.3059064](https://doi.org/10.1145/3059009.3059064).
- Shumailov, I., Z. Shumaylov, Y. Zhao, N. Papernot, R. Anderson and Y. Gal (July 2024). “AI Models Collapse When Trained on Recursively Generated Data”. In: *Nature* 631.8022, pp. 755–759. issn: 1476-4687. doi: [10.1038/s41586-024-07566-y](https://doi.org/10.1038/s41586-024-07566-y).
- Simões, A. and R. Queirós (2020). “On the Nature of Programming Exercises”. In: *First International Computer Programming Education Conference (ICPEC 2020)*. Ed. by R. Queirós, F. Portela, M. Pinto and A. Simões. Vol. 81. Open Access Series in Informatics (OASISs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 24:1–24:9. isbn: 978-3-95977-153-5. doi: [10.4230/OASISs.ICPEC.2020.24](https://doi.org/10.4230/OASISs.ICPEC.2020.24).
- Simon (Jan. 2017). “Designing Programming Assignments to Reduce the Likelihood of Cheating”. In: *Proceedings of the Nineteenth Australasian Computing Education Conference*. ACE '17. New York, NY, USA: Association for Computing Machinery, pp. 42–47. isbn: 978-1-4503-4823-2. doi: [10.1145/3013499.3013507](https://doi.org/10.1145/3013499.3013507).

- Simon, B. Cook, J. Sheard, A. Carbone and C. Johnson (Nov. 2013). “Academic Integrity: Differences between Computing Assessments and Essays”. In: *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. Koli Calling '13. New York, NY, USA: Association for Computing Machinery, pp. 23–32. isbn: 978-1-4503-2482-3. doi: [10.1145/2526968.2526971](https://doi.org/10.1145/2526968.2526971).
- Simon, B. Cook, J. Sheard, A. Carbone and C. Johnson (June 2014). “Student Perceptions of the Acceptability of Various Code-Writing Practices”. In: *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*. ITiCSE '14. New York, NY, USA: Association for Computing Machinery, pp. 105–110. isbn: 978-1-4503-2833-3. doi: [10.1145/2591708.2591755](https://doi.org/10.1145/2591708.2591755).
- Simon and J. Sheard (Feb. 2016). “Academic Integrity and Computing Assessments”. In: *Proceedings of the Australasian Computer Science Week Multiconference*. ACSW '16. New York, NY, USA: Association for Computing Machinery, pp. 1–8. isbn: 978-1-4503-4042-7. doi: [10.1145/2843043.2843060](https://doi.org/10.1145/2843043.2843060).
- Simon, J. Sheard, M. Morgan, A. Petersen, A. Settle, J. Sinclair, G. Cross and C. Riedesel (July 2016). “Negotiating the Maze of Academic Integrity in Computing Education”. In: *Proceedings of the 2016 ITiCSE Working Group Reports*. ITiCSE '16. New York, NY, USA: Association for Computing Machinery, pp. 57–80. isbn: 978-1-4503-4882-9. doi: [10.1145/3024906.3024910](https://doi.org/10.1145/3024906.3024910).
- Slobodkin, E. and A. Sadovnikov (Mar. 2023). *Towards a Dataset of Programming Contest Plagiarism in Java*. doi: [10.48550/arXiv.2303.10763](https://doi.org/10.48550/arXiv.2303.10763). arXiv: [2303.10763 \[cs\]](https://arxiv.org/abs/2303.10763).
- Stack Overflow (May 2024). *2024 Stack Overflow Developer Survey*. <https://survey.stackoverflow.co/2024/technology/>.
- Strijbol, N., C. Van Petegem, R. Maertens, B. Sels, C. Scholliers, P. Dawyndt and B. Mesuere (May 2023). “TESTed—An Educational Testing Framework with Language-Agnostic Test Suites for Programming Exercises”. In: *SoftwareX* 22, p. 101404. issn: 2352-7110. doi: [10.1016/j.softx.2023.101404](https://doi.org/10.1016/j.softx.2023.101404).
- Strozanski, P. (Nov. 2024). “Anti-Cheat for Programming Courses: Detecting AI-Generated Source Code”. MA thesis. Aalto University School of Science.
