UNIVERSITÄT
BAYREUTH

University of Bayreuth

Institute for Computer Science

Bachelor Thesis

in Applied Computer Science

Topic:	Randomly Generated CFGs, The CYK-Algorithm And A GUI
Author:	Andreas Braun <andreasbraun5@aol.com> Matrikel-Nr. 1200197
Version date:	February 21, 2017
1. Supervisor:	Prof. Dr. Wim Martens
2. Supervisor:	Tina Trautner

To my parents.

Abstract

The abstract of this thesis will be found here.

Zusammenfassung

Hier steht die Zusammenfassung dieser Bachelorarbeit.

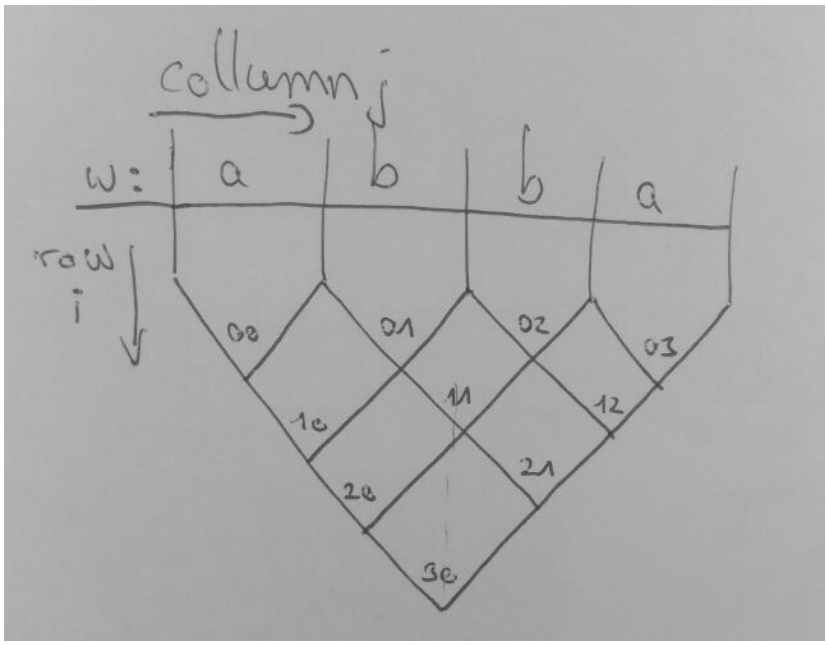
Contents

1 Introduction

2 Technology Overview

3 Algorithms

Analogue to the script:
grammar $G = (V, \Sigma, S, P)$.
V is a finite set of variables.
Σ is an alphabet.
S is the starting symbol and $S \epsilon V$.
P is a finite set of rules: $P \subseteq V \times (V \cup \Sigma)^*$.
LHSE and RHSE are sets that shouldn't be defined. It is possible to have context free grammars(cfgs) with other kind of combinations.
$LHSE$ is finite set of left hand side elements: $LHSE = V$.
$RHSE$ is finite set of right hand side elements: $RHSE = (V \cup \Sigma)^*$.
$G.P.RHSE$ allows access to $RHSE$.
Variables that are of type set begin with a upper case letter.
$p.rhse$ allows access to the one specific $rhse$ of the specific production p.
So P is a finite set of rules: $P \subseteq LHSE \times RHSE$.
Multisets allow duplicate elements. Notation of a multiset is done via the index b: $multiset_b$
pyramid is a structure where one $cell_{i,j}$ holds a set $Cell_{i,j} = \{v \mid v \epsilon V\}$. If initialised each $cell_{i,j}$ holds $Cell_{i,j} = \emptyset$, which is named *empty pyramid*. $cell_{i+1,j} = cellDown$; $cell_{i,j} = cellUpperLeft$; $cell_{i+1,j+1} = cellUpperRight$;
RHSEs are terminals like "a, b, ..." and compound variables like "AB, AC, ...".
LHSEs are variables like "A, B, ...".



Algorithm 1: distributeRhseRandomly

Input: $G,\ Rhse \subseteq\ RHSE,\ 0 \leq minCount \leq maxCount \leq \mid G.V \mid$

Output: *Grammar G with randomly distributed Rhse's.*

```
1 foreach rhse in Rhse do
2     addCount = random(minCount, maxCount);
3     VarsToAddTo = randomSubSet(addCount, LHSE);
4     foreach var in VarsToAddTo do
5         G.P = G.P ∪ {"var ⟶ rhse"};
6     end
7 end
8 return G;
```

.

