# Segmentation of graphical user interface elements based on topological decomposition for GUI testing tasks

Artyom Abakumov [1[0000-0001-5784-7147]] and Sergey Eremeev[1[0000-0001-8482-1479]]

[1] Murom Institute of Vladimir State University, Russia
artem210966@yandex.ru
sv-eremeev@yandex.ru

**Abstract.** The paper addresses the issue of automating the testing process for graphical interfaces. It is shown that one of the main tasks in this area is segmentation of screen elements with further construction its internal structure. Emphasis is placed on the stability of the proposed method, regardless of changes in interface layout or the operating system employed. Our approach is based on decomposing a window screenshot into specific components that correspond to the elements of the original window and their hierarchy. We demonstrate the method's resilience to the resizing of objects within windows. The research was conducted using interface element segmentation for the QGIS geographic information system on both Windows and Ubuntu operating systems. Experimental results revealed high levels of accuracy, ranging from 94 to 100 percent, in extracting segmented areas within QGIS windows.

**Keywords:** topological analysis, GUI testing, segmentation.

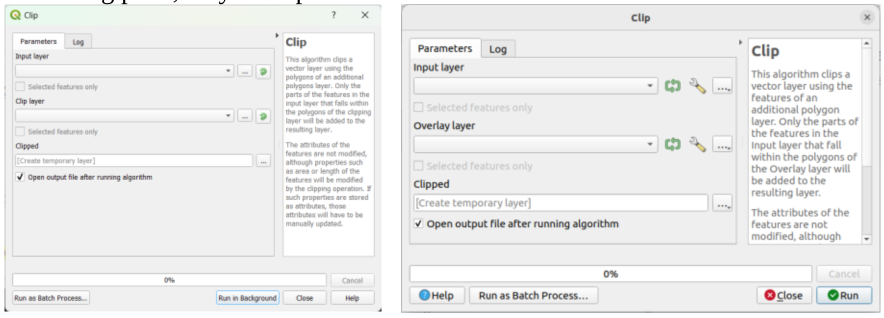
## 1 Introduction

The number of software programs continues to grow annually, and the quality of these applications is directly tied to the effectiveness of testing. While various types of automated testing exist to verify program functionality by executing code and validating its results, developers encounter difficulties when it comes to testing the graphical user interface (GUI) of applications.

There are testing utility that extracts meta-information about buttons, fields, and other interface elements. But, unlike regular code, the rendering of graphics in GUIs is reliant on the underlying operating system. Consequently, new frameworks and libraries for creating GUIs are constantly being developed and improved. Updates to these frameworks or changes in modules can break working this meta-extracting uptilts. In some cases, vendors intentionally restrict or complicate the extracting meta-information to enhance protection against reverse engineering. These and other scenarios necessitate the use of methods that do not rely on a deep understanding of the rendering approach or operating system, but instead interact with the interface through computer vision.

The task at hand involves identifying the desired interaction elements and validating their response to specific actions. For instance, one might need to click a button and ensure that a modal window opens. The typical solution in the computer vision approach involves performing a screenshot to locate the button for subsequent testing. Also, it is crucial to note that recognition errors are not permissible in this case, given the nature of the task.

Although it may appear that the problem could be solved by employing straightforward pixel-by-pixel comparison algorithms, in practice, program elements can exhibit slight variations in their rendering across different systems and machines. There are two primary challenges associated with this issue. Firstly, in diverse environments, windows may stretch, resulting in distorted displays of elements. Secondly, font glyphs can be rendered differently on various systems, and at times, a program may employ similar but distinct fonts on different operating systems. The disparity in GUI rendering is illustrated in Figure 1. Even if primitive methods attempt to ignore the differing parts, they have proven ineffective.


Clip

Parameters | Log

Input layer

☐ Selected features only

Clip layer

☐ Selected features only

Clipped

[Create temporary layer]

✓ Open output file after running algorithm

Clip

This algorithm clips a vector layer using the polygons of an additional polygons layer. Only the parts of the features in the input layer that falls within the polygons of the clipping layer will be added to the resulting layer.

The attributes of the features are not modified, although properties such as area or length of the features will be modified by the clipping operation. If such properties are stored as attributes, those attributes will have to be manually updated.

0% | Cancel

Run as Batch Process... | Run in Background | Close | Help

(a)

Clip

Parameters | Log

Input layer

☐ Selected features only

Overlay layer

☐ Selected features only

Clipped

[Create temporary layer]

✓ Open output file after running algorithm

Clip

This algorithm clips a vector layer using the features of an additional polygon layer. Only the parts of the features in the Input layer that fall within the polygons of the Overlay layer will be added to the resulting layer.

The attributes of the features are not modified, although

0% | Cancel

Help | Run as Batch Process... | Close | Run

(b)

Fig. 1. Different font and buttons in QGIS on Windows (a) and Ubuntu (b) systems.

These distortions not only add complexity to the testing process but also contribute to increased costs, highlighting the necessity for a method that can deliver consistent results while being resilient to such distortions. The primary focus of this work is to address this practical problem effectively.

In this paper, we present a new approach for decomposing application screenshots and propose a testing scheme built upon this approach.