- Svetkin, A. (May 2024). *Testing LLMs on Solving Leetcode Problems* | HackerNoon.
- Tahaei, N. and D. C. Noelle (Aug. 2018). “Automated Plagiarism Detection for Computer Programming Exercises Based on Patterns of Resubmission”. In: *Proceedings of the 2018 ACM Conference on International Computing Education Research*. ICER '18. New York, NY, USA: Association for Computing Machinery, pp. 178–186. isbn: 978-1-4503-5628-2. doi: [10.1145/3230977.3231006](https://doi.org/10.1145/3230977.3231006).
- Tomita, M. (Aug. 1985). “An Efficient Context-Free Parsing Algorithm for Natural Languages”. In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'85. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 756–764. isbn: 978-0-934613-02-6.
- Ukkonen, E. (Sept. 1995). “On-Line Construction of Suffix Trees”. In: *Algorithmica* 14.3, pp. 249–260. issn: 1432-0541. doi: [10.1007/BF01206331](https://doi.org/10.1007/BF01206331).
- Valiente, G. (2021). “Tree Isomorphism”. In: *Algorithms on Trees and Graphs: With Python Code*. Ed. by G. Valiente. Cham: Springer International Publishing, pp. 113–180. isbn: 978-3-030-81885-2. doi: [10.1007/978-3-030-81885-2_4](https://doi.org/10.1007/978-3-030-81885-2_4).
- Van der Jeugt, F., R. Maertens, A. Steyaert, P. Verschaffelt, C. De Tender, P. Dawyndt and B. Mesuere (June 2022). “UMGAP: The Unipept MetaGenomics Analysis Pipeline”. In: *BMC Genomics* 23.1, p. 433. issn: 1471-2164. doi: [10.1186/s12864-022-08542-4](https://doi.org/10.1186/s12864-022-08542-4).

- Van Petegem, C., P. Dawyndt and B. Mesuere (June 2023). “Dodona: Learn to Code with a Virtual Co-teacher That Supports Active Learning”. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2. ITiCSE 2023*. New York, NY, USA: Association for Computing Machinery, p. 633. isbn: 979-8-4007-0139-9. doi: [10.1145/3587103.3594165](https://doi.org/10.1145/3587103.3594165).
- Van Petegem, C., L. Deconinck, D. Mourisse, R. Maertens, N. Strijbol, B. Dhoedt, B. De Wever, P. Dawyndt and B. Mesuere (Mar. 2023). “Pass/Fail Prediction in Programming Courses”. In: *Journal of Educational Computing Research* 61.1, pp. 68–95. issn: 0735-6331. doi: [10.1177/07356331221085595](https://doi.org/10.1177/07356331221085595).
- Van Petegem, C., K. Demeyere, R. Maertens, N. Strijbol, B. D. Wever, B. Mesuere and P. Dawyndt (Apr. 2024). *Mining Patterns in Syntax Trees to Automate Code Reviews of Student Solutions for Programming Exercises*. doi: [10.48550/arXiv.2405.01579](https://doi.org/10.48550/arXiv.2405.01579). arXiv: [2405.01579](https://arxiv.org/abs/2405.01579) [cs].
- Van Petegem, C., R. Maertens, N. Strijbol, J. Van Renterghem, F. Van der Jeugt, B. De Wever, P. Dawyndt and B. Mesuere (Dec. 2023). “Dodona: Learn to Code with a Virtual Co-Teacher That Supports Active Learning”. In: *SoftwareX* 24, p. 101578. issn: 2352-7110. doi: [10.1016/j.softx.2023.101578](https://doi.org/10.1016/j.softx.2023.101578).
- Visser, E. (Aug. 1997). *Scannerless Generalized-LR Parsing*. Tech. rep. P9707.
- Wager, E. (2014). “Defining and Responding to Plagiarism”. In: *Learned Publishing* 27.1, pp. 33–42. issn: 1741-4857. doi: [10.1087/20140105](https://doi.org/10.1087/20140105).