.

Algorithm 2: GeneratorGrammarDiceRollMartens

Input: $word \in \Sigma^*,\ V,\ \Sigma,\ S,\ P = \emptyset,\ minCount\Sigma,\ maxCount\Sigma,$
$minCountVarComp,\ maxCountVarComp$

Output: G

```
1  G = (V, Σ, S, P);
2  G = distributeRhseRandomly(G, Σ, minCountΣ, maxCountΣ);
3  pyramid = CYK.calculatePyramid(G, word);
4  foreach cell_{i+1,j} in pyramid ∧ i > 0 do
5      VarComp = {XY | X ∈ cellUpperLeft ∧ Y ∈ cellUpperRight};
6      foreach vc in VarComp do
7          while cell_{wordLength+1,0} = ∅ do
8              distributeRhseRandomly(G, vc, minCountVarComp,
               maxCountVarComp);
9              pyramid = CYK.calculatePyramid(G, word);
10         end
11     end
12 end
13 return G;
```

Line 3: Fills the i=0 row of the pyramid.
Line 4: = foreach cellDown, but skipping the first row of the pyramid.

Algorithm 3: checkRightCellCombination

Input: $cellDown \subseteq V$ $cellUpperLeft \subseteq V$, $cellUpperRight \subseteq V$, $G.P$

Output: $varsThatForce \subseteq V$

```
1  isForced = false;
2  VarsThatForce = ∅;
3  VarComp = {XY | X ∈ cellUpperLeft ∧ Y ∈ cellUpperRight};
4  foreach v in cellDown do
5      VProd = {p | p ∈ G.P ∧ p.lhse = v};
6      VRhse = {vRhse | vRhse ∈ VProd.RHSE};
7      if VarComp ⊈ VRhse then
8          isForced = true;
9          VarsThatForce = VarsThatForce ∪ v;
10     end
11 end
12 return VarsThatForce;
```

Algorithm 4: checkRightCellCombinationForced

Input: $pyramid,\ G,\ minCountForced$

Output: G

1 $countForced = 0$;
2 $isForced = true$;
3 $varsThatForce = empty\ pyramid$;
4 **foreach** $cell_{i+1,j}\ in\ pyramid\ \wedge\ i > 1$ **do**
5 $\quad varComp = \{XY \mid X\ \epsilon\ Cell_{i,j}\ \wedge\ Y\ \epsilon\ Cell_{i+1,j+1}\}$;
6 $\quad$;
7 **end**
8 **return** $isForced,\ countForced,\ varsThatForce$;

Line 3: $i > 1 \rightarrow$ the upper two rows aren't included, because they would produce trivial cases that fulfil the restriction always.

```
Algorithm: distributeDiceRollRightHandSideElements
Input: grammar, setRhse, minCount, maxCount, listVars;
Output: grammar;

for(RightHandSideElement rhse : setRhse){
        // countWillBeAdded is between [minCount, maxCount].
        countWillBeAdded = diceRoll();
        while(listVars.size() > countWillBeAdded){
                Remove one variable of listVars via dice roll;
        }
        for(Variable varLeft : listVars){
                // An exception is thrown if the production
                // already exists.
                grammar.addProduction "varLeft --> rhse";
        }
}
return grammar;
```

Listing 1: distributeDiceRollRightHandSideElements

```
Algorithm: distributeDiceRollRightHandSideElements
Input: grammar G, setRhse, minCount, maxCount, setGrammarVars;
Output: grammar;

foreach rhse in setRhse {
        countWillBeAdded = dice roll a number within [minCount, maxCount];
        setVarsToAddRhse = {}dice roll countWillbeAdded to times vars from
                setGrammarVars that get the rhse added to;
        foreach var in setVarsToAddRhse {
                add production "var --> rhse" to grammar;
        }
}
return grammar;
```

Listing 2: distributeDiceRollRightHandSideElementsShort

```
Algorithm: CYK.calculateSetVAdvanced
Input: grammar, word;
Output: Set<VariableK>[][] cYKMatrix;

Set<VariableK>[][] cYKMatrix = new Set<VariableK>[wordSize][wordSize];
cYKMatrix = calculateCYKMatrix;
return cYKMatrix;
```

