Proposed algorithm of Grammatical Evolution/Inference

In this section the GE algorithm presented in this thesis is going to be analyzed. The main objective of the proposed algorithm is defined as
Given a sampled set of words from a language L(G), whereas L(G) is a CFG, and an arbitrary set of words $\nsubseteq$ L(G), find the corresponding grammar G in the form of a 4-tuple (N,T,P,S).
In the proposed algorithm of GE, GAs and GNF grammars are used. GAs offer a robust way to search the candidate solution space, while GNF offers some exploits that can improve the convergence of the algorithm. The usage of GNF is a novelty as, it has never been used again for grammatical inference or grammatical evolution methods in previous works. Note that GNF does not make the algorithm less general as every CFG can be transformed to GNF.

The steps of the proposed algorithm are going to be presented in the next section, while in section 4.2 the improvements that this implementation offers are going to be analyzed. Finally, in the last section of this chapter, the C++ implementation that is used for all the conducted experiments is going to be presented.

Proposed GE algorithm using GNF

Data requirements and parameters definition

The proposed GE inference algorithm for GNF estimation requires a number of data strings and the definition of several parameters which affect the algorithm efficiency.

1. Two training datasets. The *positive set* of terminal sequences Sp is used to be effectively parsed by the GFN solution, i.e. $Sp \subset L(GNF)$ and the *negative set Sn* of terminal sequences is used to avoid generalization of the candidate solutions, i.e. $Sn \cap L(GNF) = \{\}$.

2. Evolutionary variables:

   The *mutation rate* which is the chance of the mutation operator being applied to an individual.

   The *elitism rate*, the percentage of best individuals being migrated to a new population.

   The *size of the population pool* which is the number of candidate solutions in every iteration of the algorithm.

The *convergence criterion* in the form of a fitness value that is considered sufficient for a good candidate solution. An additional cap to the maximum iterations number can be used as a second criterion.

The *Parental portion* which is the percentage of the best individuals that will be assigned with a probability to generate offsprings.

3. GNF variables:

   The *Size of the set N* (non-terminal symbols) in order to generate this set in the next step. Let the size of the set N being n_N.

   The *maximum length of the production rule* (RL_{max}) and a *maximum number of the production rules* (NoR_{max}) in a GNF grammar to be used in the pool initialization step. The length of a production rule is defined as the number of symbols that the rule is consisted of. (i.e. $S \rightarrow aABC$ has length 5).

   The small positive number a (< 1) as a sub-parse variable. The purpose of this variable will be described in the next paragraph.

The proposed GNF inference using GE

The proposed algorithm is presented step by step in this paragraph. There are seven steps in the algorithm and are visually summarized after the presentation of the algorithm.

Step 1 **Preprocessing**:
This step is optional and may be skipped but here is presented the mapping of the data that are going to be used by the algorithm as this step is redundant in later iterations of this implementation. It is beneficial to scan the datasets and register all unique symbols. This set of symbols is the set T and let the size of the set T being n_T. After the set T is defined all data are enumerated. In the next steps the process of approximating the set P of production rules. Because in this implementation all data and sets are enumerated the set P that corresponds to a grammar G is going to be called the phenotype of an individual and its enumerated form will be called the genotype of an individual in the population.

Step 2 **Initialization**:
The three sets of the GNF: N,T,S are defined in the first steps of the algorithm, whereas P will be estimated in the iterative steps. In this implementation, all the sets are enumerated to accelerate the computer based processes. According to this approach, the starting symbol S $\equiv 0$, and the set T (with n_T symbols) will be the one to one map of each unique character or terminal symbol to an integer value. The number n_N of non-terminal symbols is used to generate the set N = $\{n_T + 1 \sim n_T + n_N\}$.

Set	From	To
S	0	0
T	1	n_T
N	$n_T + 1$	$n_T + n_N$

Note that in the enumeration map for the three sets (S,N,T), the full integers set is continuous (no integer is skipped).

A maximum genotype length must be chosen as the populations that are going to be derived may have arbitrary genotype length due to the genetic operators. In order to not interfere with the generality of the algorithm, a maximum genotype length is chosen as RL_{max} * NoR_{max} (the maximum length of a rule times the maximum number of rules). This means that as individuals evolve individuals may have either GNF grammars with either, many but small production rules or, few with but bigger production rules for a given genotype size.