- Walker, H. M. (Mar. 1997). “Collaborative Learning: A Case Study for CS1 at Grinnell College and Austin”. In: *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '97. New York, NY, USA: Association for Computing Machinery, pp. 209–213. isbn: 978-0-89791-889-3. doi: [10.1145/268084.268164](https://doi.org/10.1145/268084.268164).
- Wang, Z., M. Shao, J. Bhandari, L. Mankali, R. Karri, O. Sinanoglu, M. Shafique and J. Knechtel (Apr. 2025). *VeriContaminated: Assessing LLM-Driven Verilog Coding for Data Contamination*. doi: [10.48550/arXiv.2503.13572](https://doi.org/10.48550/arXiv.2503.13572). arXiv: [2503.13572](https://arxiv.org/abs/2503.13572) [cs].
- Wang, Z., M. Shao, M. Nabeel, P. B. Roy, L. Mankali, J. Bhandari, R. Karri, O. Sinanoglu, M. Shafique and J. Knechtel (Apr. 2025). *VeriLeaky: Navigating IP Protection vs Utility in Fine-Tuning for LLM-Driven Verilog Coding*. doi: [10.48550/arXiv.2503.13116](https://doi.org/10.48550/arXiv.2503.13116). arXiv: [2503.13116](https://arxiv.org/abs/2503.13116) [cs].
- Watters, A. (Mar. 2025). *The Plagiarism Machine*. <https://2ndbreakfast.audreywatters.com/the-plagiarism-machine/>.
- Weber-Wulff, D. (Mar. 2019). “Plagiarism Detectors Are a Crutch, and a Problem”. In: *Nature* 567.7749, pp. 435–435. doi: [10.1038/d41586-019-00893-5](https://doi.org/10.1038/d41586-019-00893-5).
- Williams, L., C. McDowell, N. Nagappan, J. Fernald and L. Werner (Sept. 2003). “Building Pair Programming Knowledge through a Family of Experiments”. In: *2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings*. Pp. 143–152. doi: [10.1109/ISESE.2003.1237973](https://doi.org/10.1109/ISESE.2003.1237973).
- Wise, M. J. (Dec. 1993). *String Similarity via Greedy String Tiling and Running Karp-Rabin Matching*. Tech. rep. Australia: Department of Computer Science, University of Sydney.
- Xu, D. and Y. Tian (June 2015). “A Comprehensive Survey of Clustering Algorithms”. In: *Annals of Data Science* 2.2, pp. 165–193. issn: 2198-5812. doi: [10.1007/s40745-015-0040-1](https://doi.org/10.1007/s40745-015-0040-1).

- Xu, Z. and V. S. Sheng (Mar. 2024). “Detecting AI-Generated Code Assignments Using Perplexity of Large Language Models”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.21, pp. 23155–23162. issn: 2374-3468. doi: [10.1609/aaai.v38i21.30361](https://doi.org/10.1609/aaai.v38i21.30361).
- Yu, H., B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang and T. Xie (Feb. 2024). “CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. New York, NY, USA: Association for Computing Machinery, pp. 1–12. isbn: 979-8-4007-0217-4. doi: [10.1145/3597503.3623316](https://doi.org/10.1145/3597503.3623316).
- Yu, Z., Y. Wu, N. Zhang, C. Wang, Y. Vorobeychik and C. Xiao (July 2023). “CodeIPPrompt: Intellectual Property Infringement Assessment of Code Language Models”. In: *Proceedings of the 40th International Conference on Machine Learning*. PMLR, pp. 40373–40389.
- Zhao, J., K. Xia, Y. Fu and B. Cui (Nov. 2015). “An AST-based Code Plagiarism Detection Algorithm”. In: *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*, pp. 178–182. doi: [10.1109/BWCCA.2015.52](https://doi.org/10.1109/BWCCA.2015.52).
- Zhou, L., W. Schellaert, F. Martínez-Plumed, Y. Moros-Daval, C. Ferri and J. Hernández-Orallo (Oct. 2024). “Larger and More Instructable Language Models Become Less Reliable”. In: *Nature* 634.8032, pp. 61–68. issn: 1476-4687. doi: [10.1038/s41586-024-07930-y](https://doi.org/10.1038/s41586-024-07930-y).