Listing 3: CYK.calculateSetVAdvanced

```
1  Algorithm: GeneratorGrammarDiceRollTopDownMartens
2  Input: word, settings;
3  Output: grammar;
4  Note: Keep in mind that the setV matrix is a upper right matrix. But the
5  description of how the algorithm works is done, as if the setV pyramid
6  points downwards (reflection on the diagonal + rotation to the left).
7  Regarding one cell, its upper left cell and its upper right cell
8  are looked at. setV[i][j] = down cell. setV[i + 1][j] = upper right cell
9  setV[i][j - 1] = upper left cell. With wordSize = 5, the visited indexes
10 are as following: [01->12->23->34; 02->13->24; 03->14; 04;]
11
12 Grammar grammar = new Grammar();
13 // Part1: Distribute the terminals.
14 grammar = distributeDiceRollRightHandSideElements(
15         grammar, settingsTerminals, minCountTerminals,
16         maxCountTerminals, settingsListVariables);
17 // Part2: Distribute the compound variables.
18 Set<VariableKWrapper>[][] setVAdvanced;
19 // Fill the diagonal of the matrix with:
20 setVAdvanced = CYK.calculateSetVAdvanced( grammar, word );
21 for(Cell cellToBeVisited : setVAdvanced){
22         setVariableCompound = calculate all the possible tupels of
23                 ({varLeft}, {varRight});
24         for(VariableCompound varComp : setVariableCompound){
25                 // Because of dice rolling anyways and lots of grammars
26                 // being generated, no varComp is added if the production
27                 // already exists.
28                 grammar = distributeDiceRollRightHandSideElements(
29                         grammar, varComp, minCountVarComp,
30                         maxCountVarComp, settingsListVariables);
31                 setVAdvanced = CYK.calculateSetVAdvanced( grammar, word );
32         }
33 }
34 return grammar;
35 // Forgot the while cell not empty do this from line 21 to line 33;
```

Listing 4: GeneratorGrammarDiceRollTopDownMartens

```
1  Algorithm: GeneratorGrammarDiceRollOnly
2  Input: settings;
3  Output: grammar;
4  Note: A lot of productions are generated, that later on are not needed
5  for parsing the specific word.
6
7  Grammar grammar = new Grammar();
8  // Part1: Distribute the terminals.
9  grammar = distributeDiceRollRightHandSideElements(
10         grammar, settingsTerminals, minCountTerminals,
11         maxCountTerminals, settingsListVariables);
12 // Part2: Distribute the compound variables.
13 Set<Variables> vars = settings.getVariables();
14 Set<VariablesCompound> setVarComp;
15 setVarComp = calculate all the possible tupels of ({vars}, {vars});
16 grammar = distributeDiceRollRightHandSideElements(
17         grammar, settingsTerminals, minCountVariableCompound,
18         maxCountVariableCompound, setVarComp);
19 return grammar
```

Listing 5: GeneratorGrammarDiceRollOnly

```
1  Algorithm: GeneratorGrammarDiceRollOnlyBias
2  Input: settings;
3  Output: grammar;
4  Note: A lot of productions are generated, that later on are not needed
5  for parsing the specific word.
6
7  Grammar grammar = new Grammar();
8  // Distribute the terminals.
9  grammar = distributeDiceRollRightHandSideElementsBias(
10         grammar, settingsTerminals, settingsMinCountTerminals,
11         settingsMCountTerminals, settingsListVars, settingsFavouritism);
12
13 // Distribute the compound variables.
14 Set<VariablesCompound> setVarComp;
15 setVarComp = calculate all the possible tupels of ({vars}, {vars});
16 grammar = distributeDiceRollRightHandSideElementsBias(
17         grammar, varComp, settingsMinCountVars,
18         settingsMaxCountVars, settingsListVars, settingsFavouritism);
19 return grammar;
```

