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 23, 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.

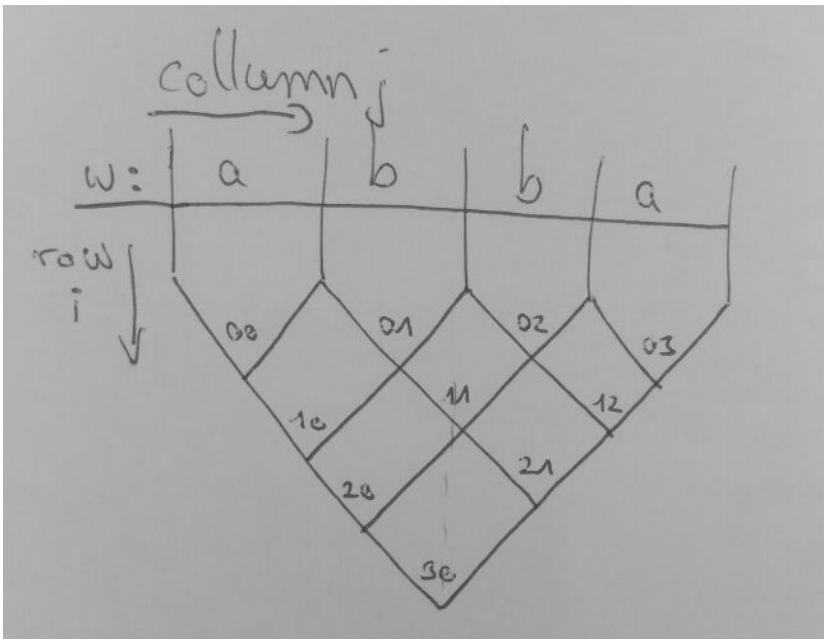
Contents

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 $S \in V$.
P is a finite set of rules: $P \subseteq V \times (V \cup \Sigma)^*$.
G is in CNF and therefore it holds, more specifically: $P\ \subseteq\ V \times\ (V^2 \cup \Sigma)$. Think about if it is ok.
Vs are Variables like "A, B, ...".
$(V \cup\ \Sigma)^*$ are terminals like "a, b, ..." and compound variables like "AB, BS, AC, ...".
$i, j \in \mathbb{N};\ [i,\ j] := \{i,\ i+1, ..., j-1,\ j\} \subseteq \mathbb{N}_{\geq 0}$. $word \in \Sigma^*$
$Pyramid := \{cell_{i,j} \mid i \in \mathbb{N}_{\geq 0},\ j \in [0,\ j_{max} - i],\ i_{max} = j_{max} = |word| - 1\}$.
$cell_{i,j} = \{Cell_{i,j} \mid Cell_{i,j} \subseteq V\}$. If each $cell_{i,j} = \emptyset$ then it is called $EmptyPyramid$.
Note that regarding one $cell_{i,j}$: $cell_{i,j} = cellDown$, $cell_{i-1,j} = cellUpperLeft$ and $cell_{i-1,j+1} = cellUpperRight$
Multisets allow duplicate elements. Notation of a multiset is done via the index b, that stands for bag: $multiset_b$



Algorithm 1: distributeRhse

Input: $G,\ Rhse \subseteq\ (V^2 \cup \Sigma),\ i,\ j$

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

foreach $rhse \in Rhse$ **do**

    choose n *uniform randomly in* $[i, j]$;

    choose $V_{add} :=$ *uniform random subset of size* n *from* V;

    $G.P = G.P\ \cup\ \{"v \longrightarrow rhse" \mid v \in V_{add}\}$;

end

return G;

Algorithm 2: calculateSubsetForCell

Input: $Pyramid,\ cell_{i,j}$

Output: $V_{i,j} \subseteq V^2$

$V_{i,j} = \emptyset$;

for $m := i - 1 \rightarrow 0$ **do**

    $V_{i,j} = V_{i,j} \cup \{X \mid X \longrightarrow YZ,\ Y \in V_{m,j},\ Z \in V_{i-m-1,m+j+1}\}$;

end

return $V_{i,j}$;

Algorithm 3: checkForceCombinationForCell

Input: $cell_{i,j} \subseteq V^2\, cell_{i-1,j} \subseteq V^2,\ cell_{i-1,j+1} \subseteq V^2,\ G.P$

Output: $varsForcing \subseteq V$

$varsForcing = \emptyset$;