## 2 Motivation

One of the main problems associated with open-source software is the issue of quality. Many developers contribute to open-source projects as a hobby and may not have the inclination to invest a significant amount of time in GUI testing. Consequently, there is a pressing need to develop a methodology that is straightforward and accessible to ordinary developers, without necessitating manual adjustments for different platforms.

To address this issue, it is logical to employ a method that not only compares images but also enables automatic highlighting of interactive GUI elements. This would allow users to select the desired interaction element without having to create a screenshot of it. One such method is the decomposition of images into topological features, which satisfies the requirements.

The technique for decomposing an image (screenshot) into components that identify high-frequency and low-frequency objects will be described in detail later. For now, it is sufficient to understand that this method enables the identification of all "disturbances" in the image, which are then represented as components. These components form a hierarchical structure, akin to a tree-like graph. Consequently, accessing an element becomes possible by traversing a component path in the Decomposition Tree, like selecting elements in XML or HTML schemas. The process of creating a test using this method is depicted in Figure 2. It is necessary to note that in a real scenario keyboard actions are also included in the testing process. But, since our goal is to test the GUI, we will not take keyboard actions into account.

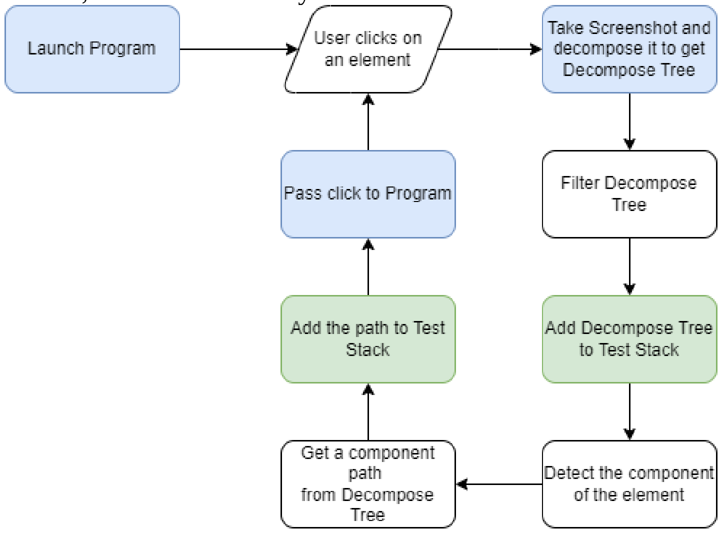

Launch Program
User clicks on an element
Take Screenshot and decompose it to get Decompose Tree
Pass click to Program
Filter Decompose Tree
Add the path to Test Stack
Add Decompose Tree to Test Stack
Get a component path from Decompose Tree
Detect the component of the element

Fig. 2. Test creation scheme.

The Decomposition Tree comprises components along with their associated metadata, including the start and the end times of existence, as well as decomposition matrices. Depending on the specific task, additional metrics can be incorporated as needed. A notable advantage of this method is that the size of the tree is significantly smaller than that of the source image.

During the execution of the created test (as depicted in Figure 3), the saved Test Stack is utilized.

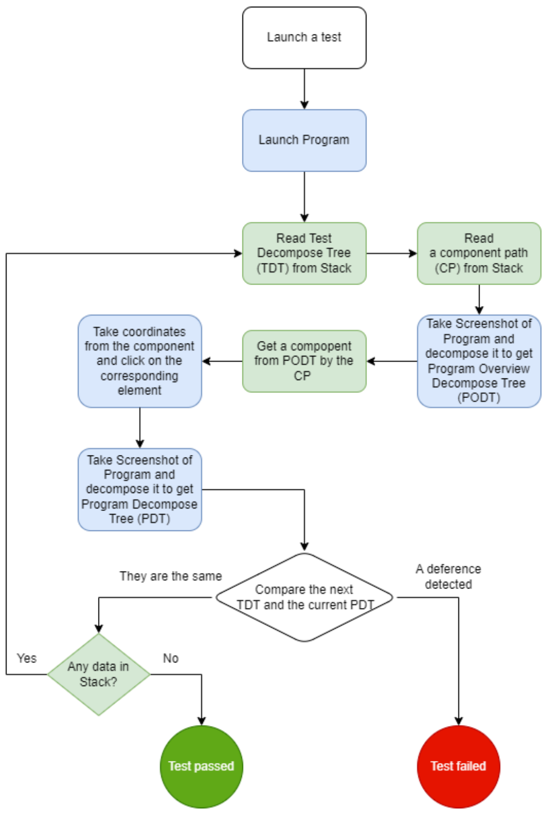

Launch a test
Launch Program
Read Test Decompose Tree (TDT) from Stack
Read a component path (CP) from Stack
Take Screenshot of Program and decompose it to get Program Overview Decompose Tree (PODT)
Get a compopent from PODT by the CP
Take coordinates from the component and click on the corresponding element
Take Screenshot of Program and decompose it to get Program Decompose Tree (PDT)
Compare the next TDT and the current PDT
They are the same
A deference detected
Any data in Stack?
Yes
No
Test passed
Test failed

Fig. 3. The scheme of the testing procedure (performs automatically).

