Case-Based System of User Interface Reverse Engineering

Pavel Myznikov
Novosibrisk State University
1 Pirogova str., Novosibirsk
630090, Russia
miznikov72@gmail.com

Abstract

The paper is devoted to the implementation of case-based reasoning in reverse engineering of graphical user interfaces. In particular, web interfaces are considered. We suggest automating HTML/CSS markup building with aggregation of code samples from previous cases. The proposal is based on the hypothesis that analogy is employed to conceptualize HTML markup construction. The article considers the original theory of building an image structure, the modification of the case-based reasoning approach and the results of practical experiments.

Introduction

Web-technology is one of the most developed areas of modern computer science. It is used not only in website development but also as an important node of any IT-infrastructure. Consequently, the whole information technology industry uses web-development in one form or another.

Nevertheless, automation of web-development itself remains incomplete. One of the key tasks – creation of html/css markups – does not have an extensive solution yet. Some properties of this process prevents using classical methods of automation. These properties are: non-formalized requirements for a layout, variability of industrial standards for code writing, and, finally, cross-browser and cross-platform compatibility.

The impact of automation of html/css markups building is not limited to speeding up a process of web-applications development. It also makes the development more flexible allowing to scale results, verify hypotheses and help testing applications.

We can expand the results of the work to a more general domain: any kind of technologies where markup languages are used for building GUI.

A methodological basis of the work is the Case-Based Reasoning (CBR) approach (Kolodner 1992). Briefly describing its essence, one can say it is a way of solving problems with adaptation of similar problems from the past to a current situation. This approach has been chosen due to the hypothesis that formalization of images (which is, in fact, creation of html markup) using transductive reasoning, or reasoning by analogy, produces a result, the most similar to what a human does. Also, such a scheme can solve the problems with the automation described above.

The state of the art solutions offer to generate code without human involvement. They work as black boxes, where the final result totally depends on the collected dataset. The CBR approach is to fill this gap with the acquisition of experts' knowledge in case storage. The ultimate role of CBR, therefore, is to help specialists to formalize their knowledge in a relatively small dataset and then automatically combine complex interfaces in such a way as if the specialists did it manually.

Related work

To fill a gap in knowledge for readers who are not familiar with the field, we overview two types of related work. The first one is the papers about interfaces reverse engineering: they are the benchmarks which we compare our approach with. The second one is an outline of CBR. Since this is a core of the given approach, this information is crucial for understanding the paper.

Interfaces reverse engineering

We consider two types of interfaces that are typical objects of the reverse engineering task: mobile interfaces and web interfaces.

Mobile interfaces Mobile development is supposed to be a more common area for interfaces reengineering than web. It can be explained by the fact that mobile interfaces are usually relatively simplier and more unified, while web pages have a great variance of different layouts. Here we point at a few mobile interface reverse engineering researches that we consider as robust baselines in terms of comparing OCR and code generation parts.

The framework of (Nguyen and Csallner 2015), REMAUI, is conceptually similar to our approach, especially in the OCR part. REMAUI is also faced with the problem of false-positive candidates and contains a merging step while parsing an input. In both questions, they use a set of heuristics specific to a mobile domain. Surely, there is a difference. Firstly, our OCR algorithm is mainly focused on image detection rather than text detection. Secondly, we suggest another model of hierarchy building. However, we can state that the idea of filtering candidates in the OCR stage was added to our framework after familiarization with their paper.

Based on REMAUI, (Natarajan and Csallner 2018) developed a tool – P2A – for generating a code for *animated* mobile application. This is the next step of the problem: not only to recognize a layout but also to find a relation between two similar interfaces screenshot. What is important is that this tool implies an interaction with a user while performing a task as well as our approach. The difference is that we suggest a long-term setting of the system in the beginning but the absence of it later, while P2A offers a user's involvement in each session of image processing.

The work of (Chen and Burrell 2001) is devoted to solving a very specific subtask of mobile interface reverse engineering: automatic conversion of an Android application into an iOS application and vice versa based on interface screenshots. Having such restricted input and output of the task, the authors were able to implement a more powerful model. Namely, they reduced the task to component detection and component classification, where for the second part convolutional neural networks were used.

Web interfaces Further, we would like to focus on web reverse engineering, because in the case of solving this task, a possible outcome can be more considerable, as web technologies are applied in an extremely wide range of domains. Researches in this field are not so numerous, however.

Asiroglu in (Asiroglu et al. 2019) presented a quite unusual reverse engineering case. Their tool accepts a hand-drawn sketch and generates an HTML document using a deep convolutional neural network. Such an application seems very helpful in the industry. Unfortunately, it cannot be used in solving the given research problem, because in our case, we should detect exact features of the interface, e.g. color, font, size, etc., while that tool perceives a common idea of an interface.

