Pain Recognition with Cartesian Genetic Programming

Project-Report

Krist Fama
1872175
krist.fama@stud.uni-bamberg.de

Daniel Schäfer
1872091
daniel.schaefer@stud.uni-bamberg.de

Supervisor: Ute Schmid, Michael Siebers

Abstract: Pain recognition has recently received significant attention. Over the past years a lot of work has been done in this field, trying to find the one equation which defines the pain expression of all humans. Hence it is not a trivial challenge and it depends on a lot of factors. Different methods have been tried in automating pain recognition and some of them achieved good results. We chose Cartesian Genetic Algorithm (CGP) to identify pain by generating a Regular Expression (RegEx) that combines Action Units (AUs). We systematically review the mutation, fitness function and the evolutionary algorithm to achieve good results. The implementation steps are summarized and the statistical result are shown. There are two underlying motivations for us to write this paper. First, whether it is possible to achieve a simple RegEx which correctly identifies pain expressions of all humans and second to find out how CGP fits with our problem and what results it achieves.
**Acronym**

**AD**  Action Descriptor
**AU**  Action Unit
**CGP**  Cartesian Genetic Programming
**EA**  Evolutionary Algorithm
**FACS**  Facial Action Coding System
**GA**  Genetic Algorithm
**GP**  Genetic Programming
**KNN**  k-Nearest Neighbors
**NN**  Neural Network
**RegEx**  Regular Expression
**VAS**  Visual Analog Scale
1 Introduction

Scientific work on facial expressions can be traced back to 1862, with the work of the French researcher Duchene [19]. He studied the electro-stimulation of individual facial muscles responsible for producing facial expressions. In 1872, Charles Darwin published “The Expression of the Emotions in Man and Animals” in which the importance of facial expressions for communication and description was explored [3].

Today, facial expressions serve as a primary nonverbal means for human beings to regulate their interactions [8]. They clarify and emphasis what is being said and help us show our emotions. The six basic emotions suggested from Paul Ekman are anger, fear, disgust, happiness, sadness and surprise. According to him, all possible facial expressions can be described with these six emotions or the combination of them [25]. The facial expression measurement in this case is a message judgment, which means that the facial expressions are only measured by the facial display, such as being sad or happy. An objective method to measure the facial expressions is the sign judgment. There, the physical signals which transmit the message are studied. Physical signals would be, for example, ‘depressed lips’ or ‘raised cheeks’ (see figure 2). The most common descriptors in sign-judgment are those described by the Facial Action Coding System (FACS), originally developed by Ekman and Friesen in 1978 [26] and revised in 2002. Specified are 32 atomic facial muscle actions, named Action Units (AUs) and 14 additional Action Descriptors (ADs). In our research we focus only on the AUs.

![Figure 1: Pain expressions collected by Duchene in [19].](image)

*Pain* is not included in the six basic emotions from Ekman, but is has a lot of significance as a subjective unpleasant feeling, experienced differently by various people. The facial expression of pain is considered to be one of the most important non-verbal pain behaviors, because of its salience, its reflexive nature, and its distinction from other affective states [17]. The diagnosis of pain based
on the manual interpretation of facial expressions by human experts seems to be strongly personality related and needs to be seen also in time and surroundings. Although pain and emotion-related processes clearly overlap to some extent, pain experiences always involve a sensory dimension and the expressions of pain have been described to be unique and distinct from those of basic emotions. In our research, we are focusing on this emotion and on the recognition of it in facial human expressions.

This project is based on the research done in the Otto-Friedrich University Bamberg, five years ago, from Stocker et al. [30]. In his work, a Genetic Algorithm (GA) which recognizes pain is implemented. The input of the algorithm is a file with different AU-sequences which represent pain. The algorithm reads through the listed AU-pain-sequences and outputs a complex Regular Expression (RegEx) which ideally represents all or at least a lot of them. The RegExs are implemented in form of a tree. Trees have nodes, which can be leaf-nodes or connection-nodes and quantifiers. Leaf-nodes are nodes which do not have children. Connection-Nodes are used to combine different nodes with the operators and the quantifier may be '?', '+' or '*'. This type of representation is chosen because it fits best with the structure of a GA and it makes mutation and crossover easier. The output RegExs are classified using a fitness value that is calculated by a fitness function. The higher the fitness value of a generated RegEx, the more input AU-pain-sequences are represented from this expression, and so the better this RegEx is.

The goal of Stockers work was to find a RegEx which describes all types of pain-sequences which a human face is able to express. This goal is not a trivial one. Nevertheless, some complex RegEx achieved high fitness function values and were able to represent a lot of the input AU-pain-sequences from the collected data. A problem of Stockers work was, that he had only positive training examples, which means the resulting RegEx may cover all of the examples that the algorithm was trained on, but it also can also cover a lot of facial expressions from other emotions. This motivated us to begin the research in this area again, while looking into another direction.