The test will launch the program automatically and construct the decomposition based on a captured screenshot. This fully automated testing process, reliant on the decomposition technique, will be accomplished as a result.

## 3 Review

Testing utilities can be categorized into two main approaches based on their operational principles: black-box and white-box testing.

White-box testing involves assessing the internal structure and logic of the system rendering. It enables the determination of element coordinates, states, data, properties, and more in real-time. This approach offers high accuracy and simplicity. However, it requires the development of separate utilities for each graphical library and is highly dependent on the specific libraries and system being utilized, as mentioned previously.

Black-box testing involves testing a system without considering or having knowledge of its internal workings. It primarily focuses on the external behavior and functionality of the system. Implementations of this technique execute actions, such as moving the cursor or clicking, in a predefined order. Black-box testing allows for independence from specific technologies or systems but can potentially result in blind-running, where errors or issues may go unnoticed. For example, if there is a missed click, the testing program may continue the test without detecting the error. These situations are mitigated in the white-box approach.

If we narrow down the classification and deviate from abstract concepts, testing methods can be categorized based on the interaction with a test application. One fundamental method is the coordinate-based approach, where a tester specifies coordinates to which the cursor should be directed. However, it is important to consider that all coordinates may need to be adjusted in the event of distortions, such as changes in the size of elements. Autopy and PyAutoGUI are examples of utilities that facilitate such operations.

A more reliable method for testing is image recognition, which has been previously discussed. Examples of applications that employ this approach include Sikuli and Lackey.

Furthermore, the most reliable yet complex approach is the Accessibility method. This method pertains to white-box techniques that possess access to the internal elements of the system. An example of a popular solution in this domain is Pywinauto. However, it is worth noting that Pywinauto is exclusively compatible with Windows, rendering it unsuitable for cross-platform applications.

The underlying principle of these utilities predominantly relies on macros, which involve the repetition of pre-defined actions. One variation of this approach, referred to as keyword testing, is widely implemented in numerous popular tools like Katalon Studio and Jubula.

Webpages present the simplest case for testing due to their open code nature. Numerous programs are available specifically designed for web testing, with Selenium being a notable example.

Scientific papers addressing GUI testing started appearing as early as the 1990s, as mentioned by the authors in [1]. These researchers extensively publications on the subject and their achievements. The recent study [2] has demonstrated that despite significant advancements in recent years, numerous challenges persist without resolution and are anticipated to persist into the future. This emphasizes the significance of further research in this area.

Researchers propose various approaches to address the challenges of GUI testing. They explore different fields, such as security aspects in GUI testing [3]. Other researchers focus not only on the means of interacting with the program but also on the issue of test coverage. They explore methods to improve test coverage. In the study [4], the authors present existing tools for automated testing and the detection of untested elements and UI states. They propose a new method and its implementation for identifying potential states of elements, which involves computing all possible combinations of interactions with the application. However, the authors also acknowledge the limitations of the method, which further supports the claims made in the first review papers.

Scientific literature recognizes the significance of resource constraints in frequent software releases. The optimization and acceleration of testing in systems with realistic graphics are discussed in [5]. The paper describes existing approaches and provides a demonstration of the process using specific technologies.

Also, it is important to consider other systems such as smartphones. Mobile applications, which are prevalent in today's technological landscape, heavily rely on GUIs, making the discussed problem of GUI testing even more relevant in this context.

In [6], the authors highlight a common issue with testing tools that often rely on static GUI models, which may lack accuracy. To address this, the authors propose a novel approach where the GUI model is dynamically optimized during program execution. It is worth noting that many modern smartphone applications utilize web technologies, effectively functioning as miniature browsers with web pages. This presents an opportunity to leverage existing tools used for testing websites, thereby warranting further exploration of existing works in this particular area.

Methods for improving the testing of web pages based on their HTML are presented in [7]. The authors propose two versions of their method: one focuses on generating test cases for each web element, while the other explores different paths between web elements. These approaches aim to address the issue of insufficient test coverage and improve the overall testing process for web pages.

When considering papers closely related to the problem, it becomes evident that many of them are focused on mobile applications, thereby supporting our previous thesis. An intriguing work in this regard is [8], where the authors tackle the same task by detecting interactive elements and utilizing them for testing through the Accessibility methodology. They employ a trained computer model to identify interaction elements within mobile applications and describe their approach as comprehensive in their conclusions. However, it is worth noting that the use of machine learning confines the research to the realm of mobile applications exclusively. A similar issue, with a greater emphasis on training neural networks, is addressed in [9], also focusing on mobile applications. This trend is further exemplified in [10], where the authors employ machine learning techniques to enhance the existing methodology of random GUI testing.

In general, there is a noticeable trend in scientific papers towards the application of machine learning, particularly neural networks. For instance, some of the previously mentioned works utilize the popular YOLO v3 system, which not only segments but also classifies objects. However, it is important to note that despite the prevalence of neural networks in research, their practical adoption in existing testing utilities is not widespread. This is likely due to the significant overhead costs associated with resource-intensive classifiers. In practical terms, the usage of neural networks often involves cloud technologies, which introduces complexities and increases the cost of GUI testing.