The maximum length of each production rule RL_{max} and the maximum number of production rules NoR_{max} is used to generate random new individuals in the initial population. For the initial population, individuals are generated with genotypes that have maximum length of double the allowed, as evolutionary techniques tend to search randomly in initial generations. This technique showed to significantly raise the chance of good initial grammars.

The process of random individuals generation is as follows. Random production rules are generated first in two steps. Firstly, a symbol from the set N and a symbol from the set T are selected and are used as the head and the body of the production rule respectively. Then, the body of the rule is appended with 0 up to $RL_{max} - 2$ (equally probable) random symbols from the set N. This process generates random enumerated production rules with length of 2 up to RL_{max}. This process is repeated 1 up to $2 * NoR_{max}$ (equally probable) in order to generate one candidate individual. If a random individual does not include the starting symbol S in the head of any production rule, this individual is discarded and the random generation process is re-activated for this individual. Finally, the genotype of each valid individual is stored into the population pool until the predifined pool size is reached.

An example of the enumeration process with a phenotype to genotype conversion follows.

Let the production rules set P = {S $\rightarrow$ d A, A $\rightarrow$ e B C, B $\rightarrow$ e, C $\rightarrow$ f}, nonterminals set N = {A,B,C,S}, terminals set T = {d,e,f} and the distinct starting symbol S = {S}.

1. Every symbol is put to an array rule by rule. In this example the array is:

S	d	A	A	e	B	C	B	e	C	f

2. The generated enumeration map becomes:

S	d	e	f	A	B	C
0	1	2	3	4	5	6

3. The phenotype is converted to an enumerated array as:

0	1	4	4	2	5	6	5	2	6	3

Step 3 **Genotype to GNF phenotype**:

The inverse mapping of Step 2 must be estimated. Each genotype must be tesselated in NoR points and mapped to the original CFG in the GNF. The tesselation points are the previous indexes of each terminal symbol in the genotype array and the end of the array due to the special construct of the production rules in the GNF.

Consider the genotype used in step 2. The inverse mapping is estimated as shown next. There are terminal symbols in the indices (1,4,8,10) indexing from zero. Therefor, the tesselation points are (0,3,7,9,end of array). The inverse mapping is summarized in the table bellow.

Intervals	Enumerated Rules	GNF Rules
0 - 2	0 1 4	$S \rightarrow dA$
3 - 6	4 2 5 6	$A \rightarrow eBC$
7 - 8	5 2	$B \rightarrow e$
9 - end	6 3	$C \rightarrow f$

Step 4 **Fitness calculation**: The fitness value is calculated by parsing both positive and negative string databases with each individual GNF in the population.

After parsing, there are 4 possible results. An instance in the positive and the negative database, may succeed or fail to parse in respect to an individual in the population. These four possibilities are also known as true positives (TP), false negatives (FN), false positives (FP) and true negatives (TN) respectively. TPs and TNs contribute +1 to the fitness value of each individual GNF, while FPs contribute -1 and FNs contribute -1 + a - D, whereas a is the sub-parse variable defined in the previous paragraph. D is the ratio of the string that the parser could not reach times a. This means that individuals that succeed parsing on the same number of instances are given a small edge if they tend to parse more instances but 'just' fail. In some experiments a beneficial optional resource was found, rule of thumb alike. If the two databases are differing by more than an order of magnitude in size, the ratio of their size can be used in the FPs, FNs contribution to the fitness (e.g. FP -0.1, FN -10 if the positive dataset is 10 times smaller than the negative one). From the above it is obvious that maximum fitness equals to the size of both datasets and minimum fitness equals to minus the size of both datasets.

An example of the term D calculation follows. Let the production rules set P being the one defined bellow :

Rules
$S \rightarrow dA$
$A \rightarrow eBC$
$B \rightarrow e$
$C \rightarrow f$

with the rest sets of the 4-tuple being, N = {A,B,C}, T = {d,e,f} and S = {S} as in the previous example. If the string deefd was in the positive dataset then the term D should be equal to 0.2 as deef can be parsed by the above grammar but in the string deefd only the four out if the five symbols can be reached. Hence, D should be 0.2. If the candidate string was deefde then D should be 0.333 and so forth.

Step 5 **Convergence check**:
The maximum fitness value of the current population of GNFs is compared with the predefined convergence criterion. If the best individual fitness surpasses the convergence criterion or the maximum number of generations is reached we accept the GNF with maximum fitness value as a solution to the inference problem and the algorithm is terminated.