In this work we will focus on two questions:

- Can we find RegExs describing pain, which are not as complex as in previous works?
- Is it possible to achieve good results using our data and Cartesian Genetic Programming (CGP)?

This work describes our project implementing a CGP which recognizes pain, using different AU-pain-sequences collected from different researches made with people. We structure our survey into six sections. The remainder of the paper is structured as follows. Section 2 presents a theoretical background, describing the FACS, related works done with pain recognition and a brief introduction to CGP. Section 3 describes the implementation steps of our algorithm using CGP, the population, search operators and the fitness function. Section 4 contains a detailed review of the parameters chosen and the evaluation. Section 5 shows different statistics done with our data and with the results of the implementation. Finally, our conclusion for this project and future challenges are provided in section 6.
2 Theoretical background

In this section the theoretical background needed to understand the project is described. The first subsection will define the FACS with the corresponding AUs. The second subsection provides an overview of previous works made until now with the emotion pain and lists some automated systems for pain recognition, that already exist. At the end of the chapter a general overview of CGP is made and mutation and crossover techniques are described. Furthermore an overview of the Evolutionary Algorithm (EA) is given and the reasons why CGP is seen to be efficient towards Genetic Programming (GP) is discussed.

2.1 Facial Action Coding System

The Facial Action Coding System (FACS) is a comprehensive, anatomically based system for measuring all visually discernible facial movements. FACS describes all visually distinguishable facial activity on the basis of 44 unique AUs, as well as several categories of head and eye position and movement.

<table>
<thead>
<tr>
<th>AU number</th>
<th>Descriptor</th>
<th>Muscular Basis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Inner Brow Raiser</td>
<td>Frontalis, Pars Medialis</td>
</tr>
<tr>
<td>2.</td>
<td>Outer Brow Raiser</td>
<td>Frontalis, Pars Lateralis</td>
</tr>
<tr>
<td>4.</td>
<td>Brow Lowerer</td>
<td>Depressor Glabellae, Depressor</td>
</tr>
<tr>
<td>5.</td>
<td>Upper Lid Raiser</td>
<td>Supercili; Corrugator</td>
</tr>
<tr>
<td>6.</td>
<td>Cheek Raiser</td>
<td>Orbicularis Oculi, Pars Orbitalis</td>
</tr>
<tr>
<td>7.</td>
<td>Lid Tightener</td>
<td>Orbicularis Oculi, Pars Palpebralis</td>
</tr>
<tr>
<td>9.</td>
<td>Nose Wrinkler</td>
<td>Levrator Labii Superioris, Alaueque Nasi</td>
</tr>
<tr>
<td>10.</td>
<td>Upper Lip Raiser</td>
<td>Levrator Labii Superioris, Caput</td>
</tr>
<tr>
<td>11.</td>
<td>Nasolabial Fold Deepener</td>
<td>Infratrobiialis</td>
</tr>
<tr>
<td>12.</td>
<td>Lip Corner Puller</td>
<td>Zygomatic Minor</td>
</tr>
<tr>
<td>13.</td>
<td>Check Puffer</td>
<td>Zygomatic Major</td>
</tr>
<tr>
<td>14.</td>
<td>Dimpler</td>
<td>Caninus</td>
</tr>
<tr>
<td>15.</td>
<td>Lip Corner Depressor</td>
<td>Triangularis</td>
</tr>
<tr>
<td>16.</td>
<td>Lower Lip Depressor</td>
<td>Depressor Labii</td>
</tr>
<tr>
<td>17.</td>
<td>Chin Raiser</td>
<td>Mentalis</td>
</tr>
<tr>
<td>18.</td>
<td>Lip Puckerer</td>
<td>Incisivii Labii Superioris; Incisivii Labii Inferioris</td>
</tr>
<tr>
<td>20.</td>
<td>Lip Stretcher</td>
<td>Risorius</td>
</tr>
<tr>
<td>22.</td>
<td>Lip Funneler</td>
<td>Orbicularis Oris</td>
</tr>
<tr>
<td>23.</td>
<td>Lip Tightener</td>
<td>Orbicularis Oris</td>
</tr>
<tr>
<td>24.</td>
<td>Lip Pressor</td>
<td>Orbicularis Oris</td>
</tr>
<tr>
<td>25.</td>
<td>Lips Part</td>
<td>Depressor Labii, or Relaxation</td>
</tr>
<tr>
<td>26.</td>
<td>Jaw Drop</td>
<td>Masketter, Temporal and Internal</td>
</tr>
<tr>
<td>27.</td>
<td>Mouth Stretch</td>
<td>Pieryoide, Digastric</td>
</tr>
<tr>
<td>28.</td>
<td>Lip Suck</td>
<td>Orbicularis Oris</td>
</tr>
</tbody>
</table>

Figure 2: Single AUs in FACS as described in [9].