If we consider the body of this chapter, it becomes apparent that there is a multitude of approaches for testing mobile applications, while there seems to be a relative lack of new ideas specifically focused on desktop applications. This observation further underscores the relevance of the problem we have chosen to address.

The decomposition method has demonstrated successful applications in satellite image segmentation [11] and object classification in images [12]. Therefore, it can be reasonably asserted that in simpler cases, it will yield results no less satisfactory than those achieved in the aforementioned articles.

## 4 Methodology

### 4.1 Decomposition of Interface Elements Based on Topological Features

The main principle employed in persistent homology is the systematic traversal of a point cloud and the construction of simplex complexes based on it. In our specific case, the image (matrix) functions as the point cloud, while the components, composed of pixels with distinct brightness values, act as the simplex complexes. Consequently, this enables us to establish a comprehensive framework for constructing components (Equation 1) and representing complexes (Equation 2).

$$M = \{C_1, C_2, \dots, C_n\} = F_M(I), \tag{1}$$

where $M$ is a set of output components, $C$ is a component, $F_M$is component creation functions, and $I$ is an input image.

$$C = \{b_1, b_2, \dots, b_k\}, \tag{2}$$

where $b$ is a pixel brightness, $C \in M$ and $k$ is a number of component points.

From Equation (2), we can derive various persistent features of a component. These include the brightness values of its start and end (Equations 3 and 4, respectively), the duration of its existence (Equation 5), and the square it occupies (Equation 6).

$$P_{start} = min\,\{b_1, b_2, \dots, b_k\} \tag{3}$$

$$P_{end} = max\,\{b_1, b_2, \dots, b_k\} \tag{4}$$

$$P_{length} = P_{end} - P_{start} \tag{5}$$

$$P_{square} = |C| \tag{6}$$

The features can be used to uniquely identify an individual component among the entire set.

In the paper [13], several approaches to constructing components are described. The simplest method involves analyzing pixels from top to bottom (from 255 brightness to 0) or from bottom to top. These two approaches we mark as base ones.

The base method connects pixels in a certain order. First, they are sorted by brightness and then added to the pad by X and Y indices. If there is another pixel next to the added pixel, they are joined into a component. If there is a component next to it, the pixel is joined to it. If a pixel can be attached to several components at once, one absorbs the other. The absorbed one does not grow any more.

The base methods only segments either bright or dark regions. Since interfaces can consist of various shades, it is necessary to perform two passes — one from top to bottom and another from bottom to top — to capture all interaction zones and accurately segment the components.

The bottom-up method iterates through all the pixels in ascending order of their brightness and analyzing the surrounding area. If there are no nearby components, a new component is created. If there is an existing component nearby, the current point merges with it, forming a larger component. If two components are found adjacent to each other, the older component assimilates the younger one. The younger component is designated as a child of the older component, resulting in a hierarchical, tree-like structure. One important feature of the decomposition is independent of the scale and, accordingly, of the resolution of the image.

The construction process for the window depicted in Figure 4 is illustrated in Figure 5 as an example.

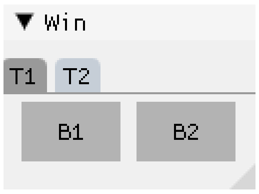


Fig. 4. A simple window with two tabs and two buttons.

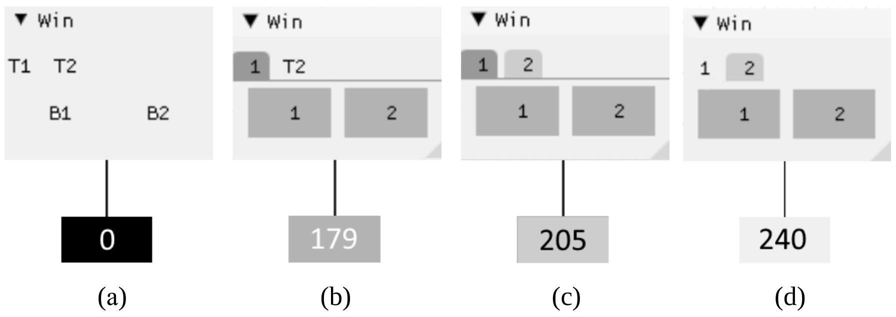


Fig. 5. Bottom-up construction. Pixels are stored in sorted order and added based on the increasing value of their brightness, b. Each displayed element represents a component.

The construction process begins by connecting all pixels with a brightness of zero, as shown in Figure 5(a). The adjacent pixels form separate components, representing the letters and the symbol in the top left corner. Then, the process continues by connecting pixels with a brightness of 1, and this progression continues for subsequent brightness levels. In Figure 5(b), at brightness level 179, the components representing the letter "B" absorb the pixels of the buttons. Finally, the same process occurs with the tab component labeled as "T1".

The example clearly demonstrates that some information is lost during the construction process. To recover this lost information, an additional pass from brightness 255 to 0 is required. However, this approach is inefficient. Therefore, it is advisable to further develop the algorithm to prevent information loss and aim for achieving accurate results in a single pass. This can help improve the efficiency and effectiveness of the component construction process.