Step 6 **Evolution**:

Taking into account the plethora of evolutional methods in genetic algorithm implementations, several alternative crossover methods were implemented and are described in the implementation section. The adopted crossover method is efficient due to fast convergence at high fitness values shown in several experiments.

For the crossover, two or three parents are used to derive a new candidate GNF, selected from the pool with respect to their crossover probability which is estimated from the corresponding GNF fitness values following the typical procedure of GAs. In the two parents case, a random offset from $[0, genotype - length]$ is generated for each one. For the first parent a sub-vector from $[0, ..., offset_1]$ is taken and for the second parent sub-vector from $[offset_2, .., genotype - length]$. Then, the two vectors are concatenated in order to generate a new offspring. The offspring is discarded and the process is redone only in three cases; if the resulting offspring is not in the GNF, if the starting symbol S is not found in the head of any production rule or if the derived genotype is bigger than the maximum genotype length defined in step 1. For the three parents case the process of two parents case is done once again with a new parent and the first offspring and the final offspring is considered as the new candidate. The two or three parents cases are equally probable. An example follows. Let two individuals' genotypes selected to produce an offspring $[1, 5, 4, 7, 9, 3, 4, 2, 8]$, $[9, 1, 3, 2, 7, 5, 4]$ and the randomly generated offsets 3,4. Then, from the first parent the $[1, 5, 4]$ and from the second parent the $[7, 5, 4]$ is selected and the the resulting offspring's genotype is the concatenation of the selected sub-strings $[1, 5, 4, 7, 5, 4]$.

The process of crossover is repeated until enough offsprings to populate the next pool are generated. After the crossover step, the mutation operator is applied with probability equal to mutation rate. In the proposed method, a terminal symbol is only mutated to a terminal symbol and a non-terminal symbol is only mutated to a non-terminal symbol so that the GNF is conserved.

Finally, the production rules in the genotype are shuffled. The benefits of this step are analyzed in Paragraph 3.2. This is an exploit that every formal grammar offers as every permutation of the production set P leads to the same formal grammar. This, also implies that multiple genotypes can lead to the same phenotype. In example, consider the GNF from step 2, whereas the production rules set P = {S $\rightarrow$ d A, A $\rightarrow$ e B C, B $\rightarrow$ e, C $\rightarrow$ f}, the non-terminals set N = {A,B,C,S}, the terminals set T = {d,e,f} and the distinct starting symbol S = {S} and the enumeration map as presented in step 3. Below there are depicted some genotypes that lead to the same phenotype.

0	1	4	4	2	5	6	5	2	6	3
4	2	5	6	0	1	4	5	2	6	3
4	2	5	6	5	2	0	1	4	6	3
0	1	4	4	2	5	6	6	3	5	2

The number of genotypes for a corresponding genotype are equal to the number of permutations of the production rules set. In the general case there are $n!$ genotypes whereas n is the number of production rules in a grammar G. This means that the possible shuffles of a genotype as mentioned above are also $n!$.

Step 7 **Iterative loop**:

Return to step 3 in order to re-evaluate a new iteration of the GNF grammatical inference algorithm.