Every possible facial expression can be objectively described as a combina-
tion of AU. A list with the AU coded in [FACS] and the muscle groups involved in each action are shown in figure 2 and figure 3 [9]. AU can occur either alone or in combination. When AU occur in combination, they can be additive, which means that the combination does not change the appearance of the constituent AU. If they are non-additive, the appearance of the constituents does change. Although the number of atomic AU is relatively small, more than 7,000 different AU-combinations have been observed. FACS provides the descriptive power that is necessary to describe the details of facial expressions [32].

With facial action unit coding it is possible to dissociate between the three following facial expression categories:

- **Macroexpressions:**
  Typically last between 0.5 – 4 seconds, occur in daily interactions and are generally obvious.

- **Microexpressions:**
  Last less than half a second, occur when trying to consciously or unconsciously conceal or repress the current emotional state. Microexpressions are much harder to be detected than macroexpressions.

- **Subtle expressions:**
  These expressions are associated with the intensity and depth of the underlying emotion. The intensity of these facial actions constantly varies.

<table>
<thead>
<tr>
<th>AU number</th>
<th>FACS name</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.</td>
<td>Lips Toward Each Other</td>
</tr>
<tr>
<td>19.</td>
<td>Tongue Out</td>
</tr>
<tr>
<td>21.</td>
<td>Neck Tightener</td>
</tr>
<tr>
<td>29.</td>
<td>Jaw Thrust</td>
</tr>
<tr>
<td>30.</td>
<td>Jaw Sideways</td>
</tr>
<tr>
<td>31.</td>
<td>Jaw Clencher</td>
</tr>
<tr>
<td>32.</td>
<td>Lip Bite</td>
</tr>
<tr>
<td>33.</td>
<td>Blow</td>
</tr>
<tr>
<td>34.</td>
<td>Puff</td>
</tr>
<tr>
<td>35.</td>
<td>Cheek Suck</td>
</tr>
<tr>
<td>36.</td>
<td>Tongue Bulge</td>
</tr>
<tr>
<td>37.</td>
<td>Lip Wipe</td>
</tr>
<tr>
<td>38.</td>
<td>Nostril Dilator</td>
</tr>
<tr>
<td>39.</td>
<td>Nostril Compressor</td>
</tr>
<tr>
<td>43.</td>
<td>Eyes Closure</td>
</tr>
<tr>
<td>45.</td>
<td>Blink</td>
</tr>
<tr>
<td>46.</td>
<td>Wink</td>
</tr>
</tbody>
</table>

Figure 3: More general defined AU in the [FACS] [9].
2.2 Pain using FACS

Pain is described differently from researches over the years. Keefe et al.\[13\] defined pain behavior in the following way:

...People who are in pain may vocalize their distress by moaning, crying or complaining, or may exhibit pain related body postures or facial expressions. These verbal and nonverbal behaviors have been called pain behaviors because they serve to communicate the fact that pain is being experienced.

Here pain behaviors are seen as communicative ones. Charles Darwin \[8\] characterizes pain expressions this way:

...[in pain] the mouth may be closely compressed, or more commonly, the lips are retracted; with the teeth clenched or ground together... the eyes stare wildly as if in horrified astonishment.

Darwin was the first to make a major attempt to place the study of pain expression on a scientific basis. He focused on expressive behaviors as a means of understanding the origin and functions of motivational and affective states.

Later on different researches were made related to pain, some of them from Hollander \[6\], who studied experimental pain by placing a metal grater under a blood pressure cuff and inflating it, noting the pressure at which the subject 'winced' and other researches from Chapman and Jones \[31\], who used heat to evaluate the 'pain-reaction'.

LeResche \[16\], Craig and Patrick \[12\] and Patrick et al. \[27\] used the FACS from Ekman and Friesen \[26\] in order to characterize facial expressions in various situations. The relative consistency with which the same actions were associated with pain in these studies supported the concept of a characteristic, possibly universal 'pain expression'. A combination of facial actions that appear to be specific to pain is identified. Core \(\text{AU}4\), in adults are brow lowering (\(\text{AU}4\)), nose wrinkling and upper lip raising (\(\text{AU}9\) and \(\text{AU}10\)), cheek raising and lid tightening (\(\text{AU}6\) and \(\text{AU}7\)), and eye closure (\(\text{AU}43\)).