The most recent successful case of solving the problem in the industry is the result of UIzard Technologies company (Beltramelli 2017) with the title "pix2code". The authors note the similarity of the task of generating a code based on an interface screenshot with the task of generating a text in a natural language based on an image. Consequently, they use a similar method: the basis of the solution is a cascade of long-short term memory recurrent neural networks. At the moment, this project is only on a proof-of-concept stage, so it is too early to make valid conclusions about the viability of the idea.

As you can see above, there are many points of view on the interface reverse engineering problem. Each of them is suitable for specific conditions and goals. The contribution of the given paper is building a reverse engineering framework more as an expert system rather than an automation tool. It means that compared with alternatives, it can eventually generate less accurate results in terms of pixel-to-pixel similarity, but an important thing is that specialists can contribute their knowledge in a case storage, thereby managing a style or methodology that they expect to see in the final result.

Case-Based Reasoning

As mentioned above, case-based reasoning is a core of the approach. This section outlines basics of CBR.

The origin of CBR is the work of Roger Schank (Schank 1982). His idea is representing knowledge as memory organization packets (MOPs) that hold results of a person's experience. When a person solves a problem, he or she uses MOPs to reapply a previously successful solution scheme in a new similar context. This approach contradicts the rule-based approach, where a person uses predefined scripts for all cases. You can find more information in (Watson and Marir 1994).

A lot of works devoted to this approach were written in the 1990s (Kolodner 1992; Watson and Marir 1994; Aamodt and Plaza 1994; Ram and Santamaría 1997). In the 2000s case-based reasoning was implemented in different domains: medicine (Gómez-Vallejo et al. 2016; Delir Haghighi et al. 2013), business (Chou 2009; Chang, Liu, and Lai 2008; Carrascosa et al. 2008), finance (Sartori, Mazzucchelli, and Gregorio 2016; Sun et al. 2014), government (Lary et al. 2016), education (Zeyen, Müller, and Bergmann 2017), information technologies (De Renzis et al. 2016; Navarro-Cáceres et al. 2018; He 2013), systems design (Shen et al. 2017), geosciences (Lary et al. 2016).

In short, CBR is a method of solving a problem by adaptation of solutions to similar problems in the past to a current situation.

The core concept in CBR is a case. A case is a threesome of elements:

- **Problem** is a state of the world when the case occurs.
- **Solution** is a suitable answer for the problem in the given context.
- **Outcome** is a state of the world after the problem is solved.

The process of finding the solution is called a CBR-cycle that consists of 4 stages (Kolodner 1992):

1. **Retrieve.** The best matching case is extracted from the case base.
2. **Reuse.** The solution to the case retrieved is adapted to solve the current problem.
3. **Revise.** The adapted solution is tested: if it is not satisfactory, then either additional cases are retrieved, or the retrieved solution must be adapted again.
4. **Retain.** The case with the give solution is added to the case base.

Task definition

When defining the task, we aim to reproduce the way that a human performs HTML markup building. All criteria, con-

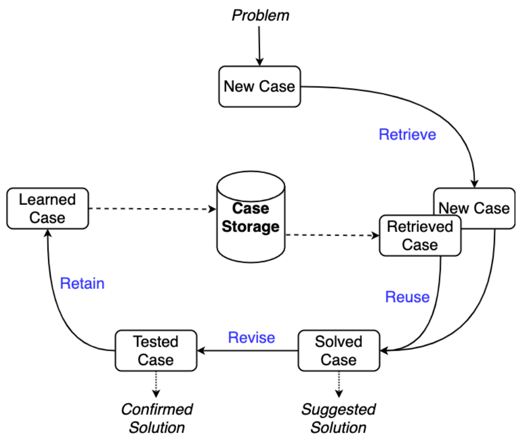

Figure 1: CBR cycle according to Aamodt and Plaza (Aamodt and Plaza 1994)

strains and conditions given below are based on how the industry operates. For more information, see (Ali 2017).

The following life-cycle of interaction with the system is suggested. In the very beginning, an expert (or a group of experts) populates a case storage with code templates to setup the system. Or, they can use a ready case storage populated with someone else. This step is required to be done only once. Then, the system is used to get a design of interface as an input and generate code automatically. At this point, we don't expect an absolutely accurate result. A specialist, in the end, makes final improvements in the code, or uses it as a basis for more high-level tasks.

Consequently, the main requirements to an image processing part are that the algorithm must recognize an image structure, find as many features (color, type, margins, etc.) of objects as possible, and generate the code implementing the structure recognized. Wherein, it is not necessary to

- recognize *all* features
- recreate the picture pixel-to-pixel

The introduction and related work sections contain conditions that should be considered in solving the task. It is necessary to make a set of requirements for the system such that it would be possible to get positive results in these conditions.

- **Condition 1. Non-formalized requirements.** An image itself does not contain complete information about what should be a final HTML-markup. Therefore, it must be available to include additional knowledge about the requirements for the system.
- **Condition 2. Variability of industrial standards to writing the code.** There are a few methodologies of HTML-markups (adaptive, BEM (block-element-modificator), bootstrap and so on). Moreover, as a rule, each developer's team has its own standards. Thus, the system must generate code in several styles and standards.
- **Condition 3. Cross-browser and cross-platform compatibility.** HTML-code must be not only valid, but also identically processed in all modern browsers and operating systems.