One possible modification is to alter the order of pixels connections. Based on our experiments, the most promising approach is to connect points in pairs, considering the increasing difference in brightness, denoted as $D$, between them. The visualization of this method, along with the graph construction for the window depicted in Figure 4, is illustrated in Figure 6. We mark this version as the new method and named it as Radius-Based (RB).

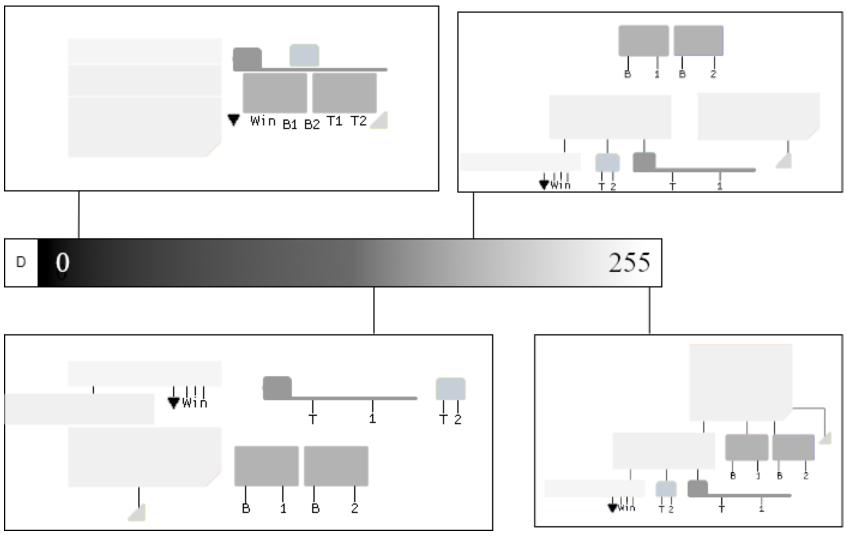


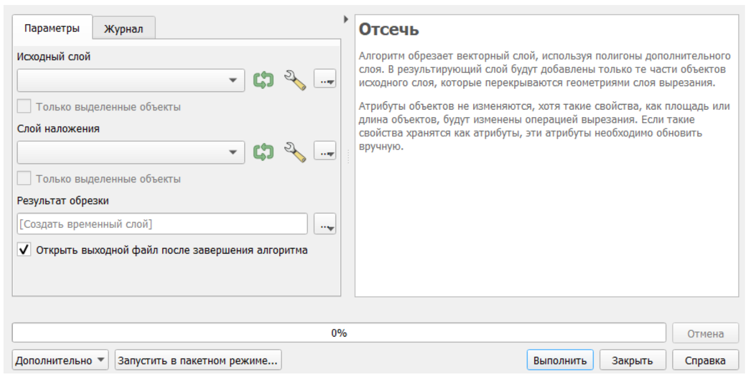
Fig. 6. Constructing components in the order of increasing difference ($D$) between pixels.

The RB method is predicated on connecting the closest points in terms of brightness. Initially, all pixels with zero brightness are connected, creating the first components. As the difference in brightness between other pixels becomes smaller than a threshold value, $D$, they are connected, with one component absorbing the others, forming a tree-like structure. In the considered case, there are the buttons, letters, and tabs in Figure 6. Their average brightness is close to 0. As the brightness approaches 170, the difference in brightness between the button component and the symbols "B", "1" and "2" exceeds the threshold $D$. As a result, the largest component (the button) absorbs the smaller ones (the symbols "B", "1", "2"), creating a hierarchy. The same process occurs with the remaining components. Closer to a brightness of 200, larger components start merging, forming an even more visible hierarchy. This process continues until all components form a tree.

To summarize, the new approach localizes the zones by the nearest brightness between points, which, given the specifics of the task, was the best fit. The results demonstrate that this modified approach achieves a more accurate segmentation. The one more advantage of the new approach is ability to use RGB colors because we sort pixels not by brightness, but by distance. RGB represents a vector (Red, Green, Blue), making it easy to calculate the distance. The bottom-up and up-bottom methods require grayscale pixels. In the paper, we will utilize the new approach to process the colored images.

### 4.2 Filtering Components for Obtaining Segmented Interface Elements

Certainly, filtering is an important step in the process. First and foremost, it is necessary to discard the text as it is not considered an interactive element. Let's consider Figure 7 for further analysis.



Параметры | Журнал

Исходный слой

☐ Только выделенные объекты

Слой наложения

☐ Только выделенные объекты

Результат обрезки

[Создать временный слой]

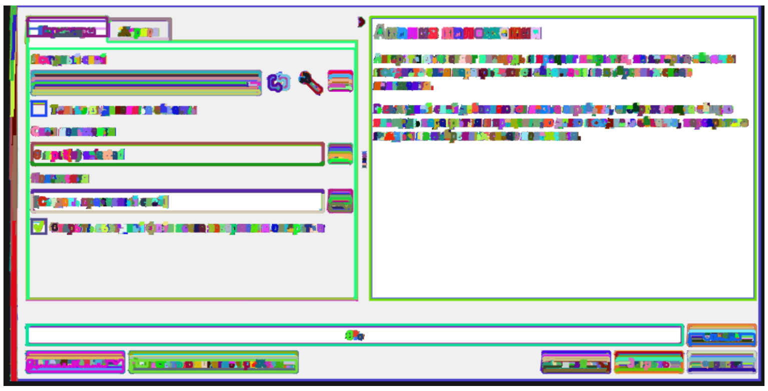
✓ Открыть выходной файл после завершения алгоритма

**Отсечь**

Алгоритм обрезает векторный слой, используя полигоны дополнительного слоя. В результирующий слой будут добавлены только те части объектов исходного слоя, которые перекрываются геометриями слоя вырезания.