In a recent follow up to this work, Prkachin and Solomon \[15\] confirmed, that these four "core" actions contained the majority of pain information. They defined pain as the sum of intensities of orbital tightening, levator contraction, brow lowering and eye closure. The Prkachin and Solomon pain scale is defined as:

\[
Pain = \text{AU}4 + (\text{AU}6 \| \text{AU}7) + (\text{AU}9 \| \text{AU}10) + \text{AU}43
\]

In this equation pain is defined as the sum of \(\text{AU}4\), \(\text{AU}6\) or \(\text{AU}7\) (whichever is higher), \(\text{AU}9\) or \(\text{AU}10\) (whichever is higher), and \(\text{AU}43\). \(\text{AU}43\) which presents the eye closing can be true or false (0 = absent or 1 = present) and the other \(\text{AU}\)s are scored from 0 (= absent) to 5 (= maximum intensity). This way it is possible to define pain on a 16-point scale. Figure 4 illustrates the mentioned \(\text{AU}\)s used to describe pain.
2.2.1 Automated Analysis of Pain

Pain is difficult to access and manage. It is fundamentally subjective and is typically measured by patient self-report, either through clinical interview or Visual Analog Scale (VAS). Using the VAS a patient indicates the intensity of pain by marking a line on a horizontal scale, anchored at each end with words such as "no pain" and "the worst pain imaginable". This and similar techniques are popular because they are convenient, simple, satisfy a need to attach a number to the experience of pain and often yield data that confirm expectations [2].

In the past several years, significant progress has been made toward the development of computer-based automated analysis of facial expression [4]. Information technology approaches hold considerable promise to advance this area. In the following some of the work made in automated analysis of pain will be listed with their performances. An automated system which is able to classify pain in video sequences is implemented in the paper of [18] using the FACS with the different AUs. Pain was detect in two possible ways: directly from facial features or indirectly through the combination of individual AU-detectors. For the recognition of pain based on AU the Prkachin and Solomon pain scale was utilized. Ashraf et al. [2] use reference points on video sequences to classify pain. Their achievement was an accuracy of 71%. Monwar and Rezaei [24] used images extracted from videos. They tried different classifiers and achieved the highest accuracy, 92%, with Neural Network (NN). Zafar and Khan [33] used geometric features extracted from 22 facial points and a single step k-Nearest Neighbors (KNN) classifier. A limitation of their approach is that it requires a prior annotation of the neutral face for each subject. Egete et al. [7] use Deep Learning and hand crafted features of appearance, shape and dynamics and achieved 67.3%.

Attempts to automate analysis of pain are constantly increasing. Despite current achievements, there are still open challenges with automatic pain recognition. For most recognition systems, the performance of pain recognition models is largely dependent on the quantity and quality of data used in training. Human expression data is limited in supply to begin with but pain data is particularly difficult to obtain.
2.3 Cartesian Genetic Programming

CGP was developed by Julian Miller and Peter Thomson in 2000 [23]. It is a popular and efficient form of GA and represents programs as indexed, acyclic and directed graphs via genotype-phenotype-mapping. The benefit of this type of representation is that it allows the implicit re-use of nodes in the directed graph [5]. It is called ‘Cartesian’ because it represents a program using a two-dimensional grid of nodes. This algorithm can represent Neural Networks (NNs), programs, circuits and many other computational structures. An important feature is the ease with which it is able to handle problems involving multiple outputs.

Figure 5: Genotype and phenotype for the CGP individuals. [21]

Figure 5 shows an example of a genotype and its corresponding phenotype. The sequence of numbers at the top of the figure represents the genotype, in which each group maps to one node of the graph except for the last. This group specifies the output of the program. The underlined numbers in front of the group describe the function to perform and the non-underlined numbers describe the index of the inputs to work on. The algorithm evaluates a program via recursive backward search through the directed graph. It starts from program output and processes all nodes directly or indirectly linked to it. In this way, only active nodes are processed whereas inactive nodes, which are not connected to the output, remain untouched.
Figure 6 shows the decoding procedure. In the a) Section of figure 6 the output A (\(O_a\)) connects to the output of node 8 and moves to node 8. In b), node 8 connects to the output of nodes 2 and 7 and moves to nodes 2 and 7. In c), nodes 2 and 7 connect to the output of node 6. In d), node 6 connects to the output of nodes 2 and 4 and moves to node 4, as node 2 has already been decoded. In e), node 4 connects to the output of nodes 2 and 3 and moves to node 3. In f), node 3 connects to program input 1. The genotype is fully decoded when the recursive process has finished.

CGP has three parameters that are chosen by the user. These are the number of columns, the number of rows and level-back. These are denoted by \(n_c\), \(n_r\) and \(l\) respectively. The product of the two first parameters determine the maximum number of computational nodes allowed

\[
L_n = n_c n_r
\]

The parameter \(l\) controls the connectivity of the graph encoded. Levels-back constrains which columns a node can get its inputs from. If \(l = 1\), a node can get its inputs only from a node in the column on its immediate left or from a primary input. If \(l = 2\), a node can have its inputs connected to the outputs of any nodes in the immediate left two columns of nodes or a primary input [23]. Varying these parameters can result in various kinds of graph typologies.