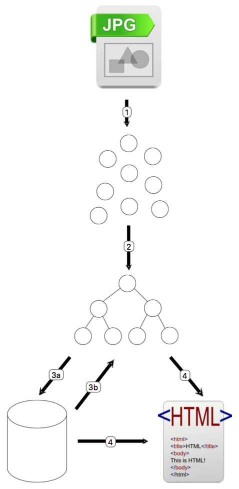

Figure 2: Scheme of generation HTML code based on a bitmap. 1. Extraction elements. 2. Building a hierarchy of elements. 3a and 3b. Transforming a hierarchy to html document using a case storage. 4. Assembling final files.

In addition, we set input constraints on the current stage. An image must not include

- gradient background
- animation elements
- popup elements

In the future, we intend to make an algorithm working beyond these conditions.

Architecture

The basis of the architecture is an attempt to recreate the process of writing code by a human. The hypothesis is, a human uses transductive reasoning when formalizing visual information: one extracts structures of different levels from an image and describes this structure by analogy with one's own or others' previous experience.

In particular, such a situation is observed in writing HTML-code. Meeting an unknown layout pattern, a specialist finds in the literature, how to implement it on HTML. Further, referrals to directories are becoming less and less, but the principle remains the same: facing another pattern, a human extracts from memory a suitable example and adapts it to a current situation.

The process is implemented in the following way (Fig. 2):

1. The elements are extracted from the image (text, pictures, blocks, etc.).
2. The extracted images are combined in a tree-like structure according to a special algorithm.
3. A prefix tree traversal is performed; on each iteration, there is a request to the case base to find a suitable HTML description of an architectural pattern in the tree node.

4. Artifacts received on the previous step are saved in files.

The next sections are devoted to the description of these steps.

Image structure building

The goal of this step is to receive a so-called structure of a bitmap, i.e. a mathematical object describing a mutual arrangement of image parts. In (Myznikov 2018), we described an approach to solving this problem. We detect borders of objects (embedded images, texts, background) and then use a greedy algorithm to build a hierarchy of these objects (see algorithms 1 and 2). The hierarchy then is processed with the CBR cycle (see the next section).

Algorithm 1 Algorithm of hierarchy construction

```
Require: $nodes$ – set of elements ordered by horizontal and vertical axes of the top left coordinate
Ensure: $|nodes| == 1$ and $nodes_1$ is a root node of the tree containing all elements from the origin set
 1: while $|nodes| > 1$ do
 2:     for $orient \leftarrow$ [HOR,VERT] do
 3:         $i \leftarrow 0$
 4:         while $i < |nodes|$ do
 5:             $suitNode \leftarrow findNode(nodes_i, orient)$
 6:             if $suitNode$ is not NULL then
 7:                 if $node_i$ is composite then
 8:                     $nodes_i.addChild(suitNode)$
 9:                 else
10:                     $newNode \leftarrow$ new $Node$
11:                     $newNode.addChild(nodes_i)$
12:                     $newNode.addChild(suitNode)$
13:                     $newNode.orientation \leftarrow orient$
14:                     $nodes_i \leftarrow newNode$
15:                 $nodes \leftarrow nodes \setminus suitNode$
16:                 $i \leftarrow i + 1$
```

Development of the CBR-system

Remember that generation of code is performed with the prefix tree traversal, where a tree is an image structure. On each iteration, the CBR-cycle runs. We need to define the problem, solution, and outcome in the context of the task. In other words, to describe a case format.

- **Problem** is a features vector of a node and its children.
- **Solution** is templates of HTML and CSS code, that implement markup of the structure described.
- **Outcome** is HTML/CSS code generated from the template applied to a specific context.

Thus, when each node of a tree is processed, a “problem” is formed: description of the current element and elements on a lower level in an image structure. In storage, there are cases in which problems are typical cases of elements layout, and solutions are typical ways of markup. The task of a CBR-cycle is to get HTML/CSS code to build a markup of a current structure using the solution stored. Let us describe the cycle in detail. As a notation, denote the case processed as *new case*. Further terms will be included as they appear.