$varComp = \{XY \mid X \in cell_{i-1,j}\ \wedge\ Y \in cell_{i-1,j+1}\}$;

foreach $v \in cellDown$ **do**

    $prods = \{p \mid v \in G.P.V\}$;

    $rhses = \{rhse \mid rhse \in prods.(V^2 \cup \Sigma)\}$;

    if $varComp \not\subseteq rhses$ **then**

        $varsForcing = varsForcing\ \cup\ v$;

    end

end

return $varsForcing$;

Input: $cell_{i,j} = cellDown$, $cell_{i-1,j} = cellUpperLeft$ and $cell_{i-1,j+1} = cellUpperRight$

Algorithm 4: GeneratorGrammarDiceRollMartens

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

Output: G

1 $G = (V, \Sigma, S, P);$
2 $G = distributeRhse(G,\ \Sigma,\ minCount\Sigma,\ maxCount\Sigma);$
3 $Pyramid = CYK.calculatePyramid(G,\ word);$
4 **for** $i := 1\ \rightarrow\ i_{max}$ **do**
5   **for** $j := 0\ \rightarrow\ j_{max} - i$ **do**
6         $sub = calculateSubsetForCell(Pyramid,\ i,\ j);$
7   **foreach** $vc \in sub$ **do**
8   **while** $cell_{i_{max},0} = \emptyset$ **do**
9                 $distributeRhse(G,\ vc,\ minCountVarComp,$ $maxCountVarComp);$
10                 $Pyramid = CYK.calculatePyramid(G,\ word);$
11   **end**
12   **end**
13   **end**
14 **end**
15 **return** G;

Line 3: Fills the i=0 row of the pyramid.
line 5: Instead of going from left to right, choose j uniform randomly with the restrictions that one cell is only visited one time.
Note: The algorithm tends to finish already within $i = 1$ loop.

Algorithm 5: GeneratorGrammarDiceRollMartens2

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

Output: G

1 $G = (V, \Sigma, S, P)$;
2 $G = distributeRhse(G,\ \Sigma,\ minCount\Sigma,\ maxCount\Sigma)$;
3 $Pyramid = CYK.calculatePyramid(G,\ word)$;
4 $sub = \emptyset$;
5 **for** $i := 1\ \rightarrow\ i_{max}$ **do**
6   **for** $j := 0\ \rightarrow\ j_{max} - i$ **do**
7         $sub = sub \cup \{(A, i) \mid A \in calculateSubsetForCell(Pyramid,\ i,\ j)\}$;
8   **end**
9     $sub_b = \{B \mid B \in sub,\ sub_b\ models\ i\text{-}dependent\ priority\ mechanism\}$;
10   **while** $cell_{i_{max},0} = \emptyset\ \wedge\ threshold_i = false$ **do**
11         $choose\ one\ vc\ uniform\ randomly \in sub_b$;
12         $distributeRhse(G,\ vc,\ minCountVarComp,\ maxCountVarComp)$;
13         $Pyramid = CYK.calculatePyramid(G,\ word)$;
14         $evaluate\ and\ update\ threshold_i$;
15   **end**
16 **end**
17 **return** G;

Line 3: Fills the i=0 row of the pyramid.
Line 7: $(AB, 1), (AB, 2), (BC, 3)... \in sub \rightarrow$ multiple occurrences of AB are allowed. This considers "more important" compound variables.
Line 11: One vc can be chosen several times.
Note: Threshold: Linear or log function $f(i)$?
Note: Priority mechanism: In line $i + 1$ the $k = \{(A, l) \mid (A, l) \in sub,\ l = i\}$ are preferred over the $m = \{(A, n) \mid (A, n) \in sub,\ n < i\}$. In what way are they preferred? Using some kind of factor to weight the i of (A, i).