### 2.3.1 Mutation in Cartesian Genetic Programming

The mutation operator used in CGP is a point mutation operator. If a function gene is chosen for mutation, then a valid value is the address of any function in the functionset, whereas if an input gene is chosen for mutation, then a valid value is the address of the output of any previous node in the genotype or of any program input. Also, a valid value for a program output gene is the address of the output of any node in the genotype or the address of a program input. The
number of genes in the genotype that can be mutated is defined by the user. This is normally a percentage of the total number of genes in the genotype.

![Diagram of CGP genotype and phenotype before and after mutation](image)

Figure 7: Example of a mutation in CGP [23].

In figure 7, an example of the point mutation operator before and after it is applied to a CGP genotype, and the corresponding phenotypes are shown. This figure highlights how a small change in the genotype can sometimes produce a large change in the phenotype. A single point mutation occurs in the program output gene (OA), changing the value from 6 to 7. This causes nodes 2, 5 and 6 to be inactive, and nodes 3 and 7 to become active.

2.3.2 Crossover Technique in Cartesian Genetic Programming

GP was first introduced by Koza in 1992 [10], [11]. Tree structures were used for the representation of the members of the population. A crossover technique in which random sub-branches of the parents tree-structures are swapped, was suggested. This sub-tree crossover was, at the time, though to be the dominant operator within the optimization process: responsible for exploiting existing genetic material while searching for better solutions. Some later researches [1], [28], [29], in this direction showed that this sub-tree crossover technique does not always perform well. In [1], the performance of a sub-tree crossover, with a
crossover technique which simply mutated the sub-branches of the tree were compared. It was found that the difference between the performances of sub-tree crossover and that of simply mutating the sub-branches was statistically insignificant. This result implied that, in some cases, sub-tree crossover was not better than some simple mutation of the sub-branches. In [28] and [29], a comparison between sub-tree crossover and a simple mutation of the branches of the trees, over a range of problems was also made. They also concluded that sub-tree crossover performed little better than a simple mutation of the branches. Due to findings like these, the implementation of a crossover does not seem to be necessary in some cases of GP and instead the focus can completely lie on the mutation. For this reason CGP, which also uses directed graphs to represent programs, was introduced by Miller [23], including only mutation-functions for the evolution process. Our implementation of the CGP also implements the algorithm without the crossover technique, using the mutation only.

Recently, Miller et al. [20] represented a new crossover technique for CGP. The CGP representation is modified in order to enable the new crossover technique to be applied. Crossover, when applied to a CGP using the traditional representation, hinders its performance rather than improves it, which was their motivation in modifying the representation and creating the new crossover techniques. In our case, the limited time did not allow us to change the representation and to prove, if our results would improve with this new crossover technique. But for the future, it would be a possibility to work towards.

### 2.3.3 Evolutionary Algorithm

An Evolutionary Algorithm (EA), which is widely used for the implementation of an CGP algorithm, is the 1+α algorithm. Usually α is chosen to be 4. The algorithm has the form as shown in Figure 8.

```plaintext
1: for all i such that 0 ≤ i < 5 do
2: randomly generate individual i
3: end for
4: select the fittest individual, which is promoted as the parent
5: while a solution is not found or the generation limit is not reached do
6: for all i such that 0 ≤ i < 4 do
7: mutate the parent to generate offspring i
8: end for
9: generate the fittest individual using the following rules:
10: if an offspring genotype has a better or equal fitness than the parent then
11: offspring genotype is chosen as fittest
12: else
13: the parent chromosome remains the fittest
14: end if
15: end while
```

Figure 8: The (1+4) evolutionary strategy [23].

On line 10 of the algorithm, an extra condition takes place. When one or more offspring genotypes in the population have the same fitness as the parent and when there is no offspring that is better than the parent, an offspring with equal fitness is chosen as the new parent. This extra condition is very important and makes good use of redundancy in CGP genotypes.
2.3.4 The Efficiency of Cartesian Genetic Programming

In its most common form, CGP uses a very small population (usually five individuals). This is often evolved for a large number of generations, until a solution is found for a particular problem or a stop criterion (e.g. generation limit) is reached. In this case, it differs from GP which starts with a very large population (usually several thousand individuals). This population is then involved until a solution is found or generation limit is reached. However, when the number of fitness evaluations \(\text{(number of generations x population size)}\) for the two different approaches is calculated, the difference between the efficiency of the two techniques is more comparable. In papers of Miller like [22] and [21] CGP has been shown to be more efficient than GP.

There are two ways in which CGP often has been used to find efficient solutions to different problems. On one hand, by modifying the fitness function, so that when solutions are found they are discarded in favor of more efficient solutions. In this case, the EA is continued for longer time. The advantage in this case is not knowing the right time to stop the evolutionary cycle. Often the smallest solution is unknown. Therefore, in these cases, an open-ended evolutionary process is left running for extremely long periods, to see if a more compact solution can be found.