Атрибуты объектов не изменяются, хотя такие свойства, как площадь или длина объектов, будут изменены операцией вырезания. Если такие свойства хранятся как атрибуты, эти атрибуты необходимо обновить вручную.

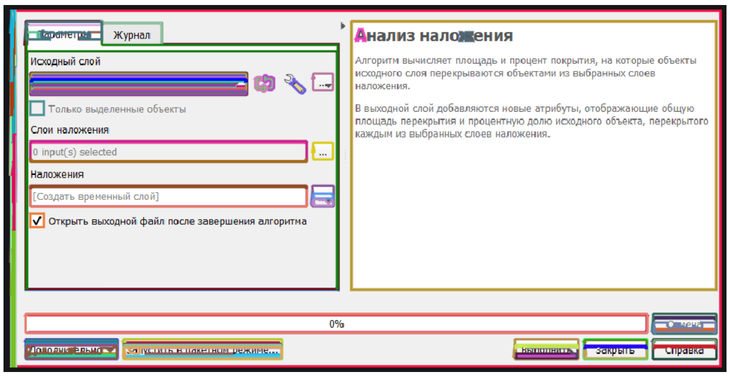
(a)



(b)

Fig. 7. The source "Clip" window (a) in QGIS and the components constructed based on it (b).

It is necessary to impose restrictions on certain features to filter the letters. Initially, we utilize the simplest one, the square of the component ($P_{square}$) which is calculated by formula (6). The resulting filtering outcome is presented in Figure 8. Some letters and intermediate components remain. To further refine the filtering process, we employ another feature, the existence length ($P_{length}$) of a component, calculated using formula (5). We will limit its maximum value. The result is displayed in Figure 9.

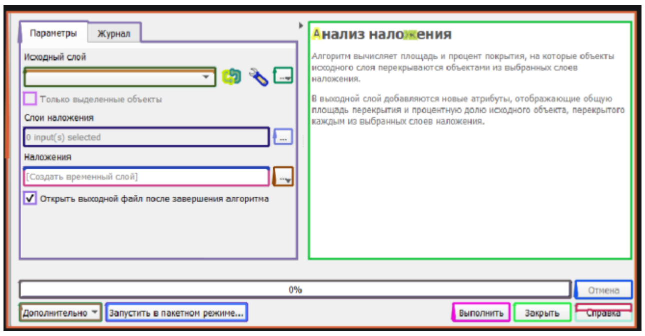


Анализ наложения

Алгоритм вычисляет площадь и процент покрытия, на которые объекты исходного слоя перекрываются объектами из выбранных слоев наложения.

В выходной слой добавляются новые атрибуты, отображающие общую площадь перекрытия и процентную долю исходного объекта, перекрытого каждым из выбранных слоев наложения.

Fig. 8. The segmentation result with $P_{square} >= 100$



Анализ наложения

Алгоритм вычисляет площадь и процент покрытия, на которые объекты исходного слоя перекрываются объектами из выбранных слоев наложения.

В выходной слой добавляются новые атрибуты, отображающие общую площадь перекрытия и процентную долю исходного объекта, перекрытого каждым из выбранных слоев наложения.

Fig. 9. The segmentation result with $P_{length} >= 10$ and $P_{square} >= 100$.

The artifacts have been successfully eliminated, and the segmentation has significantly improved. Although there are a few misclassifications where some letters and a portion of the button were mistakenly highlighted, however, they do not have a significant impact on the overall result.

Another way to perform filtering is by using the component path. This means that, like the testing process, specific elements can be indicated to be discarded and not considered when comparing the tree structure.

## 5 Results

Unfortunately, there are no public ready-to-use datasets for desktop GUI testing. The closest ones contain screenshots of mobile applications, which makes it difficult to verify the method. So, it was decided to compile its own set of test data.

So, there is a popular open-source solution called QGIS, which provides a graphical interface for working with geospatial data, such as GDAL library integration. However, in practice, bugs are often encountered, which makes many GUI windows useless. A variety of different windows can be utilized to test the decomposition method.

Typically, the following elements are used by the QGIS windows:

- Tab Panel: a container with other elements;
- Tab: an inactive tab;
- CheckBox: allows selecting either True or False;
- ComboBox: enables selecting from multiple items;
- InputBox: allows input of numbers and text;
- Button: a clickable button;
- MultiLine TextBox (Read only): an element for displaying text.

Visually, all these elements and their highlighting are shown in Figure 10.