Algorithm 2 Search for a suitable node (*findNode*)

```
Require: $node_i$ – a node for that a suitable node is being found, $node^{x_1}, node^{y_1}$ – a left top coordinate, $node^{x_2}, node^{y_2}$ – a right bottom coordinate
Require: $orientation$ – a direction of joining nodes
Ensure: $node_j$ is a node demanded or NULL if there is no such a node
 1: if $orientation$ is HOR then
 2:     $node_j \leftarrow$ the next node to the right of the $node_i$
 3: else if $orientation$ is VERT then
 4:     $node_j \leftarrow$ the next node to the bottom of the $node_i$
 5: $x^1 \leftarrow \min(node_i^{x^1}, node_j^{x^1})$
 6: $y^1 \leftarrow \min(node_i^{y^1}, node_j^{y^1})$
 7: $x^2 \leftarrow \max(node_i^{x^2}, node_j^{x^2})$
 8: $y^2 \leftarrow \max(node_i^{y^2}, node_j^{y^2}))$
 9: $R \leftarrow$ rectangle $(x^1, y^1)$ $(x^2, y^2)$
10: for all $node_k \in nodes$ do
11:     if $node_k \cap R \neq \emptyset$ then
12:         $node_j \leftarrow$ NULL
```

Retrieval phase *A new case* has a problem but no solution and outcome. The task of this stage is to retrieve cases, which are the most similar to the problem of a *new case*. In general, it is a task of n-class classification, where n is the number of cases in storage. On the proof-of-concept stage, we choose the method of k-nearest neighbors with Euclid distance. For this purpose, we vectorize and normalize problems. This means that first, all categorial features are transformed to a numerical type, second, absolute values are replaced with relative ones by the formula: $x_i = \frac{x_i}{x_{\max} - x_{\min}}$. Then, calculate a distance between the given vector and vectors in a case base $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$ and select the case, where distance is minimal. In terms of CBR, this case is called *retrieved*.

Adaptation phase The adaptation stage is performed with an algorithm, receiving a problem of a *new case* and a solution of a *retrieved case* and returning an outcome. A case containing a problem of a *new case*, solution of a *retrieved case* and the generated outcome is a *solved case*, and the solution is called *suggested*.

In the current work, a derivative type of adaptation is suggested, that means an old solution is to be regenerated in new conditions. Since a solution is a code template, a result is built with using templating language. The difficulty is that the results of different cases may conflict with each other during the processing. It is especially true for conflicts in CSS code: situations when different results describe the same class differently. That is why conflict resolution is an important part of the adaptation stage.

The base of conflict resolution is a resolving of three cases: 1) absence of conflict; 2) a complete conflict, that is classes describe incompatible styles; 3) one class is a special case of another. The following strategies are applied accordingly: 1) the code is saved as is; 2) different classes with own styles are created, and appropriate replacements in an

Table 1: Differences of local and global storages

Local storage	Global storage
Stored in RAM	Stored in external memory
Destroyed after a document is processed	Exists regardless of documents processing
Filled during a document processing	Filled with a special procedure before documents processing
Cases have a higher priority when extracting	Cases have a lower priority when extracting

HTML-document are made; 3) a new class is added to a styles set, which is a special case, and appropriate inserts in an HTML-document are made.

Evaluation phase In the classic CBR, an evaluation step serves as validation of the suggested result. A case that passed the test is called *tested*, and the solution is *approved*. Regarding this work, the task of evaluation of the quality of HTML/CSS code deserves a separate study. In the current state of the research, this step is skipped, and validation of the result is resolved for the whole document and not on each iteration of a CBR-cycle.

Updating phase The goal of the last step of the cycle is to save an approved solution in case storage so that it can be reused in the future. In the current work, this step differs from the classic approach. Unlike using a centralized case storage, the CBR-system developed maintains a "global" and few "local" case storages (see table 1). The global case storage is used for all images. The local ones are used only in processing a specific document. We can say that a "local" storage is a context of a document: it contains the results of solved problems during its processing.

Specifically, a tested case is always saved only into a local storage. Wherein, on a retrieval step, a search is performed in both global and local repositories, where local ones have priority. This approach has several advantages:

1. A risk of different code generation for the same blocks of an image is decreased. The system may process identical blocks differently, because noise on an image can affect retrieval of a suitable case. At the same time, the size limitation of a local storage prevents generating redundant solutions.
2. System performance is increased:
   (a) a global storage is located in external memory, while a local one is in RAM; as a result, the number of requests to a hard disk reduces
   (b) graphical user interfaces have a property to contain a lot of similar blocks; when processing the next block, a search in a relatively small local storage, which already contains a suitable case, is performed much faster than a repeating search in a global storage; also, an adaptation stage requires less resources than initial adaptation of an "unprepared" case.
3. There is no "clogging" of a global storage with specific cases, which, first, positively influences the size of global storage, and second, it reduces a level of overfitting of the system.

In other words, a global storage serves as a source of cases that have solutions of code generation of a typical layout. Then, local storage is used for accurate and rapid adaptation of these solutions in the context of a specific document.

Case storage population The case storage is populated by experts with a share of automation. Namely, a set of cases with their problem parts is built automatically, and corresponding solution parts are filled by specialists. The following are the steps for generating cases:

1. A list of real websites is collected.
2. DOM-trees of random pages are saved.
3. Trees are split by sub-trees with prefix traversal of a whole tree and selecting each node with its children.
4. A given set of sub-trees is clustered with DBSCAN algorithm, when tree-edit distance is used to define the similarity between objects. (See tree-edit distance overview in the section below.)
5. From each cluster, a random object is selected. The selected objects form the case storage, where each object is a problem part of a single case.