On the other hand, another approach that is used, is the one in which the genotypes have very short lengths. These were also evolved for a very large number of generations [20]. In this case, compact solutions were found but the very short genotypes hindered performance. This is because the amount of effort required to evolve solutions in CGP is strongly dependent on the chosen genotype length. Large genotypes require markedly less effort to evolve solutions [20]. Both of these approaches were aimed at producing correct but also compact solutions. Of course, for making fair comparisons between CGP and GP the nature of the experiment should be the same. Furthermore techniques with approximately equal program lengths should be compared and the primitive function sets should be the same.

Looking at the efficiency of CGP in different directions toward GP it is motivational for us to try out in our project and to hope that we get more profound results with the given input test data.
3 Implementation

The implementation of the CGP algorithm in this project is based on the work of J. Miller [23]. The algorithm uses a network of nodes which are connected to each other to represent a regular expression. Each individual has an own network and multiple individuals create a population. The evolution of the population is done by mutation and with a fitness-function each individual is evaluated.

3.1 Representation

The algorithm bases on the idea of a cartesian coordinate system. It has multiple inputs that operator-nodes can use to build up a chromosome, in this case a RegEx, that combines inputs by using the operator-nodes to a string, that can then be evaluated. An overview of the structure can be seen in figure 9.

In our case the inputs are the AUs. From the given positive input-file, all AUs are used as possible input. Furthermore a more general representation is created from each AU that looks like this the following representation of au7: \texttt{au(\ldots.\ldots)?(\ldots.\ldots)?}. This is done, to prevent overfitting, when it comes to combined AUs, which appear multiple times in the training-data.

The operator-nodes can be seen in figure 10. We implemented four possible operators, which are generated and selected randomly during the execution of the algorithm, which are: \texttt{AND}, \texttt{OR}, + Quantifier and the ? Quantifier. We decided to leave out the * quantifier, since it is possible for the algorithm to generate it by chance from the + and ? quantifiers. Additionally we decided to keep the structure of the \texttt{AND} and \texttt{OR} node also for the quantifier-nodes, even though they only use the first input. This means the second input is lost but will eventually gain new value, if the operator of the node is changed by mutation.

The output is called start-node and is one of the operator-nodes. It is selected randomly from the network of nodes in the generating process of the individual. It can be any of the operator nodes, except of the input nodes. Another idea we evaluated was, to use one of the nodes in the final column. The reason, why we did not pursue this approach was, that the resulting RegEx could become unnecessarily long, even though by evolution it would also be possible to receive shorter ones. The approach with completely randomized start position offers a wider range of possible results and also includes the above described nodes in the final column.
The representation in figure 11 shows what the result of each operator is, with the same input to any of the operators. Mentionable here is the AND operator, where the two inputs are just cohered and the operator itself has no representation in the resulting string. The brackets for the quantifiers are needed, to tell the pattern-matcher the range of it.
3.2 Population

The population of our implementation contains a number of individuals, which are independent from each other but not independent from their ancestors. After the start-population, each new individual is most probably a slightly changed version of the previous one, since it is a copy that of the so called parent-individual that goes through a mutation process.

We followed the basic idea of Miller[23] and implemented an 1+4 evolutionary algorithm. This means for each individual in the population four child-individuals are generated. If the fitness of any of the child individuals is equal or better than that of the parent, this child is selected to go to the next evolution-round (see section 2.3.3).

The size of the population has huge influence on the the quality the result. Miller mentions in his book a population size of five, which we thought to be too small, since an unlucky situation of five unfit start-populations and/or operator-nodes can lead to a long running algorithm until it reaches the wished precision. The same is valid for the opposite, a huge population can slow down the development of the good individuals.

3.3 Search Operators

In general there exist two approaches, that can also be combined, to generate new individuals in an evolutionary algorithm. The crossover and the mutation. In many approaches the first step for the evolution of individuals is the crossover, where two (or sometimes even more) individuals are selected and parts of their identity are switched which finally results in two new individuals. The mutation then makes further small changes on random values, that represent the individual. For CGP this is not always the best approach and in most cases only the mutation is used.

3.3.1 Crossover

For our problem, where RegExs are generated from a given set of examples in a network of nodes, a crossover is not useful since it would completely randomize the newly generated individual. There would be no learning effect from the two parents by taking one part of the one parent and the other part of the network of the second parent. For this reason and also following the guideline of Miller[23] we decided against a crossover operation during the evolution (see also 2.3.2).

3.3.2 Mutation

The mutation is the essential part of the evolution in our CGP algorithm. First of all we decided to not mutate the input nodes, since that would falsify our result. The mutation focuses completely on the operator-nodes (see section 3.1) and on changing the start-node, meaning the node from where the RegEx is build up.

The start-node plays an important role in our implementation. It can be understood like a pointer, that in the initial population randomly selects one of the nodes of the network and sets it as the start-point from which the RegEx is generated recursively. By switching the start-node, the resulting individual
most probably becomes a completely new result. For this reason the mutation rate for this operation is recommended to stay low, due to the impact it has.