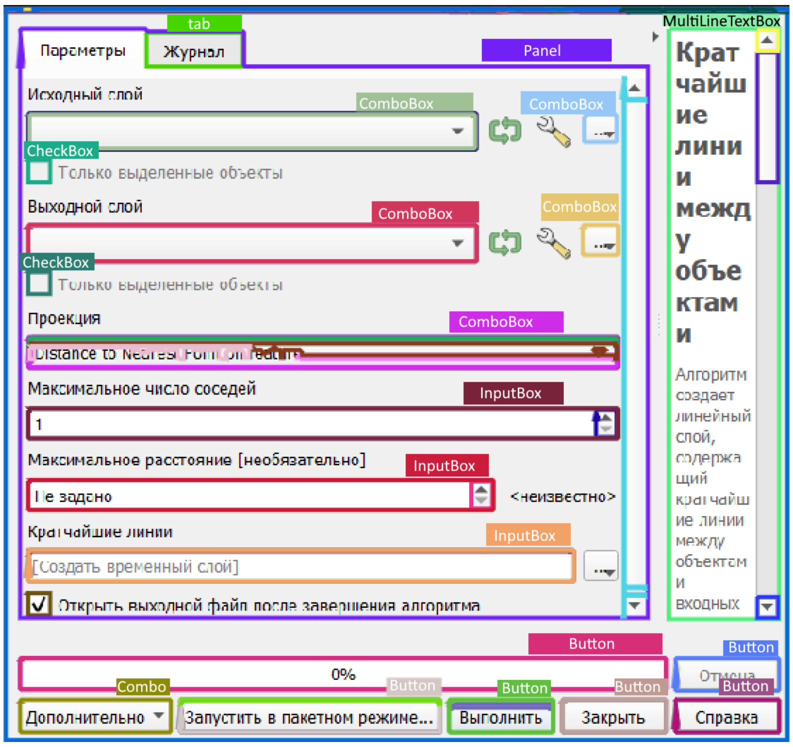

tab
Параметры
Журнал
Panel
MultiLineTextBox
Исходный слой
ComboBox
ComboBox
CheckBox
Только выделенные объекты
Выходной слой
ComboBox
ComboBox
CheckBox
Только выделенные объекты
Проекция
ComboBox
Distance to Nearest Point on feature
Максимальное число соседей
InputBox
1
Максимальное расстояние [необязательно]
InputBox
Не задано
<неизвестно>
Кратчайшие линии
InputBox
[Создать временный слой]
Открыть выходной файл после завершения алгоритма
Кратчайшие линии между объектами
Алгоритм создает линейный слой, содержащий кратчайшие линии между объектом и входных
Button
Button
0%
Отмена
Combo
Button
Button
Button
Button
Дополнительно
Запустить в пакетном режиме...
Выполнить
Закрыть
Справка

Fig. 10. The highlighted elements with labels.

Thirty QGIS windows were taken for testing purposes, and decomposition was performed on them using the criteria $P_{square} >= 100$ and $P_{length} >= 10$. The accuracy results, reflecting the found/total ratio, presented in Table 1. It is worth noting that sometimes buttons can be disable, and these cases accounted for a 6% decrease in accuracy. However, determining whether an inactive element should be identified remains a matter of debate.

**Table 1.** Accuracy results in Percentage, verified by a practicing GUI tester.

| Type | Accuracy, % |
|---|---|
| Tab | 98 |
| ComboBox | 94 |
| Tab Panel | 97 |
| InputBox | 100 |
| CheckBox | 100 |
| Button | 94 |
| MultiLine TextBox | 100 |

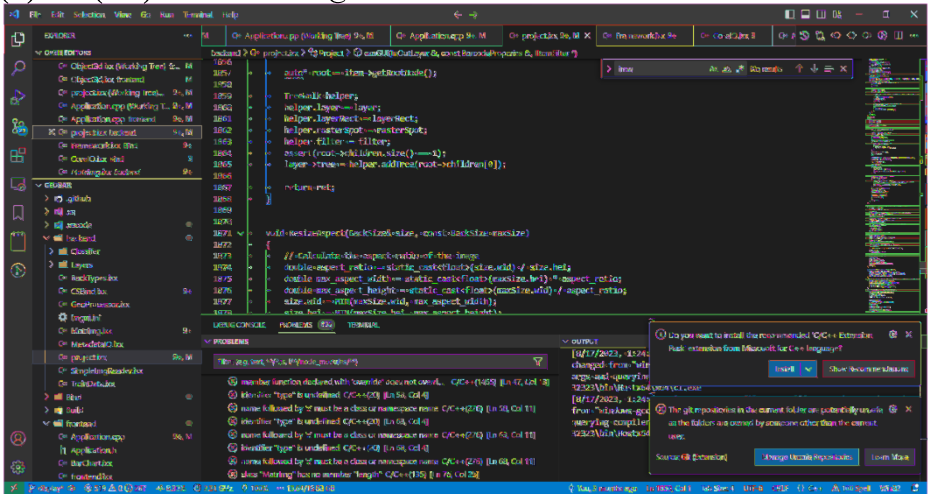
It is also important to note the errors. Table 2 indicates the number of letters that mistakenly entered the Decomposition Tree (i.e., remained after filtering), and the number of artifacts representing incomplete or incorrect segmentation of elements.