While managing parameters *eps* and *min samples* of DBSCAN, one can control the size of a case storage. The bigger the case storage, the more accurate the system, but there is more work for experts. One must find a right balance between these two criteria to set up the system adequately to the existing conditions. Then, when the set of cases with their problems is built, experts can populate it with solutions - HTML/CSS codes.

Results and further work

Before the experiment evaluation, let us illustrate how the system works in practice.

As an example, let us consider a page of Novosibirsk State University site.

As an output of the first stage of image processing, elements were extracted and grouped into nodes (Fig. 3). Then, the algorithm (described earlier in the paper) created a tree-like structure (Fig. 4). We estimate that the result is of good quality, because despite of some mistakes (wrong font was selected, some pictures were missing, tiny errors in size existed), the system managed to solve the main task: to recreate a structure of elements and generate human-like code (Fig. 5).

Evaluation method

The question of evaluation is open. What should we compare to estimate the result? One option is to compare images: an initial screenshot and a screenshot of the interface generated by the system. It is probably the most obvious way but as we stated in the task definition part, we do not aim to recreate an

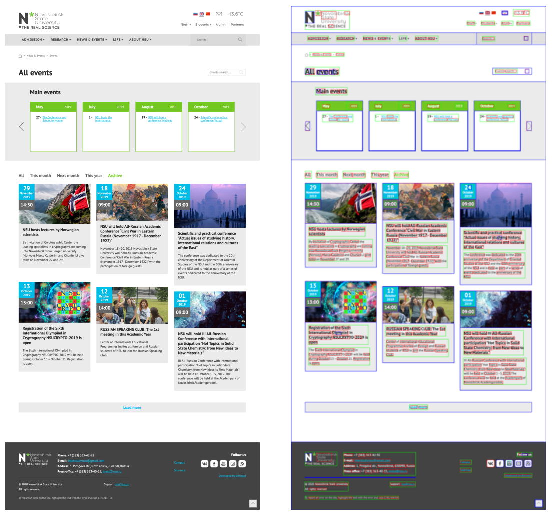

Figure 3: **The first step of processing: elements detection.** The processed image (left) and selected elements and nodes (right). The nodes are bounded with colored boxes.

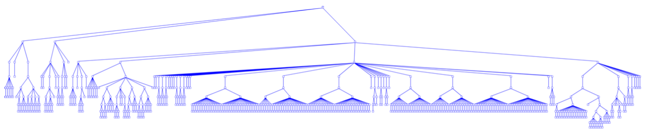

Figure 4: **The second step of processing: structure building.** The tree represents hierarchical relations between elements.

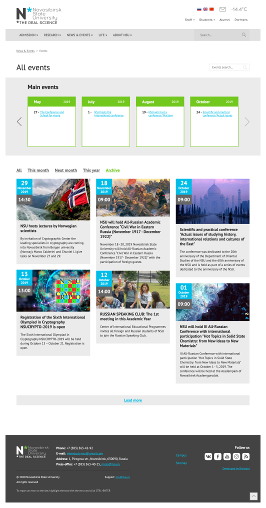

Figure 5: **The final result.** A web-page image generated by the system.

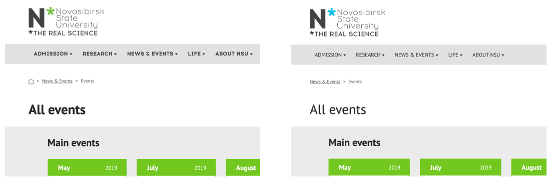

Figure 6: Top-left parts of the input image (left) and the generated result (right). Some features like margins, fonts and others have been incorrectly reproduced. But in general, the layout is generated correctly.

interface by pixel-to-pixel: it is much more important to recognize the idea of the layout. Another option is to compare source codes. We can collect a base of real applications with their source codes, make screenshots, generate a code, and find the difference between the texts. This way is too sensitive to implementation specificity. There are numerous ways to write a code for the same interface and there is no argument to consider the result incorrect if the generated code differs from the real one.

We believe the best way is to compare tree models of the source codes, because a tree model represent the way how a code is organized. We collect the data with crawling web pages so that among the screenshots we get HTML code and extract the DOM model from them. As a rule (except some tricky cases), the DOM model adequately describes elements structure. The idea is to map structures received by our algorithm with DOM trees and roughly estimate their similarity. Such a procedure allows us to evaluate our approach, although we understand that it is limited only on a specific subset of images (only web interfaces) and also some technical tricks of web technologies can affect the correctness of DOM trees and thereby decrease the estimates. Nevertheless, it is the best way of evaluating the method so far, which can prove the viability of the solution and outline the next steps.

Selection of a measure

The crucial thing is the selection of a quality measure. There are few approaches to comparing trees: some of them count operations needed for the transformation of one tree to another; others compare the longest common paths from a root to a tree node; also there are *variable-length doesn't care* (VLDC) based methods.