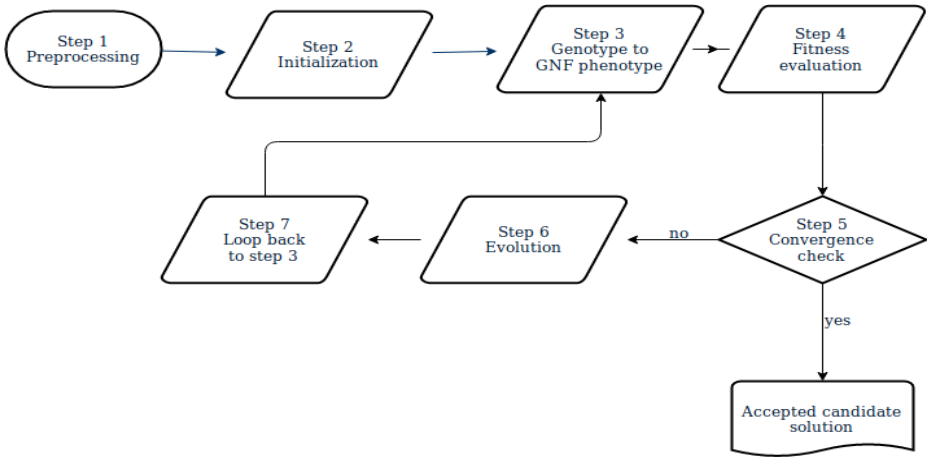

FIGURE 1: Graphical conclusion of the Grammatical Inference algorithm

Proposed parsing method

In order to estimate the fitness of each GNF individual in the proposed algorithm in the previous section, a parsing algorithm must be implemented. As analyzed in Chapter 2, GNF is an LR-free formalism of CFGs and any top down parsing method may be used to solve the parsing problem in at most n steps, whereas n is the length of the candidate word. In this thesis, a custom parsing method was developed as there is no proposed method specific to the GNF, exploiting its properties, in previous works.

The proposed parsing method is carried out in two steps used recurrently. First, the pseudocode for the method is going to be presented and the components of the method will be analyzed after this.

The special notation used for the pseudocode is the sets (S, N, T, P) of a 4-tuple GNF G represented as $(G.S, G.N, G.T, G.P)$ respectively and w is the candidate word to be parsed and consists only of symbols in set T. The notation w_n^m means the substring from index n to index m ($m > n$ and $m <$ length of w). Also, the notations $rule_{NTerminals}$, $rule_{Terminal}$ and $rule_{Head}$ represents the body of the rule without the terminal symbol in the GNF, the terminal symbol of a GNF production rule and the head of a production rule respectively. Each component of the algorithm returns a 2-tuple Parsed,Depth whereas Parsed is a boolean and Depth is an integer. The two steps are called ParseWord and Distribute.

{ Parsed,Depth } = ParseWord(StartSymbol,G,w)
    Minimum Depth = length of w
    for **rule** in $G.P$
        if **rule**$_{Terminal}$ == $w[0]$ & **rule**$_{Head}$ == StartSymbol
            case 1 : { ***rule*** *length = 2 & word length = 1* }
                return { true,0 }
            case 2 : { ***rule*** *length = 3 & word length > 1* }
                Minimum Depth -= 1
                check ParseWord{ **rule**$_{NTerminals}$, G , w_1^{end}}
            case 3 : { ***rule*** *length > 3 & word length > 2* }

```
                    Minimum Depth -= 1
                    check Distribute{ rule_{NTerminals} , G , w_1^{end}}
    end for
    return { False, Minimum Depth }
end ParseWord
```

In the above function, with the instruction check, is meant that if the recurrent call of the function returns true, then the for loop is terminated and the 2-tuple of the called function is returned by the ParseWord{ ... } function.

```
{ Parsed,Depth } = Distribute(array of Non-Terminals,G,w)
    Minimum Depth = length of w
    boolean B_1, B_2
    integer D_1, D_2
    for i in { 1,length of w -1 }
        case 1 : size of array of Non-Terminals = 2
            { B_1, D_1 } = ParseWord( NonTerminals_0, G, w_0^i)
            { B_2, D_2 } = ParseWord( NonTerminals_1, G, w_{i+1}^{end})
            if D_1 + D_2 < Minimum Depth : Minimum Depth = D_1 + D_2
            if B_1 & B_2 return { True, Minimum Depth }
        case 2 : size of array of Non-Terminals > 2
            { B_1, D_1 } = ParseWord( NonTerminals_0, G, w_0^i)
            { B_2, D_2 } = Distribute( NonTerminals_1^{end}, G, w_{i+1}^{end})
            if D_1 + D_2 < Minimum Depth : Minimum Depth = D_1 + D_2
            if B_1 & B_2 return { True, Minimum Depth }
    end for
    return { False, Minimum Depth }
end Distribute
```

In this implementation the two components are called recursively. The stack of recurrent calls may be approximates a DFS tree whereas, the ParseWord function expands the nodes to lower leafs and the Distribute function generates the branches (breadth) of the parsing tree.

In order to parse a word w given a GNF G = (S,N,T,P) the first call of the above method, which will create the root call in the recurrent calls stack, should be ParseWord(S, G, w). The recurrent calling of the two above functions generates a top-down parsing tree. If the word belongs in the corresponding language L(G) then the 2-tuple { True, 0} should will be returned. In any other case, the 2-tuple { False, Depth } would be returned whereas Depth is the number of symbols in the word w that the parser could not reach. Depth is used for the evaluation of the term D in the fitness estimation of each individual.