**Table 2.** Decomposition errors.

| Letters | Artifacts |
|---|---|
| 67 | 14 |

The letters only affect the size of the tree and do not pose problems for segmentation. Artifacts, on the other hand, can impact accuracy. To reduce these issues, it is necessary to refine the filtering process or improve the mechanism for constructing components in the case of artifacts.

Additionally, the method was tested on a variety of other applications. For example, we demonstrate the complete decomposition of a Visual Studio Code window in Figure 11. The library is written in C++. Without additional optimizations, it takes approximately ~1000 milliseconds to decompose a 1920x1080 screenshot on an Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz.



```
    auto* root = item->getRootNode();

    TreeWalk helper;
    helper.layer = layer;
    helper.layerRect = layerRect;
    helper.rasterSpot = rasterSpot;
    helper.filter = filter;
    assert(root->children.size() == 1);
    layer->tree = helper.addTree(root->children[0]);

    return ret;
}


void ResizeAspect(BackSize& size, const BackSize maxSize)
{
    // Calculate the aspect ratio of the image
    double aspect_ratio = static_cast<float>(size.wid) / size.hei;
    double max_aspect_width = static_cast<float>(maxSize.hei) * aspect_ratio;
    double max_aspect_height = static_cast<float>(maxSize.wid) / aspect_ratio;
    size.wid = MIN(maxSize.wid, max_aspect_width);
```

```
member function declared with 'override' does not overri... C/C++(1455) [Ln 47, Col 13]
identifier "type" is undefined C/C++(20) [Ln 58, Col 4]
name followed by '::' must be a class or namespace name C/C++(276) [Ln 58, Col 11]
identifier "type" is undefined C/C++(20) [Ln 63, Col 4]
name followed by '::' must be a class or namespace name C/C++(276) [Ln 63, Col 11]
identifier "type" is undefined C/C++(20) [Ln 68, Col 4]
name followed by '::' must be a class or namespace name C/C++(276) [Ln 68, Col 11]
class "Matring" has no member "length" C/C++(135) [Ln 76, Col 28]
```

Do you want to install the recommended 'C/C++ Extension Pack' extension from Microsoft for C++ language?

The git repositories in the current folder are potentially unsafe as the folders are owned by someone other than the current user.

Fig. 11. The VS Code windows with highlighted elements.

## 6 Conclusions

The study results highlight the importance of further refining the filtering mechanism and utilizing more reliable indicators. However, apart from that, the segmentation results are promising, with an average accuracy exceeding 95 "The decomposition library is available on GitHub: https://github.com/Noremos/SatHomology.

In conclusion, the topological decomposition method has proven to be effective in detecting interactive zones. It provides the advantage of disregarding potential distortions, such as element stretching and variations in the rendering of glyphs and interaction elements.

Acknowledgements: This study was supported by the Russian Science Foundation, project no. 23-21-10064, https://rscf.ru/en/project/23-21-10064/.

## References

1. Banerjee, I., Nguyen, B., Garousi V.: Graphical user interface (GUI) testing: Systematic mapping and repository. Information and Software Technology, 55(10), 1679-1694 (2013).
2. Nass, M., Alégroth E., Feldt R.:Why many challenges with GUI test automation (will) remain. Information and Software Technology, 138 (2021)
3. Mironov, S.: Technologies of security control of automated systems based on structural and behavioral testing of software. Cybernetics and programming, 5, 158-172, (2015).
4. Vartanov, S., Gerasimov, A, Ermakov, M., Kutz, D., Novikov, A.: Dynamic analysis of programs with graphical user interface based on symbolic execution. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), 29(1), 149-166 (2017).
5. Denisov, E., Voloboy, A., Birukov, E..: Technologies for automatic testing of a software package for realistic computer graphics. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), 32(1), 71-88 (2020).
6. Gu, T.: Practical GUI Testing of Android Applications Via Model Abstraction and Refinement. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 269-280 (2019). Montreal, QC, Canada.
7. Medhat, M., Saad, M.: Enhancing the Automation of GUI Testing. In Proceedings of the 8th International Conference on Software and Information Engineering (ICSIE '19). Association for Computing Machinery, pp. 66–70 (2019). New York, NY, USA.
8. Zhang, X., Lilian de Greef, Swearngin, A., White, S: Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. arXiv (2021)
9. Xue, F., Wu, F., Zhang, T.: Visual Identification of Mobile App GUI Elements for Automated Robotic Testing. Computational Intelligence and Neuroscience, 1, 1687-5265 (2022).
10. Thomas, D., White, G., Guy J.: Improving random GUI testing with image-based widget detection. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019), 307–317 (2019). New York, NY, USA.
11. Eremeev, S., Abakumov, A., Andrianov, D., Shirabakina, T.: Vectorization Method of Satellite Images Based on Their Decomposition by Topological Features. Informatics and Automation, 22(1), 110-145 (2023).

12. Eremeev S., Abakumov A.: Classification of objects in images with distortions based on a two-stage topological analysis. Scientific and Technical Journal of Information Technologies, Mechanics and Optics,.22(1), 82–92 (2022).
13. Eremeev, S., Abakumov, A., Andrianov, D., Titov, D.: Image decomposition method by topological features. Computer Optics, 46(6), 939-947 (2022).