We select two measures: *edit distance* and *bottom-up* distance.

Tree edit distance. The review of edit distance methods can be found in the survey of Philip Bille. This is how he defines the problem:

Given T is a labelled ordered tree (*ordered* means that order among siblings matters). Define tree operations as follows:

- **Relabel.** Change the label of a node.
- **Delete.** Delete a non-root node v in T with parent v' keeping children of v order.

- **Insert.** Insert a node *v* as a child of *v'* in *T* making *v* the parent of a consecutive subsequence of the children of *v'*.

Assume there is an *edit cost* is $c : V \times V \to R$. $c(v, w)$ denotes relabel, $c(v, \perp)$ denotes delete and $c(\perp, w)$ denotes insert operations. Given an *edit script S* between T_1 and T_2 is a sequence of edit operations $s_1, s_2, \ldots, s_n$, $s_i \in V \times V$ and *cost* of S is $d(S) = \sum_{i=1}^{n} c(s_i)$. Denote an *optimal edit script* between T_1 and T_2 is $S_{opt}(T_1, T_2) : d(S_{opt}) = \min_{S_t} d(S_t)$ and $d(S_{opt})$ is a *tree edit distance*. (Bille 2005)

Based on the survey (Bille 2005), we considered that the best solution of the tree edit distance problem for our case is the Klein's algorithm (Klein 1998), which requires a worst case time bound of $O(|T_1|^2|T_2| \log |T_2|)$ and a space bound of $O(|T_1||T_2|)$.

In our case, we suggest the following edit cost function. First, we denote a fixed finite alphabet Σ containing values for labels:

$$\Sigma = \{t, p, i\}$$

where t stands for an element with text, p stands for an element with an embedded picture, and i stands for an internal node.

We denote edit cost as follows:

$$c(v, w) = \begin{cases} 1, & \text{if } v \neq \perp \wedge w = \perp \\ 0.8, & \text{if } v = \perp \wedge w \neq \perp \\ 0.1, & \text{if } v \neq \perp \wedge w \neq \perp \wedge v, w \in \{p, t\} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Also, denote that a tree received by our algorithm would be the first parameter, and a tree from a data set would be the second one.

This edit cost function penalizes the method the most if a resulted tree misses any node, a little less if it contains an extra node and very little if it mixes up a text with a picture. You can mention that it does not penalize for mixing up an internal node with no internal, because in this case the algorithm either misses or adds an extra node, which already implies a big penalty.

Bottom-up distance. This method is presented in the work of Valentie (Valiente). The complexity of the algorithm is $O(|T_1||T_2| \log(T_1 + T_2))$. For two trees T_1 and T_2 the *bottom-up distance* equals $1 - f/\max(|T_1|, |T_1|)$, where f is the size of the largest common forest in T_1 and T_2. We slightly change this formula and make it asymmetric:

$$b_{asym}(T_1, T_2) = 1 - f/|T_2| \quad (2)$$

Edit and bottom-up distances as defined above can be roughly interpreted as “precision” and “recall” respectively in terms of machine learning evaluation. Technically, they are not the same, but it gives us a good idea of how to read an experiment results. Indeed, for a tree that correctly represents a part of another tree but completely does not contain another part, edit distance would be relatively low, while bottom-up distance would be relatively high. Otherwise, if a tree contains all nodes of another tree, but the structure is different and some extra nodes exist, edit distance would be relatively high and bottom-up distance would be relatively low.

To estimate the approach from different points of view we use both measures as well as the F-score that generalizes a common penalty. The problem of aggregating the scores is that they have different scales: $[0, +\infty)$ for an edit distance and $[0, 1]$ for a bottom-up one. We solve this problem by transforming an edit distance measure. In the beginning, we normalize its value by the size of a sample tree:

$$d_{norm}(T_1, T_2) = \frac{d(S_{opt}(T_1, T_2))}{|T_2|} \quad (3)$$

It allows us to compare edit distance for trees of different sizes but the measure is still not limited from above. Therefore, we apply the logarithm to the measure:

$$d_{log}(T_1, T_2) = -\log\left(\frac{1}{1 + d_{norm}(T_1, T_2)}\right) \quad (4)$$

As d is always positive, the domain of d_{log} is $[0, 1)$. Moreover, the logarithmic form of the measure perfectly suites our idea about estimating the method: we assume that the more trees differ, the less important the exact value of the difference.

Finally, we can denote the F-measure. In order to move from cost scores to measure quality, we subtract distances from one.

$$F = 2\frac{(1-b) \cdot (1-d_{log})}{(1-b) + (1-d_{log})} = 2\frac{(1-b) \cdot (1-d_{log})}{2 - b - d_{log}} \quad (5)$$

Data set collection

We used Alexa service [1] to get a list of 1000 the most popular websites. Then we made screenshots of the main and two random pages of each website. Also, we crawled HTML/CSS codes of each page and transformed them into trees using DOM parser in Python. Due to technical restrictions in some cases, we were only able to collect a dataset of 2640 examples.