Listing 6: GeneratorGrammarDiceRollOnlyBias

```
1  Algorithm: distributeDiceRollRightHandSideElementsBias
2  Input: grammar, setRhse, minCount, maxCount, listVars, favouritismList;
3  Output: grammar;
4  Note: Because of dice rolling anyways and lots of grammars being
5  generated, no rhse is added if the production already exists.
6
7  // Calculate the bloated varSet.
8  List<Variable> varsBloated;
9  for(Variables varTemp :  settings.getVariables()){
10         tempFavour = randomly pick favouritism[i];
11         varsBloated.add({tempFavour times varTemp});
12         favouritism.remove(tempFavour);
13 }
14 // Because of dice rolling anyways and lot of grammars being generated,
15 // just no rhse is added if the production already exists.
16 grammar = distributeDiceRollRightHandSideElements( grammar,
17         varsBloated, minCount, maxCount, listVars );
18 return grammar;
```

Listing 7: distributeDiceRollRightHandSideElementsBias

Algorithm 5: distributeDiceRollRightHandSideElementsBias

Input: G, $rhse \subseteq RHSE$, $0 \leq minCount \leq maxCount \leq |G.V|$, $favouritism = \{x \mid x \in \mathrm{N} \wedge |favouritism| = |G.V|\}$

Output: G

1 $favour = \{(v, f) \mid v \in V \wedge f \in favouritism \wedge$ *tupel are created via dice roll*$\}$;

2 $varsBloated_b = \emptyset$;

3 **foreach** *fav in favour* **do**

4 $\quad varsBloated_b = varsBloated_b \uplus \{v^f\}$;

5 **end**

6 **return** $distributeDiceRollRhse(G, MISTAKEHEREvarsBloated_b, minCount, maxCount)$;

7 Still working on. One more Pprameter needed for distributeDiceRollRighthandSideElement. Parameter V that defines the variables the rhse are added to. OR make this algorithm independent.

Line 6: Note that $varsBloated_b$ is a multiset, but should actually be a set. Exceptions causing a duplicate production to the grammar are not relevant because G.P is a set.

Description of the checks here.
All test of the GrammarValidityChecker class are based on the simple setV matrix.

isValid = isWordProducible && isExamConstraints && isGrammarRestrictions

isWordProducible = CYK.algorithmAdvanced()

isExamConstraints = isRightCellCombinationsForced && isMaxSumOfProductionsCount && isMaxSumOfVarsInPyramidCount && countRightCellCombinationsForced

isGrammarRestrictions = isSizeOfWordCount && isMaxNumberOfVarsPerCellCount

```
1 Algorithm: checksumOfProductions
2 Input: grammar, maxSumOfProduction;
3 Output: isSumOfProductions;
4
5 return grammar.getProductionsAsList().size() <= maxSumOfProductions;
```

Listing 8: checksumOfProductions

```
1  Algorithm: checkMaxNumberOfVarsPerCell
2  Input: setVSimple, maxNumberOfVarsPerCell;
3  Output: isMaxNumberOfVarsPerCell;
4  Note: Checking for maxNumberOfVarsPerCell <= zero isn't allowed;
5
6  int tempMaxNumberOfVarsPerCell = 0;
7  int wordLength = tempSetV[0].length;
8  for ( int i = 0; i < wordLength; i++ ) {
9          for ( int j = 0; j < wordLength; j++ ) {
10                 if ( tempSetV[i][j].size() > numberOfVarsPerCell ) {
11                         numberOfVarsPerCell = tempSetV[i][j].size();
12                 }
13         }
14 }
15 return tempMaxNumberOfVarsPerCell <= maxNumberOfVarsPerCell;
```

Listing 9: checkMaxNumberOfVarsPerCell

```
1  Algorithm: checkMaxSumOfVarsInPyramid
2  Input: setVSimple, maxSumOfVarsInPyramid;
3  Output: isMaxSumOfVarsInPyramid;
4
5  // put all vars of the matrix into one list and use its length.
6  List<Varaible> allVarsList = new ArrayList<>();
7  for ( int i = 0; i < setVSimple.length; i++ ) {
8          for ( int j = 0; j < setVSimple.length; j++ ) {
9                  tempVars.addAll( setVSimple[i][j] );
10         }
11 }
12 return allVarsList.size() <= maxSumOfVarsInPyramid;
```