3.4 Evolution

We implemented the \textit{1+4} evolution, which is the recommended technique for CGP-algorithms. In the beginning the evolution process generates the predefined number of individuals. Then their fitness is calculated. In the next step, four children are generated for each of the initial individuals. They are mutated copies of the parent-individual. The fitness is calculated for each child, if the fitness is better than the parents fitness, the child replaces the parent. If the fitness is equal, the negative fitness is taken into account. Children that have equal positive fitness and lower or equal negative fitness replace their parent as well.

In contrast to Miller, we used more than five individuals in our population and received better results with that measured on basis of 100 000 evolution steps. The increased population size increases the runtime but also provides more possible solutions and reaches higher fitness values in shorter time.

3.5 Fitness Function

The fitness function calculates the fitness-value for each individual. Since there is positive and negative training-data, where the positive training-data represents the type of facial expression that should be learned by the algorithm and the negative training-data is used to prevent from generalization, the fitness function should take into account both examples, to make the result of the evaluation valuable.

One approach that we thought of was to exclude all examples from the positive and negative input files, that appear in both files and then count the positive and negative examples, that are represented by the \texttt{RegEx}. In our given training data with pain and disgust we had 39 examples in the positive training-data (pain), that were also in the negative training-data (disgust). We decided against this approach, because we show the uncertainty with the positive and negative fitness and in some cases the facial expression is the goal-expression, which would make the \texttt{RegEx} not representative.

Furthermore we thought about filtering the input files in a way that only one of each existing kind of example is left in the positive and in the negative training-data. This means that for example \texttt{au7} appears eight times in the positive training-data but will be counted just once for the fitness. The same holds for the negative fitness. We decided against it, to keep the influence of \texttt{RegEx} that appear multiple times on the fitness, because they most probably represent a more general facial expression than the ones that appear fewer or just one time.

The function that we implemented uses both training-data-sets in a way that they both have influence on the fitness of the individual. For the positive examples, each individual that is represented by the \texttt{RegEx} is counted as one point for the positive fitness. Furthermore all negative examples that are represented by the \texttt{RegEx} are counted in a variable for negative fitness. If a child has an equal fitness to its parents, this negative fitness is compared, and if it is worse
than the one of the parent, the parent individual is said to be the better one and is kept.

The fitness value that is the evaluation of the individual only represents the positive examples that are represented by the RegEx. We decided for this approach instead of for example subtract the negative fitness from the positive fitness, because the two input files can differ in size.

### 3.6 Parameters

To arrive at good parameters for CGP normally requires some experimentation on the problem being considered. A suitable mutation rate depends on the length of the genotype (in nodes). As a rule of thumb Miller suggests to use about 1% mutation if a maximum of 100 nodes are used, which corresponds to our case\[23\]. The other parameters are also oriented on the values of Miller but are slightly changed due to our experiments with the algorithm, that can be seen in section 4. Our goal was to achieve the highest possible fitness-value. In the table below (see figure 12) the selected parameters and the sizes of the utilized input data are listed.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>10</td>
</tr>
<tr>
<td>Mutation Rate Start-Node</td>
<td>1.25%</td>
</tr>
<tr>
<td>Mutation Rate Operator-Nodes</td>
<td>1.0%</td>
</tr>
<tr>
<td>Maximal generations</td>
<td>100,000</td>
</tr>
<tr>
<td>Levels of Nodes</td>
<td>20</td>
</tr>
<tr>
<td>Nodes per Level</td>
<td>4</td>
</tr>
<tr>
<td>Selection-Process</td>
<td>(1+4)-selection</td>
</tr>
<tr>
<td>Size of Negative Training-Data</td>
<td>123 \textit{Disgust}</td>
</tr>
<tr>
<td>Size of Positive Training-Data</td>
<td>347 \textit{Pain}</td>
</tr>
<tr>
<td>Best Fitness-Value</td>
<td>29.4%</td>
</tr>
</tbody>
</table>

Figure 12: Selected Parameters and Input-Data

### 3.7 Results

The resulting program is a runnable algorithm, that is able to generate a regular expression. The quality of the results with the given training-data is not satisfying. In most cases the resulting RegEx is able to identify around 25% of the examples after 100,000 generations. On the downside there are still a lot of examples from the negative training-data that are also covered by the RegEx.
4 Statistics

This section presents statistics that we collected using different parameters on
the algorithm. This is done in order to see how the fitness value differs and in
order to optimize the setting of the parameters to further improve the algorithm.

4.1 Different Population Sizes

The first graph shows how the fitness-value changes with different population
sizes in a range from 5 to 35 individuals. We expected the results to increase
with an increasing size of the population, due to higher chance of finding a
Regex with high fitness-value. We tested the algorithm with each parameter-
setting multiple times and the values shown in figure 13 are averages of these
runs.