In addition, to make the experiment more useful and get more insights we scored each web page with a measure “Text-to-Image ratio”:

$$tti = \frac{T}{T + I} \quad (6)$$

where T is the number of nodes with text content and I is the number of nodes with embedded pictures.

The reason why we use this score is to estimate how a share of text and images affects the result and to define the next steps. It is important to understand which part of the method is the most problematic, and where efforts should be focused to increase the quality as much as possible.

Experimental results

In general, results are as follows (see Table 2):

[1]https://www.alexa.com/topsites

Table 2: The experiment results

	Mean	Variance
Edit distance	0.28	0.028
Bottom-up distance	0.15	0.043
F score	0.78	0.033

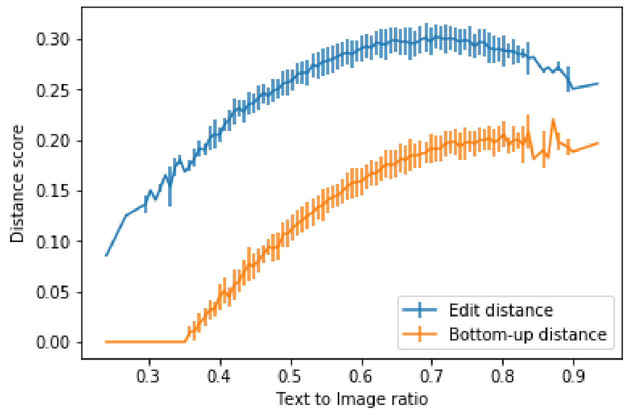

Figure 7: Mean and variance of edit and bottom-up distances depending on “Text-to-Image ratio”

The mean of F-score is 0.78, and the variance is 0.033, which appears to be a good result. Note, that the bottom-up distance is much better than the edit distance. It means that the recognition part, on average, works better than the structure building part.

Let us analyze the dependence of scores on “Text-to-Image ratio” (figures 7 and 8).

We see that the biggest issues are with the cases when an image has both text and embedded pictures, approximately in a proportion of seven to three. Comparing extreme cases, when an image consists of mainly embedded pictures or mainly text, in the first case the approach works far better than in the second one. Herewith, edit distance is decreasing when moving from mixed cases to more text-based, while for the bottom-up distance this effect is not so strong.

Also note the heteroskedasticity of the data: the variance is bigger in “mixed” cases and smaller in extreme cases. That is, the method is more unpredictable when a picture has diverse content.

Based on these outputs, we make the following conclusions:

- The method demonstrated satisfactory results.
- The results can be advanced by applying forces in the following areas:
  - enhancement of the text detection part
  - improvement of the processing of the cases when an image has diverse content

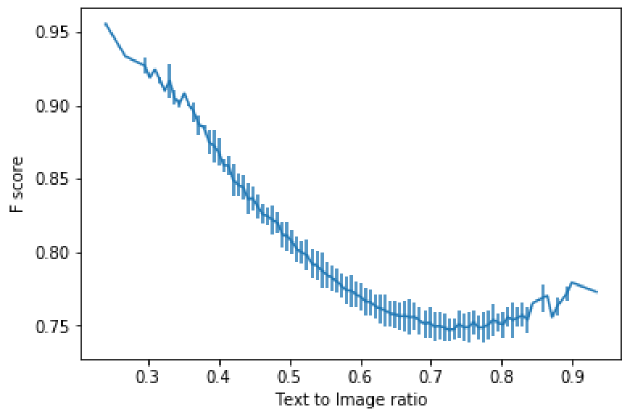

Figure 8: F-score mean and variance depending on “Text-to-Image ratio”

Next steps

In addition to making improvements from the previous section, the next steps include:

1. *Development of a procedure of automatic filling of case storage.*

   The current paper considers the development of a system where case storage is populated manually by experts. As explained above, this approach has strong advantages. However, in general, we would like to collect cases automatically, because it would be less time-consuming and less prone to human errors. Our suggestion is to analyze existing web-sites elements with one of the methods of clustering and use centers of clusters as reference cases.

2. *Development of a better similarity measure.*

   One of the crucial elements of a case-based reasoning solution is the selection of an appropriate similarity measure. The current version uses a simple KNN principle. Consequently, there is room for optimizing the measure construction, because the nearest neighbors algorithm is insensitive to categorical features. As categorical features are the majority of elements properties processed, we are planning to use classifier models based on decision trees.

3. *Development of a revise stage in the CBR-cycle.*

   When building a CBR cycle, the revising stage has been skipped, because the evaluation of HTML document correctness is an unsolved problem. This question should be investigated to make the cycle complete.

In conclusion, we developed a system that generates markup language source code for a given interface screenshot. The feature of the approach is using experts’ knowledge that is kept in a specific case storage. The experiments demonstrated the satisfactory quality of the current solution and provided grounds for the further development.

References

Aamodt, A., and Plaza, E. 1994. CBR: foundational issues, methodological variations and system approaches. *AI Communications* 7(1):39–59.