Listing 10: checkMaxSumOfVarsInPyramid

```
Algorithm: rightCellCombinationsForced
Input: setVSimple, minCountForced, grammar;
Output: isForced, countForced, setVSimpleVarsThatForce;
Note: Keep in mind that the setV matrix is a upper right matrix. But the
description of how the algorithm works is done, as if the setV pyramid
points downwards (reflection on the diagonal + rotation to the left).
Regarding one cell, its upper left cell and its upper right cell
are looked at. setV[i][j] = down cell. setV[i + 1][j] = upper right cell
setV[i][j - 1] = upper left cell.

int countForced = 0;
Set<Variable>[][] setVMarkedVarsThatForce;
for(Cell cell : setVSimple){
        // Trivial cases that would fulfil the restrictions each time.
        Ignore the upper two rows of the pyramid;
        isRightCellCombinationForced = true;
        if(!upperLeftCell.isEmpty() && !upperRightCell.isEmpty()) {
                break;
        }
        setVariableCompound = calculate all the possible tupels of
                ({varLeft}, {varRight});
        for(Variable var : cellToBeVisited) {
                varDownProdList = grammar.getProdList(varDown);
                for(VariableCompound varComp : setVariableCompound) {
                        for(Production prod : varDownProdList){
                                if(prod.getRhse() == varComp) {
                                        isForced = false;
                                }
                        }
                }
                if(isRightCellCombinationForced) {
                        rightCellCombinationsForced++;
                        // Cell has index i and j.
                        setVMarkedVarsThatForce[i][j].add(var)
                }
        }
}
boolean isForced = countForced >= minCountRightCellCombinationsForced;
return isForced, countForced, setVMarkedVarsThatForce;
```

Listing 11: rightCellCombinationsForced

```
1  Algorithm: Util.removeUselessProductions
2  Input: grammar, setVSimple, word
3  Output: grammar
4  Note: Very similar to the calculateSetVAdvanced algorithm. Additionally
5  to storing the k, it is also saved, which production have been used.
6  All productions that haven't been need are removed, from the grammar.
7
8  Set<Production> allProductions = grammar.getProductions();
9  Set<Production> onlyUsefulProductions;
10 onlyUsefulProductions = calculate useful productions with the input of
11         grammar, setVSimple and word ;
12 grammar.remove(allProductions);
13 return grammar.add(onlyUsefulProductions);
```

Listing 12: Util.removeUselessProductions

4 Course of Action

The informal goal is to find a suitable combination of a grammar and a word that meets the demands of an exam exercise. Also the CYK pyramid and one derivation tree of the word must be generated automatically as a "solution picture".
Firstly the exam exercise must have an upper limit of variables per cell while computing the CYK-pyramid.
Secondly the exam exercise must have one or more "special properties" so that it can be checked if the students have clearly understood the algorithm, e.g. "Excluding the possibility of luck."

The more formal goal is identify and determine parameters that in general can be used to define the properties of a grammar, so that the demanded restrictions are met. Also parameters could be identified for words, but "which is less likely to contribute, than the parameters of the grammar." [Second appointment with Martens]

Some introductory stuff:

Possible basic approaches for getting these parameters are the Rejection Sampling method and the "Tina+Wim" method.
Also backtracking plays some role, but right now I don't know where to put it. Backtracking is underapproach to Rejection Sampling.
Note: Starting with one half of a word and one half of a grammar.

Identify restrictions (=parameters) regarding the grammar.
Maybe find restrictions regarding the words, too.

Procedures for automated generation. Each generation procedure considers different restrictions and restriction combinations. The restrictions within one generation procedure can be optimized on its own. Up till now:
Generating grammars: DiceRolling, ...
Generating words: DiceRolling, ...

Parameter optimisation via theoretical and/or benchmarking approach.
Benchmarking = generate N grammars and test them, (N=100000).
Define a success rate and try to increase it.

The overall strategy is as following:

1.) Identify theoretically a restriction/parameter for the grammar. Think about the influence it will have. Think also about correlations between the restrictions.
2.) Validate the theoretical conclusion with the benchmark. Test out the influence of this parameter upon the success rate. Try different parameter settings.

The ordering of step 1 and step 2 can be changed.

Used software

Listings

Following are some interesting classes referenced in the thesis that were too long to fit into the text.

Tables

This section contains all tables referenced in this thesis.

References

[1] JSR 220: Enterprise Java Beans 3.0 https://jcp.org/en/jsr/detail?id=220, 09/09/2015