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

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

Listing 1: CYK.calculateSetVAdvanced

```
Algorithm: GeneratorGrammarDiceRollOnly
Input: settings;
Output: grammar;
Note: A lot of productions are generated, that later on are not needed
for parsing the specific word.

Grammar grammar = new Grammar();
// Part1: Distribute the terminals.
grammar = distributeDiceRollRightHandSideElements(
          grammar, settingsTerminals, minCountTerminals,
          maxCountTerminals, settingsListVariables);
// Part2: Distribute the compound variables.
Set<Variables> vars = settings.getVariables();
Set<VariablesCompound> setVarComp;
setVarComp = calculate all the possible tupels of ({vars}, {vars});
grammar = distributeDiceRollRightHandSideElements(
          grammar, settingsTerminals, minCountVariableCompound,
          maxCountVariableCompound, setVarComp);
return grammar
```

Listing 2: GeneratorGrammarDiceRollOnly

```
Algorithm: GeneratorGrammarDiceRollOnlyBias
Input: settings;
Output: grammar;
Note: A lot of productions are generated, that later on are not needed
for parsing the specific word.

Grammar grammar = new Grammar();
// Distribute the terminals.
grammar = distributeDiceRollRightHandSideElementsBias(
          grammar, settingsTerminals, settingsMinCountTerminals,
          settingsMCountTerminals, settingsListVars, settingsFavouritism);

// Distribute the compound variables.
Set<VariablesCompound> setVarComp;
setVarComp = calculate all the possible tupels of ({vars}, {vars});
grammar = distributeDiceRollRightHandSideElementsBias(
          grammar, varComp, settingsMinCountVars,
          settingsMaxCountVars, settingsListVars, settingsFavouritism);
return grammar;
```

Listing 3: GeneratorGrammarDiceRollOnlyBias

```
Algorithm: distributeDiceRollRightHandSideElementsBias
Input: grammar, setRhse, minCount, maxCount, listVars, favouritismList;
Output: grammar;
Note: Because of dice rolling anyways and lots of grammars being
generated, no rhse is added if the production already exists.

// Calculate the bloated varSet.
List<Variable> varsBloated;
for(Variables varTemp :  settings.getVariables()){
        tempFavour = randomly pick favouritism[i];
        varsBloated.add({tempFavour times varTemp});
        favouritism.remove(tempFavour);
}
// Because of dice rolling anyways and lot of grammars being generated,
// just no rhse is added if the production already exists.
grammar = distributeDiceRollRightHandSideElements( grammar,
        varsBloated, minCount, maxCount, listVars );
return grammar;
```

Listing 4: distributeDiceRollRightHandSideElementsBias

Algorithm 6: distributeDiceRollRightHandSideElementsBias

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

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

```
Algorithm: checksumOfProductions
Input: grammar, maxSumOfProduction;
Output: isSumOfProductions;

return grammar.getProductionsAsList().size() <= maxSumOfProductions;
```

Listing 5: checksumOfProductions

```
Algorithm: checkMaxNumberOfVarsPerCell
Input: setVSimple, maxNumberOfVarsPerCell;
Output: isMaxNumberOfVarsPerCell;
Note: Checking for maxNumberOfVarsPerCell <= zero isn't allowed;

int tempMaxNumberOfVarsPerCell = 0;
int wordLength = tempSetV[0].length;
for ( int i = 0; i < wordLength; i++ ) {
        for ( int j = 0; j < wordLength; j++ ) {
                if ( tempSetV[i][j].size() > numberOfVarsPerCell ) {
                        numberOfVarsPerCell = tempSetV[i][j].size();
                }
        }
}
return tempMaxNumberOfVarsPerCell <= maxNumberOfVarsPerCell;
```

Listing 6: checkMaxNumberOfVarsPerCell

```
Algorithm: checkMaxSumOfVarsInPyramid
Input: setVSimple, maxSumOfVarsInPyramid;
Output: isMaxSumOfVarsInPyramid;

// put all vars of the matrix into one list and use its length.
List<Varaible> allVarsList = new ArrayList<>();
for ( int i = 0; i < setVSimple.length; i++ ) {
        for ( int j = 0; j < setVSimple.length; j++ ) {
                tempVars.addAll( setVSimple[i][j] );
        }
}
return allVarsList.size() <= maxSumOfVarsInPyramid;
```

Listing 7: 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 8: rightCellCombinationsForced

```
Algorithm: Util.removeUselessProductions
Input: grammar, setVSimple, word
Output: grammar
Note: Very similar to the calculateSetVAdvanced algorithm. Additionally
to storing the k, it is also saved, which production have been used.
All productions that haven't been need are removed, from the grammar.

Set<Production> allProductions = grammar.getProductions();
Set<Production> onlyUsefulProductions;
onlyUsefulProductions = calculate useful productions with the input of
        grammar, setVSimple and word ;
grammar.remove(allProductions);
return grammar.add(onlyUsefulProductions);
```

Listing 9: 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