Ali, K. 2017. A study of software development life cycle process models. *International Journal of Advanced Research in Computer Science* 8(1).

Asiroglu, B.; Mete, B. R.; Yildiz, E.; Nalcakan, Y.; Sezen, A.; Dagtekin, M.; and Ensari, T. 2019. Automatic HTML Code Generation from Mock-Up Images Using Machine Learning Techniques. In *2019 Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science (EBBT)*, 1–4. IEEE.

Beltramelli, T. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. 1–9.

Bille, P. 2005. A survey on tree edit distance and related problems. *Theoretical Computer Science* 337(1-3):217–239.

Carrascosa, C.; Bajo, J.; Julian, V.; Corchado, J.; and Botti, V. 2008. Hybrid multi-agent architecture as a real-time problem-solving model. *Expert Systems with Applications* 34(1):2–17.

Chang, P.-C.; Liu, C.-H.; and Lai, R. K. 2008. A fuzzy case-based reasoning model for sales forecasting in print circuit board industries. *Expert Systems with Applications* 34(3):2049–2058.

Chen, D., and Burrell, P. 2001. Case-Based Reasoning System and Artificial Neural Networks : A Review. *Neural Comput & Applic* 10:264–276.

Chou, J.-S. 2009. Web-based CBR system applied to early cost budgeting for pavement maintenance project. *Expert Systems with Applications* 36(2):2947–2960.

De Renzis, A.; Garriga, M.; Flores, A.; Cechich, A.; and Zunino, A. 2016. Case-based Reasoning for Web Service Discovery and Selection. *Electronic Notes in Theoretical Computer Science* 321:89–112.

Delir Haghighi, P.; Burstein, F.; Zaslavsky, A.; and Arbon, P. 2013. Development and evaluation of ontology for intelligent decision support in medical emergency management for mass gatherings. *Decision Support Systems* 54(2):1192–1204.

Gómez-Vallejo, H.; Uriel-Latorre, B.; Sande-Meijide, M.; Villamarín-Bello, B.; Pavón, R.; Fdez-Riverola, F.; and Glez-Peña, D. 2016. A case-based reasoning system for aiding detection and classification of nosocomial infections. *Decision Support Systems* 84:104–116.

He, W. 2013. Improving user experience with case-based reasoning systems using text mining and Web 2.0. *Expert Systems with Applications* 40(2):500–507.

Klein, P. N. 1998. Computing the edit-distance between unrooted ordered trees. In *ESA*.

Kolodner, J. L. 1992. An introduction to case-based reasoning. *Artificial Intelligence Review* 6(1):3–34.

Lary, D. J.; Alavi, A. H.; Gandomi, A. H.; and Walker, A. L. 2016. Machine learning in geosciences and remote sensing. *Geoscience Frontiers* 7(1):3–10.

Myznikov, P. 2018. Development of the Case-Based Approach of Web Interfaces Reverse Reengineering. *Vestnik NSU. Series: Information Technologies* 16(4):115–126.

Natarajan, S., and Csallner, C. 2018. P2A: A Tool for Converting Pixels to Animated Mobile Application User Interfaces. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 224–235. Gothenburg, Sweden: IEEE.

Navarro-Cáceres, M.; Rodríguez, S.; Bajo, J.; and Corchado, J. M. 2018. Applying case-based reasoning in social computing to transform colors into music. *Engineering Applications of Artificial Intelligence* 72:1–9.

Nguyen, T. A., and Csallner, C. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 248–259.

Ram, A., and Santamaría, J. 1997. Continuous case-based reasoning. *Artificial Intelligence* 90(1-2):25–77.

Sartori, F.; Mazzucchelli, A.; and Gregorio, A. D. 2016. Bankruptcy forecasting using case-based reasoning: The CRePERIE approach. *Expert Systems with Applications* 64:400–411.

Schank, R. 1982. *Dynamic memory: A theory of reminding and learning in computers and people*. Cambridge: Cambridge University Press.

Shen, L.; Yan, H.; Fan, H.; Wu, Y.; and Zhang, Y. 2017. An integrated system of text mining technique and case-based reasoning (TM-CBR) for supporting green building design. *Building and Environment* 124:388–401.

Sun, J.; Li, H.; Huang, Q. H.; and He, K. Y. 2014. Predicting financial distress and corporate failure: A review from the state-of-the-art definitions, modeling, sampling, and featuring approaches. *Knowledge-Based Systems* 57:41–56.

Valiente, G. An efficient bottom-up distance between trees. In *Proceedings Eighth Symposium on String Processing and Information Retrieval*, 212–219. IEEE.

Watson, I. A. N., and Marir, F. 1994. Case-based reasoning : A review. *The Knowledge Engineering Review* 9(4):327–354.

Zeyen, C.; Müller, G.; and Bergmann, R. 2017. Conversational Process-Oriented Case-Based Reasoning. Springer, Cham. 403–419.