![Figure 13: Result with different population-sizes.](image)

The parameters which describe this graph are:

- Selection: 1+4
- Maximal Generations: 100 000
- Node Levels: 20
- Nodes per Level: 4
- Mutation Probability Startnode: 1%
- Mutation Probability Operatornode: 1%

There can be no conclusion drawn from the results. Hence, the best fitness-
value occurs with a population size of 15 individuals and has a value of 78. This
value means that 78 examples from the input-data will be recognized. Since
there are 347, this function value indicates an accuracy of 22.5%.
4.2 Different Mutation-Probability for Start-Node

The graph in figure 14 should show how the fitness-value differs with various mutation probability values for the start-node in the interval [1% – 2%]. The best fitness function value in this case is achieved with a mutation rate of 1,25% and 1,75% both recognizing 70 examples from the input data. The accuracy in this case is 20,2%. The parameters in this graph are the same as in the first graph (see section 4.1), except from the population size, which this time is 15 individuals, since it achieved the best fitness value in the first graph. We evaluated the result in a way, that the mutation probability has no measurable influence in this range and that the variation in the fitness-value, especially for the results with probability of 1,5% resulted from coincidence.

![Figure 14: Result with different mutation-rate for start-node.](image)

4.3 Different Mutation-Rate for Operator-Node

After that we tested the algorithm with different mutation-rates for the operator-nodes. The value of the probability of the mutation for operator-nodes has a high impact on the results, because for every node in every individual it is tested if this node is mutated or not. Therefore we assessed a high importance to this value and hoped, that we could see positive developments when changing this value. We tested the mutation-rate in a range from 0,1% up to 2,0%. We consider it interesting that it seems as though the value of 1,0% returns the best results. In our case 81 of 347 examples where classified correctly which represents an accuracy of 23,3%. The graph to these results can be seen in figure 15.
4.4 Different Maximum for Generations

Finally the stop criterion of maximum generations is varied. We tested it in a range from 100.000 up to 1.000.000 generations. The population-size now was set to 15 and the mutation-rate for the start-node is set to 1,25% and the mutation of the operator-nodes is set to 1,0%. We expected the result to increase according to the high number of generations, but still faced the algorithm getting stuck in local maximal values. The graph in figure 16 shows again, as in the previously discussed parameter-tests, that the average results do not get influenced by the changed parameter as wished. Even though a slight increase of the fitness can be seen this time over the amount of generations,
with a slight irritation at a generation-size of 300,000 evolution-steps. This could lead to the assumption, that the algorithm could produce better results in a long-run-test.
This paper has presented the implemented CGP on the basis of the work from Miller [23]. The Cartesian Genetic Programming is a good and well documented genetic algorithm. We found a lot of helpful papers to that topic. In our opinion the fitness-function plays a key-role. As already mentioned in section 3.5 we tested multiple approaches for it. Moreover the quality and the length of the example datasets seems to be a problem that makes the choice of the correct fitness-function and therefore the development of a strong RegEx difficult.

During the research and the development some problems appeared that could be targets for future works in this field. The data that we used for training and testing of the algorithm seems to be not precise enough. There are a lot of short and therefore low expressive examples in the data. One idea would be to only look at the long expressions due to their probably higher expressiveness. Another idea would be to reward RegExs that are able to correctly classify these longer examples with an increased fitness-value.

Additionally we tested to completely ignore data, that appears in both, the positive and negative, training-datasets. We were not able to further pursue that approach and focused on the improvement of the mutation. Furthermore contains the underlying pain-dataset has 347 examples while the disgust-dataset has 123 entries. This makes a comparison of the positive and negative fitness difficult, which is the reason, why we used the negative dataset as a secondary factor for evaluating the fitness of equal individuals (see section 3.5).

Besides the above points a ten-fold-cross-validation should be done and can be a chance to avoid the overfitting of the result and to optimize the parameters further using this technique. Due to the limited size of the dataset it might still be strongly dependent on the selected training- and testdata.

There is much more work to be done and many questions remain. A crossover technique could be accomplished with our algorithm but as mentioned in section 2.3.2 would it really confer a considerable advantage for the fitness function and would the result be better?

Coming again to the initial questions, that this work focused on were. Regarding the first question:

**First Question:** Can we find RegExs describing pain, which are not as complex as in previous works?

we can conclude, that the resulting RegExs is less complex but on the downside also less expressive regarding the amount of classifiable facial expressions. Additionally the runtime between generations is stable and predictable, where the approach from Stocker increased over the runtime. This enables to predict the necessary resources that the CGP needs.

**Second Question:** Is it possible to achieve good results using our data and CGP?

From the current work with the CGP we have to conclude, that it is not possible to receive satisfying results. The algorithm will most likely produce a RegEx over a much longer run. For our testing and due to constant improvements on the program we were not yet able to let it run for more than two days.
References


